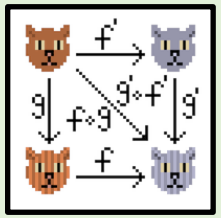
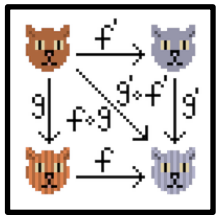
N-Queens Combinatorial Puzzle meets Cats

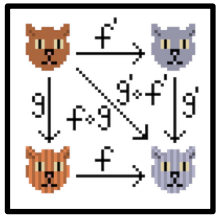
Haskell

monadic mapping, filtering, folding
- $mapM$
- $filterM$
- $foldM$

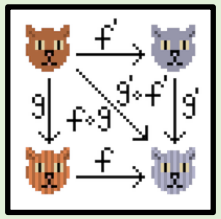


Scala





monoidal functions $fold$ and $foldMap$



If you decide you like this, you can find more content like it on http://fpilluminated.com

FP Illuminated

Some of the other places where you can find me

https://twitter.com/philip_schwarz

https://fosstodon.org/@philip_schwarz

https://github.com/philipschwarz

https://www.linkedin.com/in/philip-schwarz-70576a17/

https://speakerdeck.com/philipschwarz

https://www.slideshare.net/pjschwarz

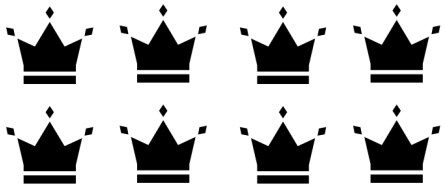
Eight queens puzzle

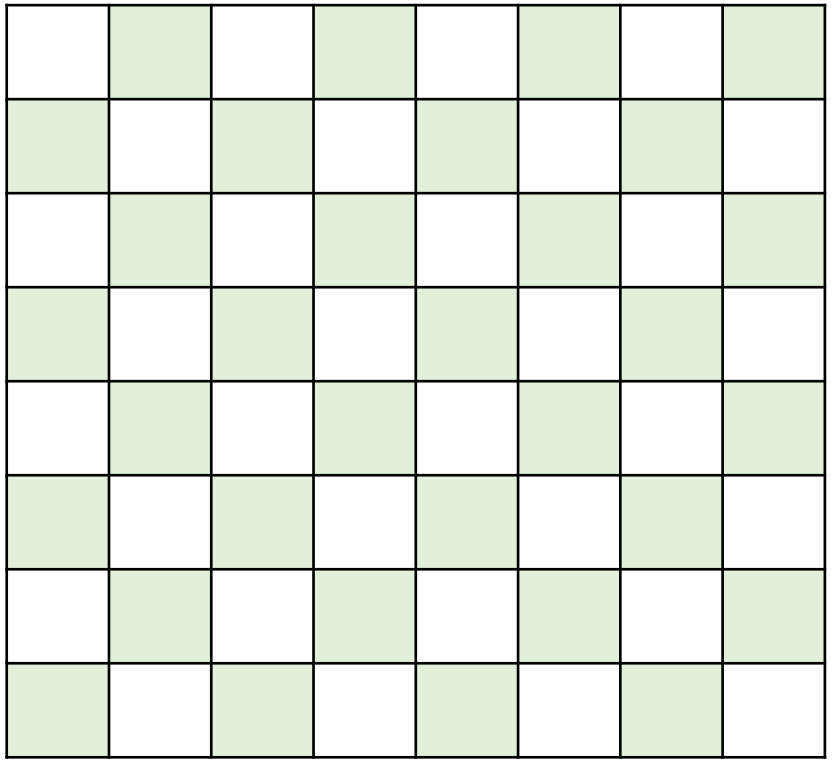
Article Talk

From Wikipedia, the free encyclopedia

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

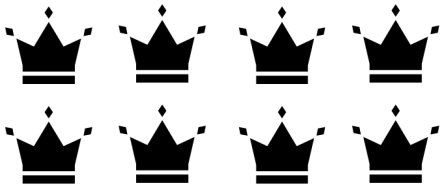
An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

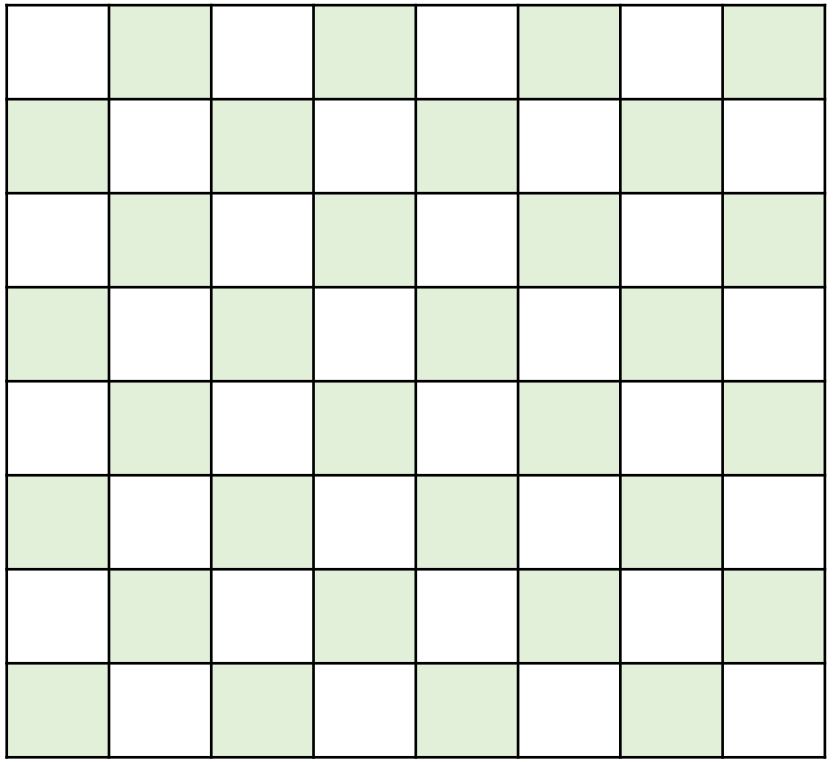




An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

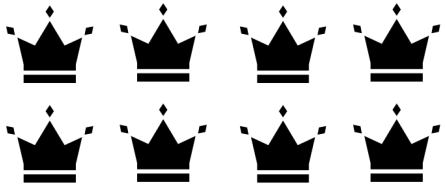


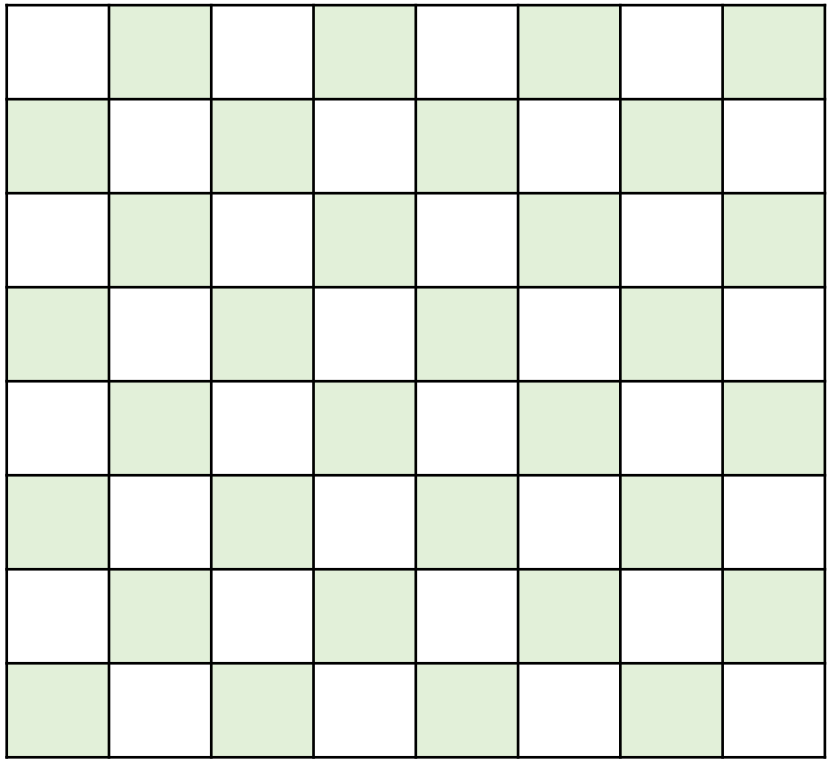


An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$



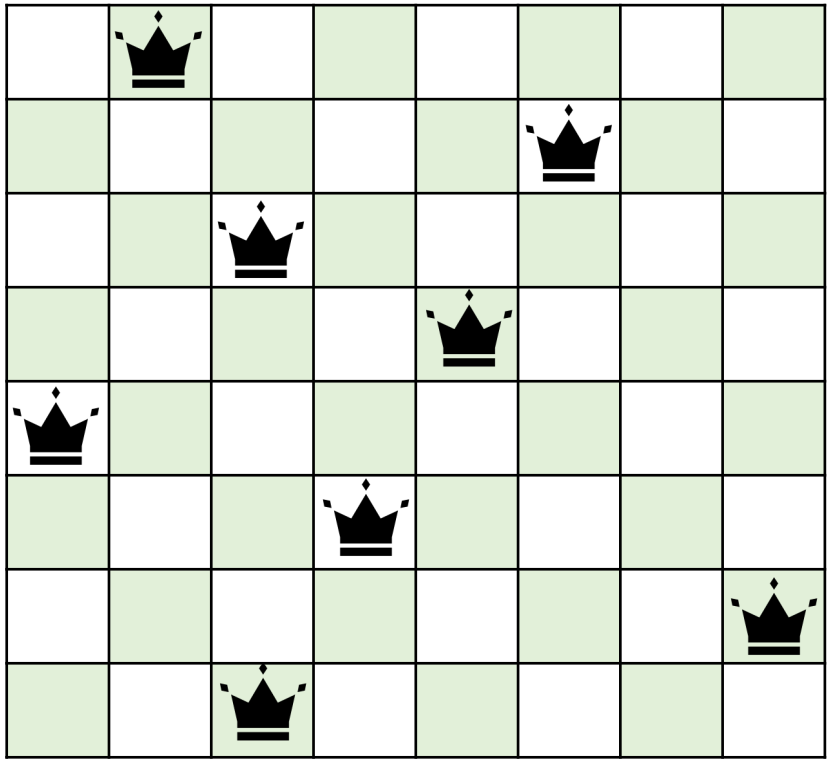


An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?



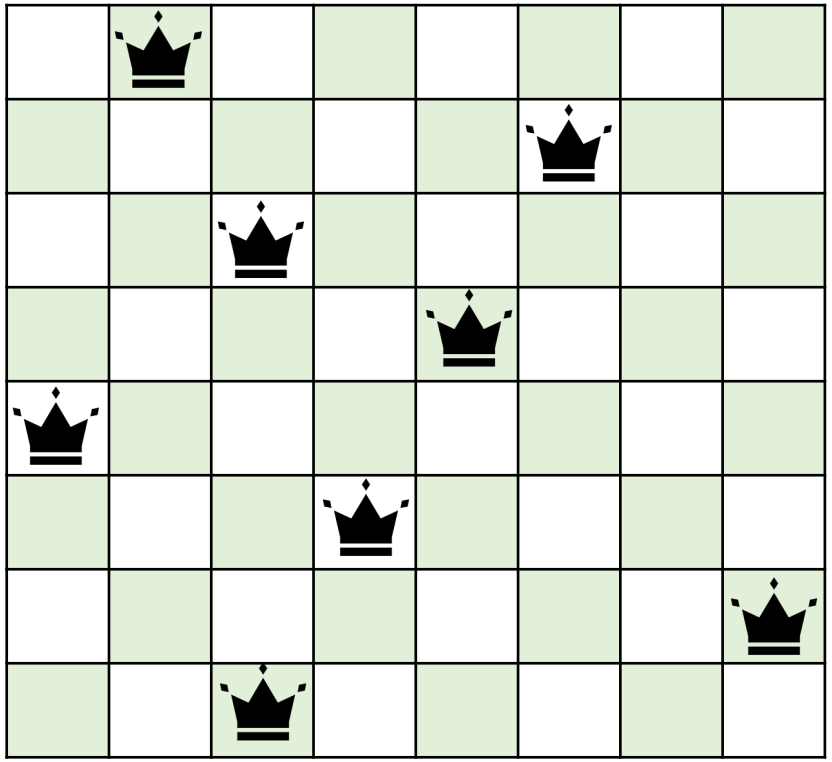
An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th



An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

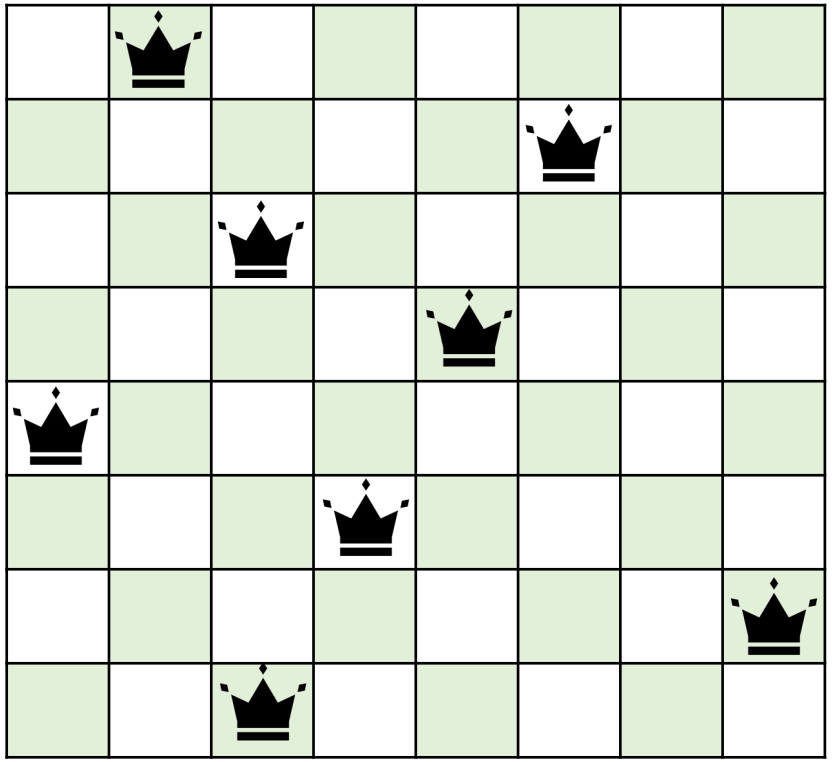
For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$



An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

$$\frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4{,}426{,}165{,}368$$

An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

$$\frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4{,}426{,}165{,}368$$

We know that no more than one queen is allowed on each row, since multiple queens on the same row hold each other in check.

We can use that fact to drastically reduce the number of candidate boards.

An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

$$\frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4{,}426{,}165{,}368$$

We know that no more than one queen is allowed on each row, since multiple queens on the same row hold each other in check.

We can use that fact to drastically reduce the number of candidate boards.

To do that, instead of picking 8 cells on the whole board, we consider each of 8 rows in turn, and on each such row, we pick one of 8 columns.

$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16{,}777{,}216$

An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

$$\frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4{,}426{,}165{,}368$$

We know that no more than one queen is allowed on each row, since multiple queens on the same row hold each other in check.

We can use that fact to drastically reduce the number of candidate boards.

To do that, instead of picking 8 cells on the whole board, we consider each of 8 rows in turn, and on each such row, we pick one of 8 columns.

$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16{,}777{,}216$

We also know that no more than one queen is allowed on each column, so we can again reduce the number of candidate boards.

An 8 x 8 chessboard has 64 cells. How many ways are there of placing 8 queens on the board?

For the 1st queen, we have a choice of 64 cells
63 for the 2nd queen
62 for the 3rd
61 for the 4th
60 for the 5th
59 for the 6th
58 for the 7th
57 for the 8th

$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 178{,}462{,}987{,}637{,}760$

Given a particular placement of 8 queens, how many ways are there of arriving at it? We pick one cell at a time, in sequence, so how many different possible sequences are there for picking 8 cells?

For our first pick, we have 8 choices.
7 for the 2nd
6 for the 3rd
5 for the 4th
4 for the 5th
3 for the 6th
2 for the 7th
1 for the 8th

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

So the number of candidate board configurations for the puzzle is

$$\frac{\text{Number of ways of placing 8 queens on the board}}{\text{Number of ways of arriving at each configuration}}$$

$$\frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4{,}426{,}165{,}368$$

We know that no more than one queen is allowed on each row, since multiple queens on the same row hold each other in check.

We can use that fact to drastically reduce the number of candidate boards.

To do that, instead of picking 8 cells on the whole board, we consider each of 8 rows in turn, and on each such row, we pick one of 8 columns.

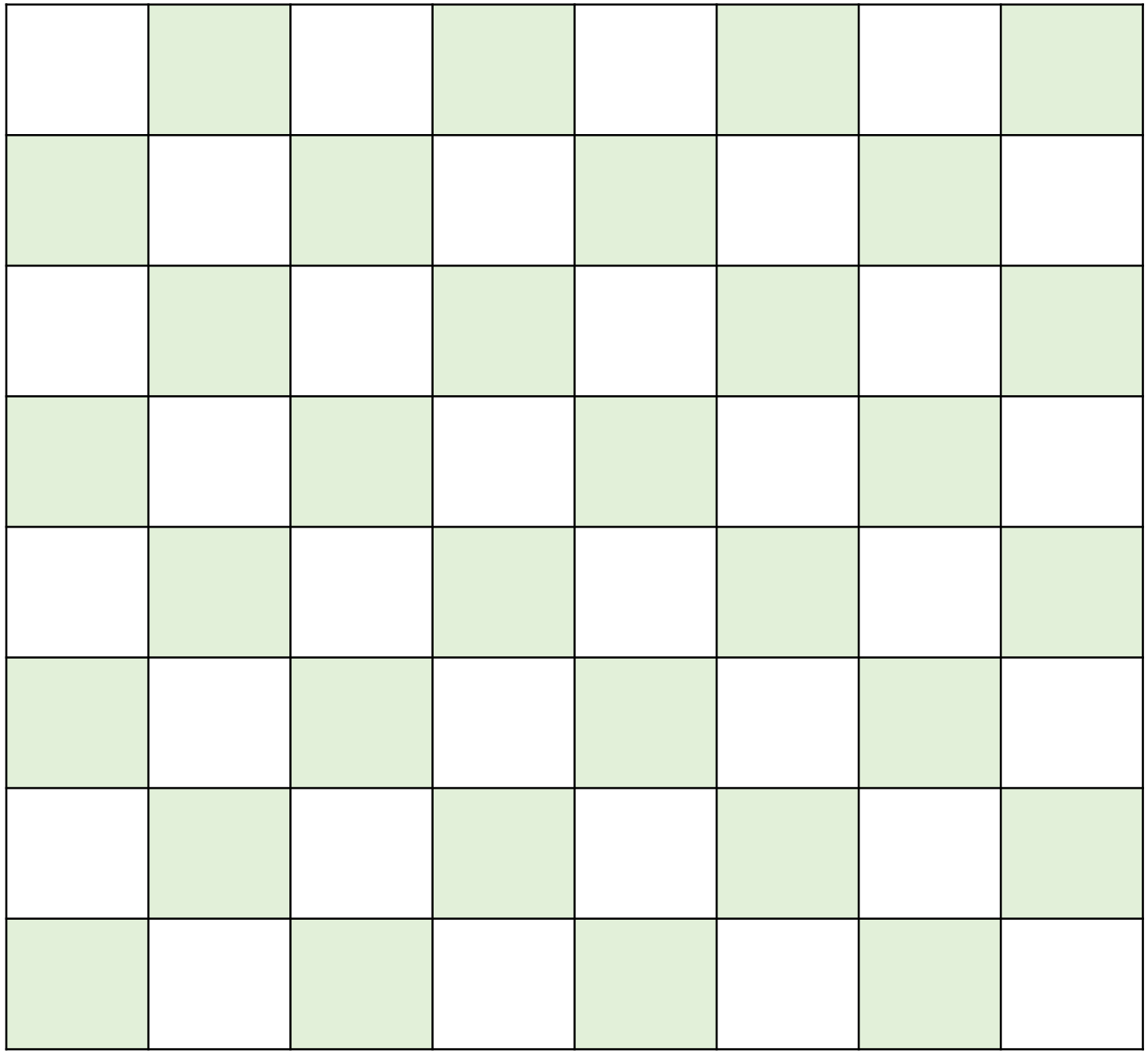
$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16{,}777{,}216$

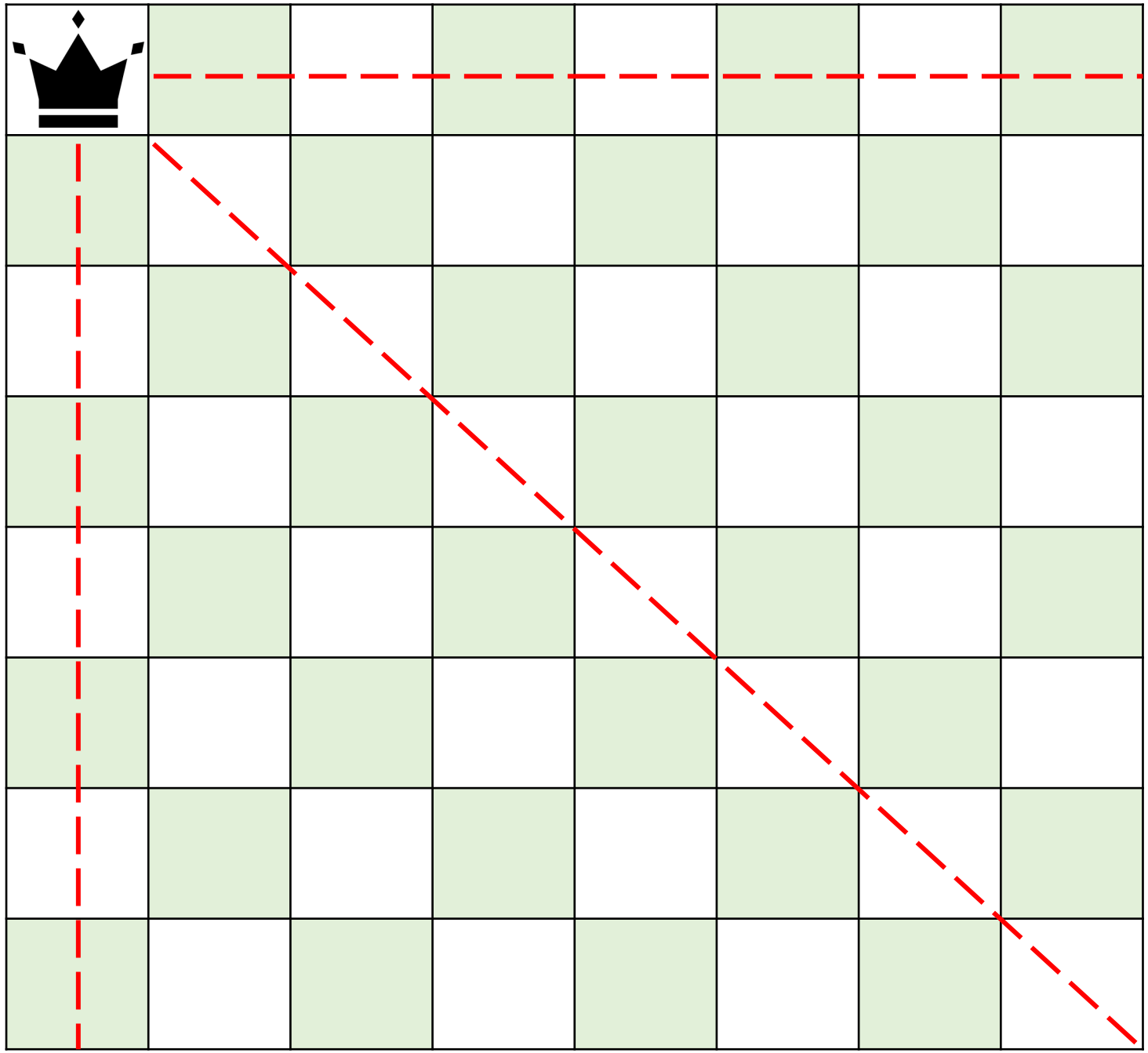
We also know that no more than one queen is allowed on each column, so we can again reduce the number of candidate boards.

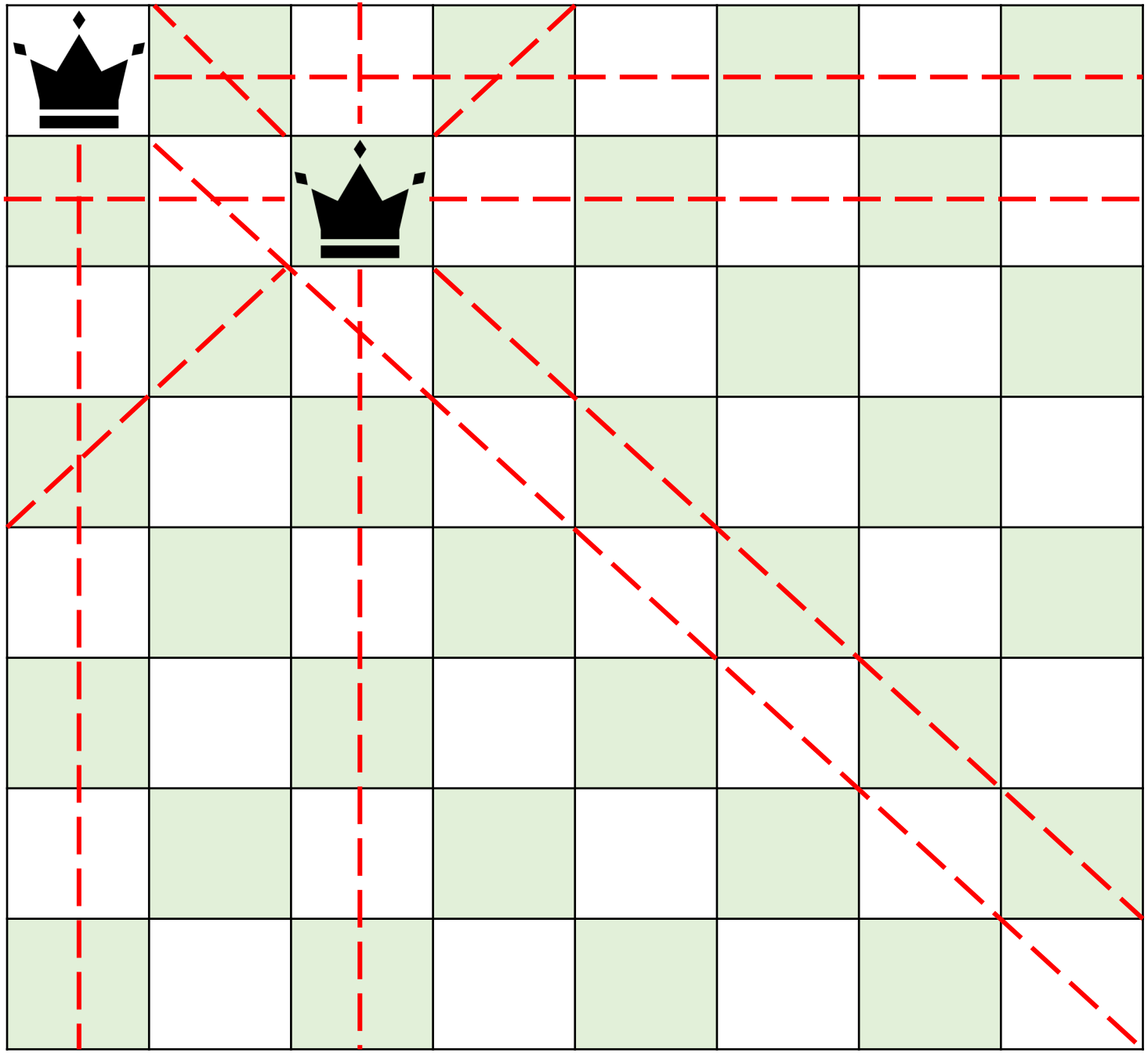
If we pick a column on one row, we cannot pick it again on subsequent rows, i.e. the choice of columns that we can pick decreases as we progress through the rows.

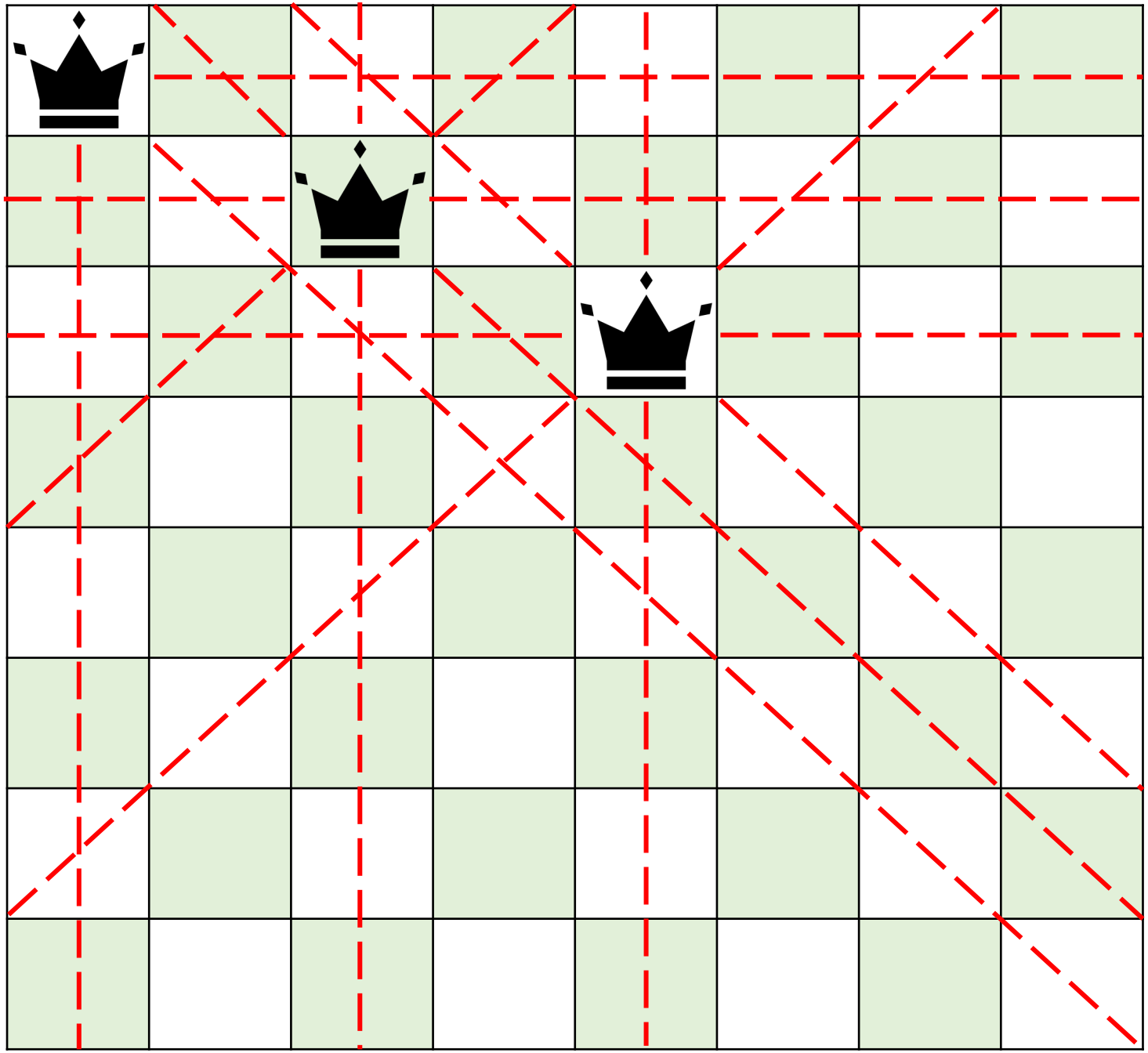
$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$

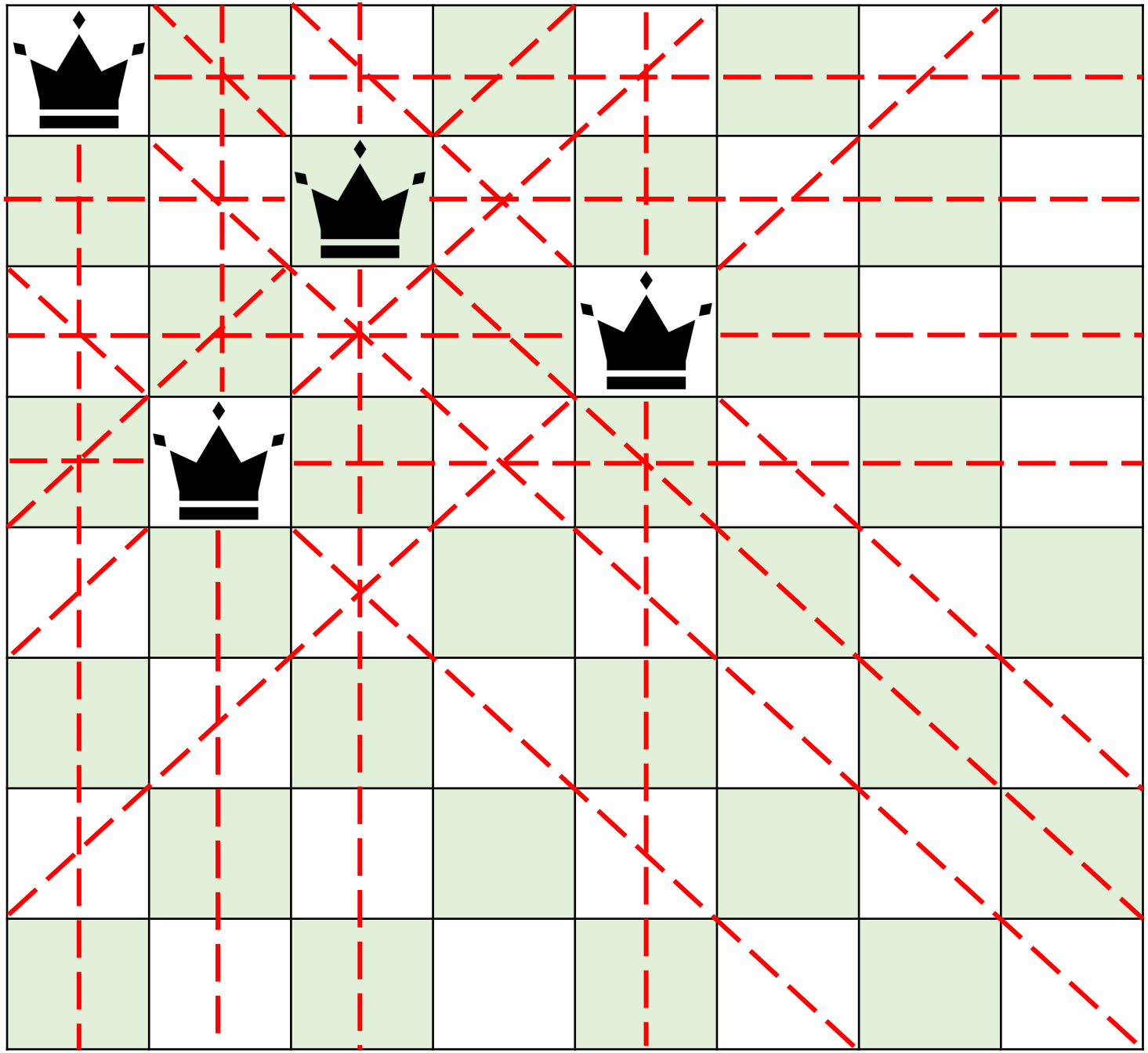
Let’s try to find a solution to the puzzle

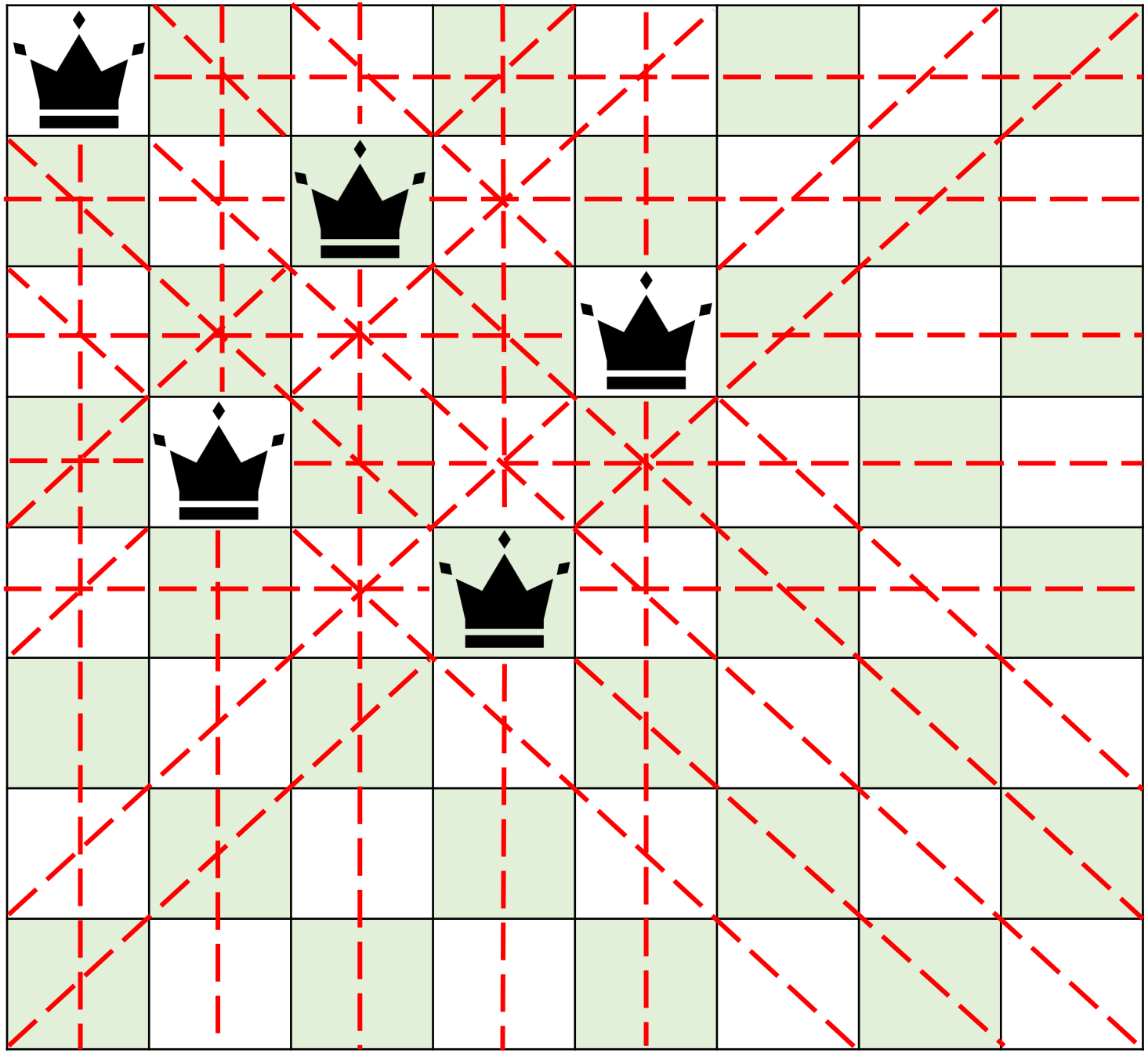


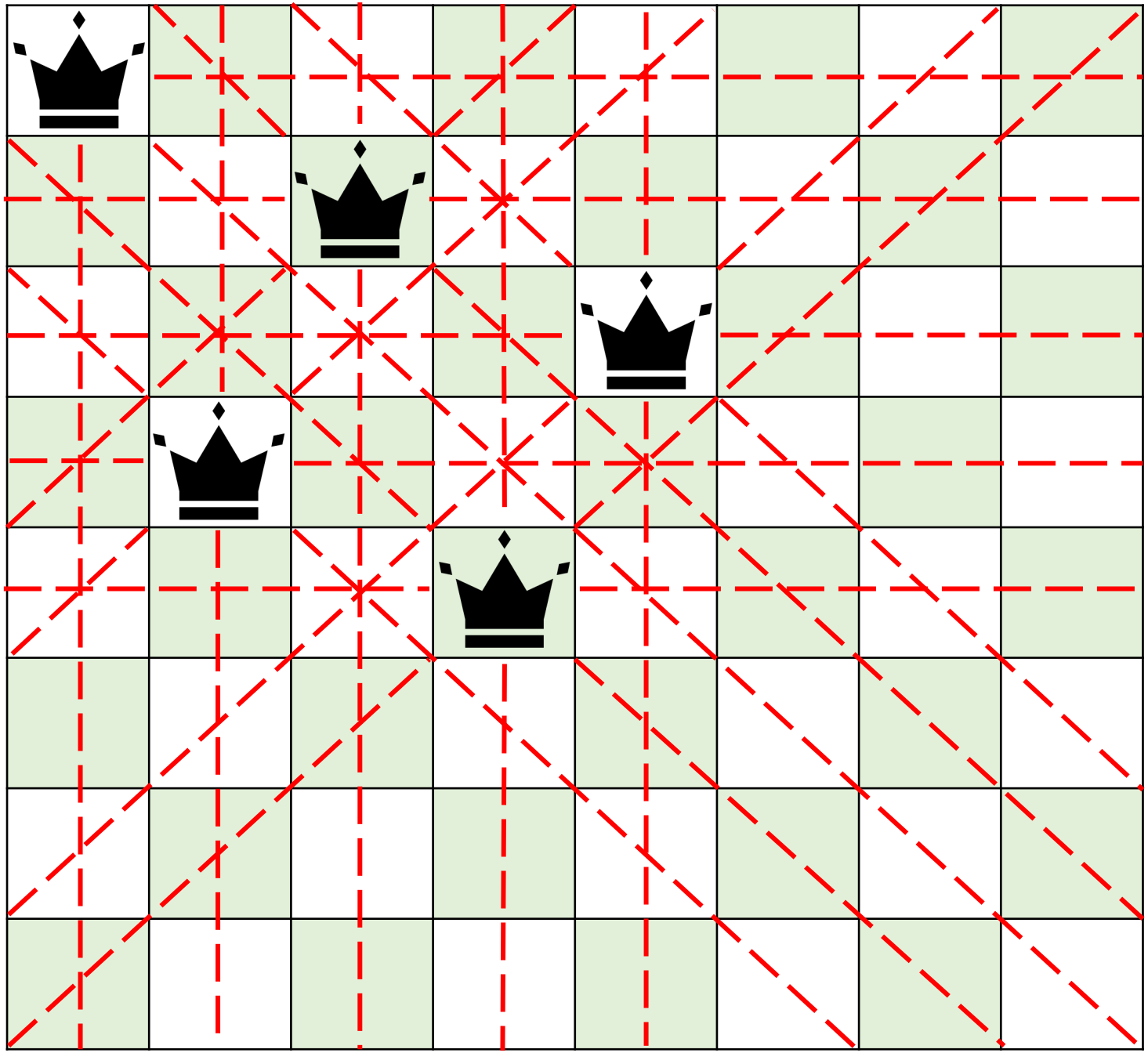


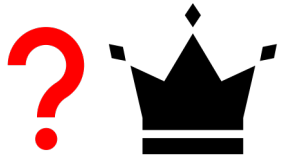






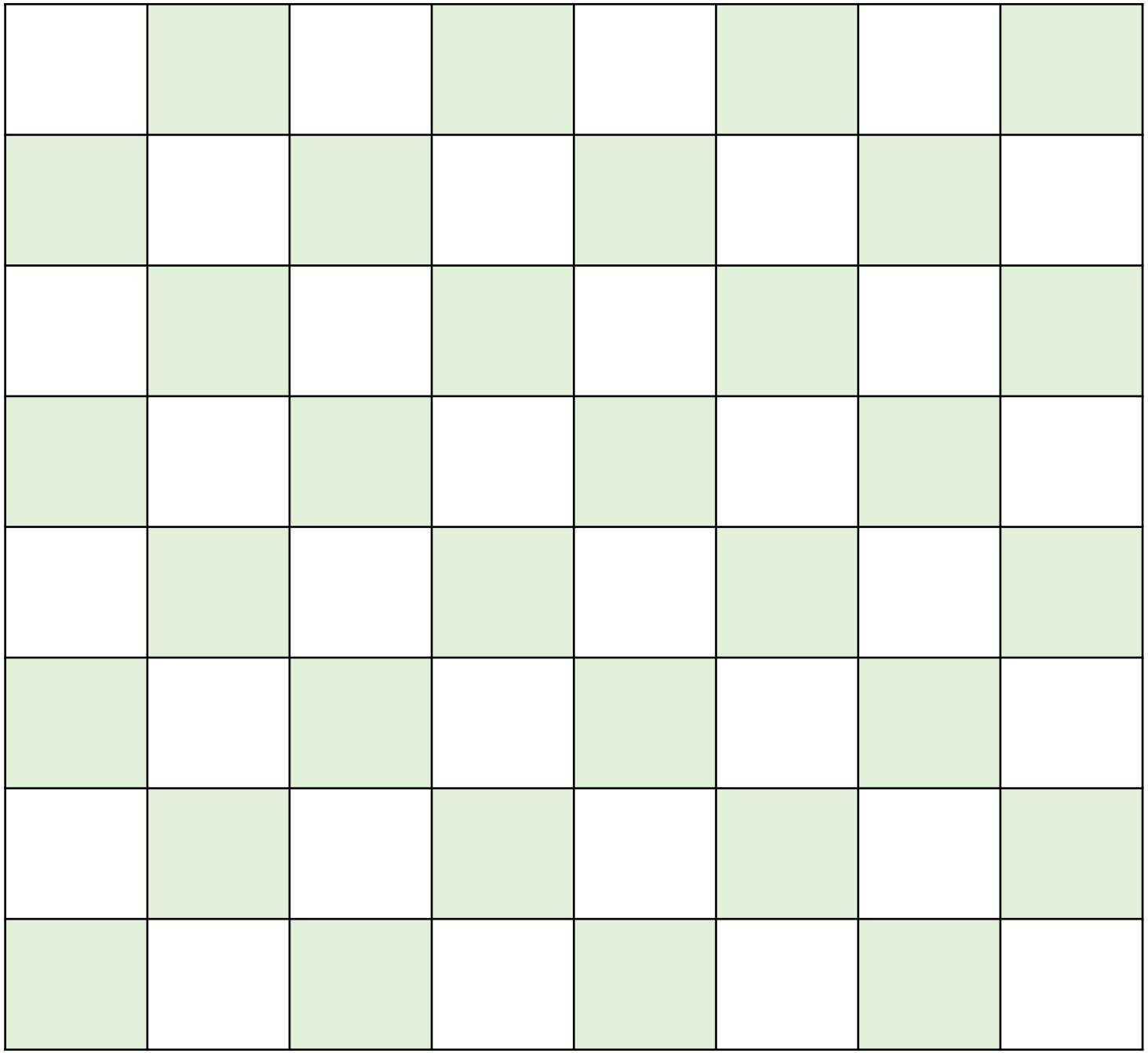


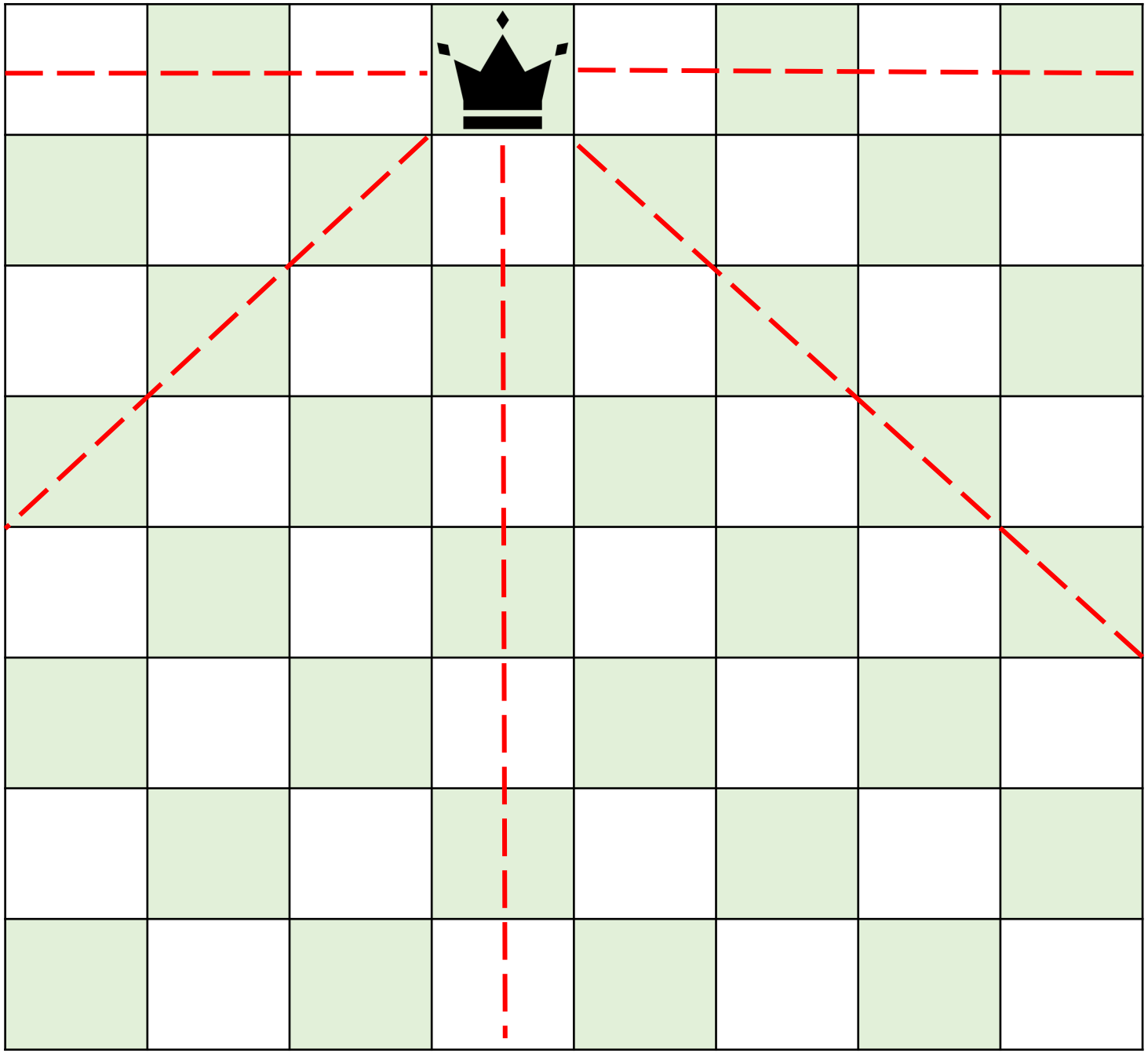


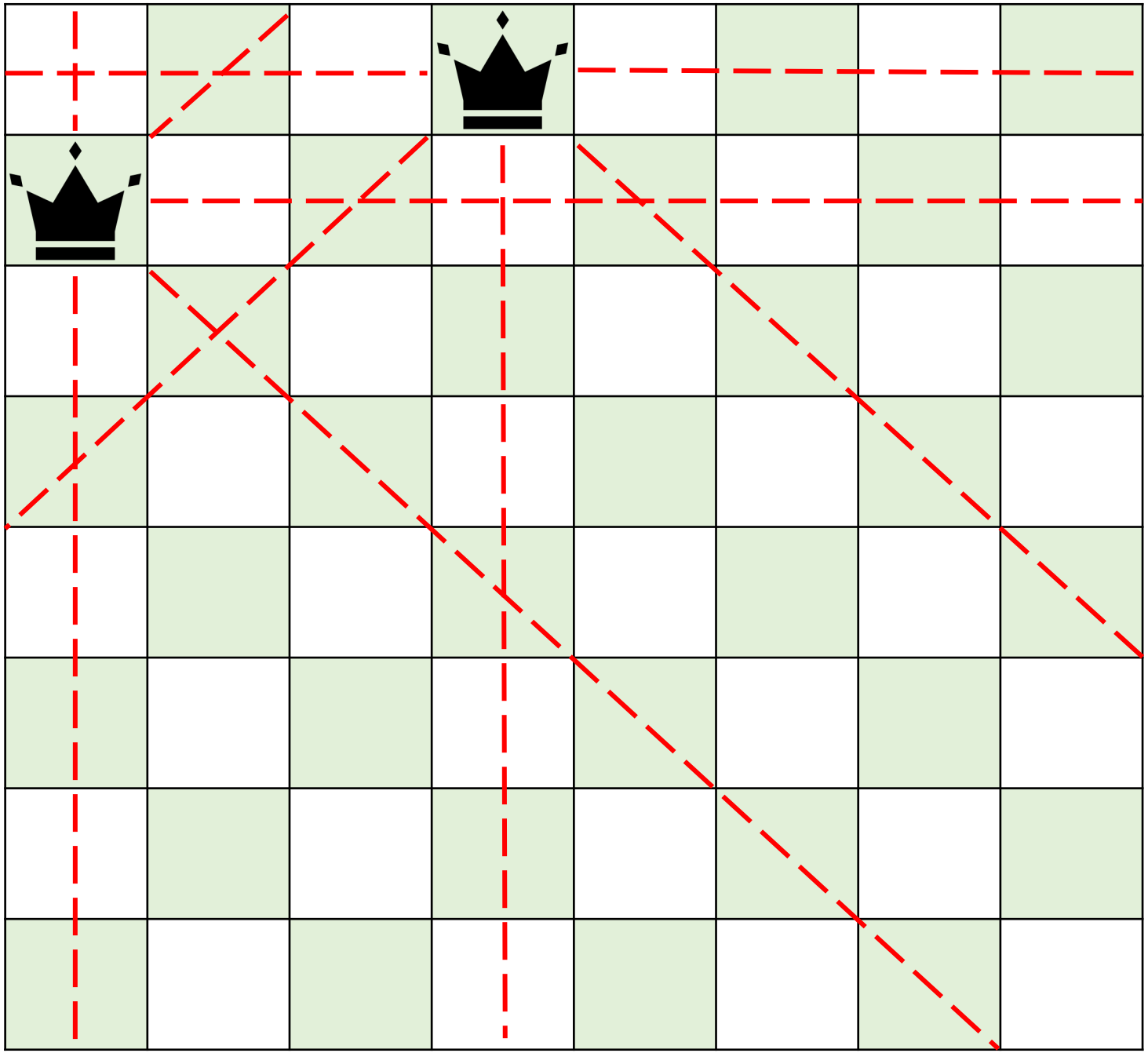


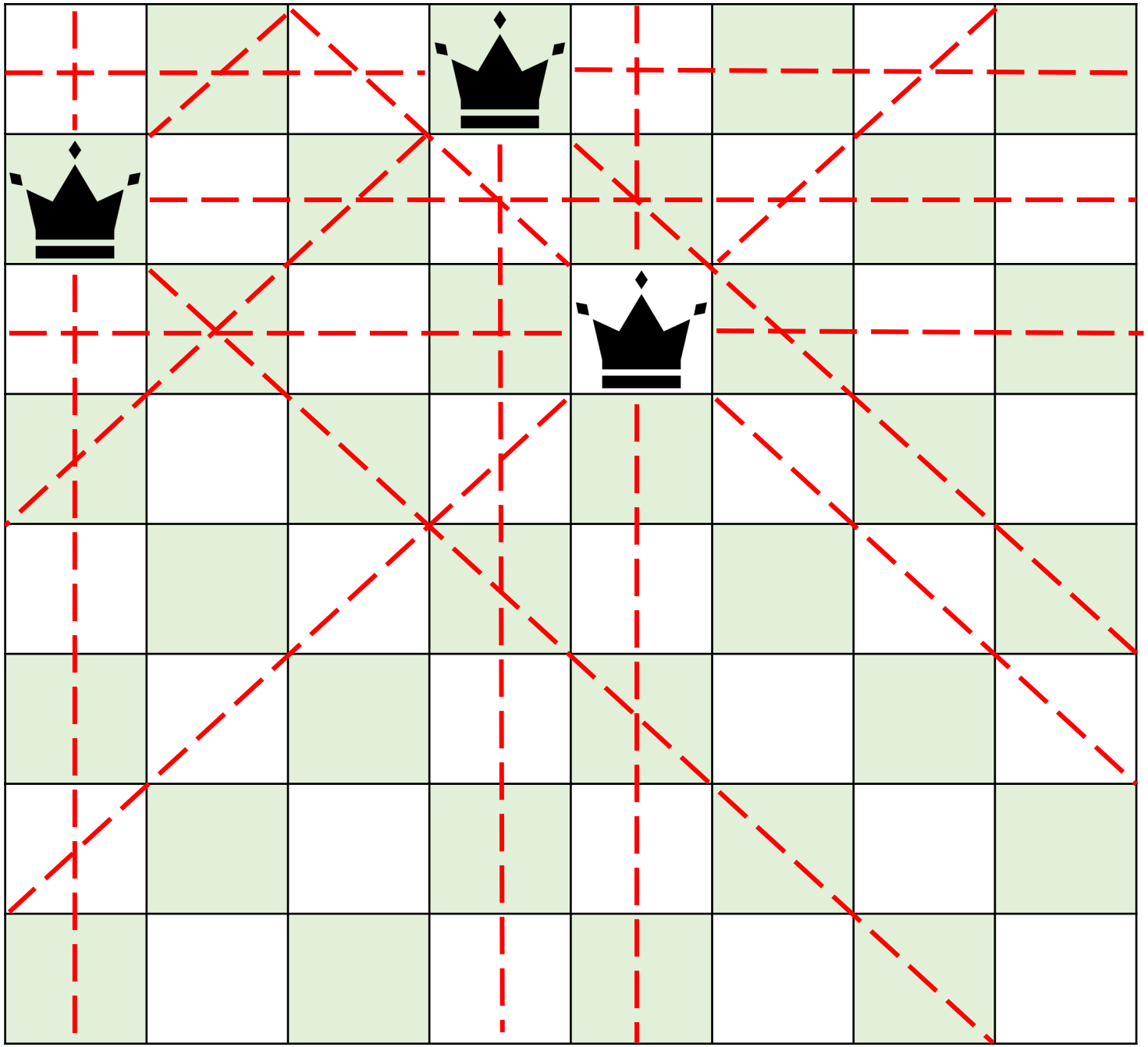
We are stuck. There is nowhere to place queen number 6, because every empty cell on the board is **in check** from queens 1 to 5.

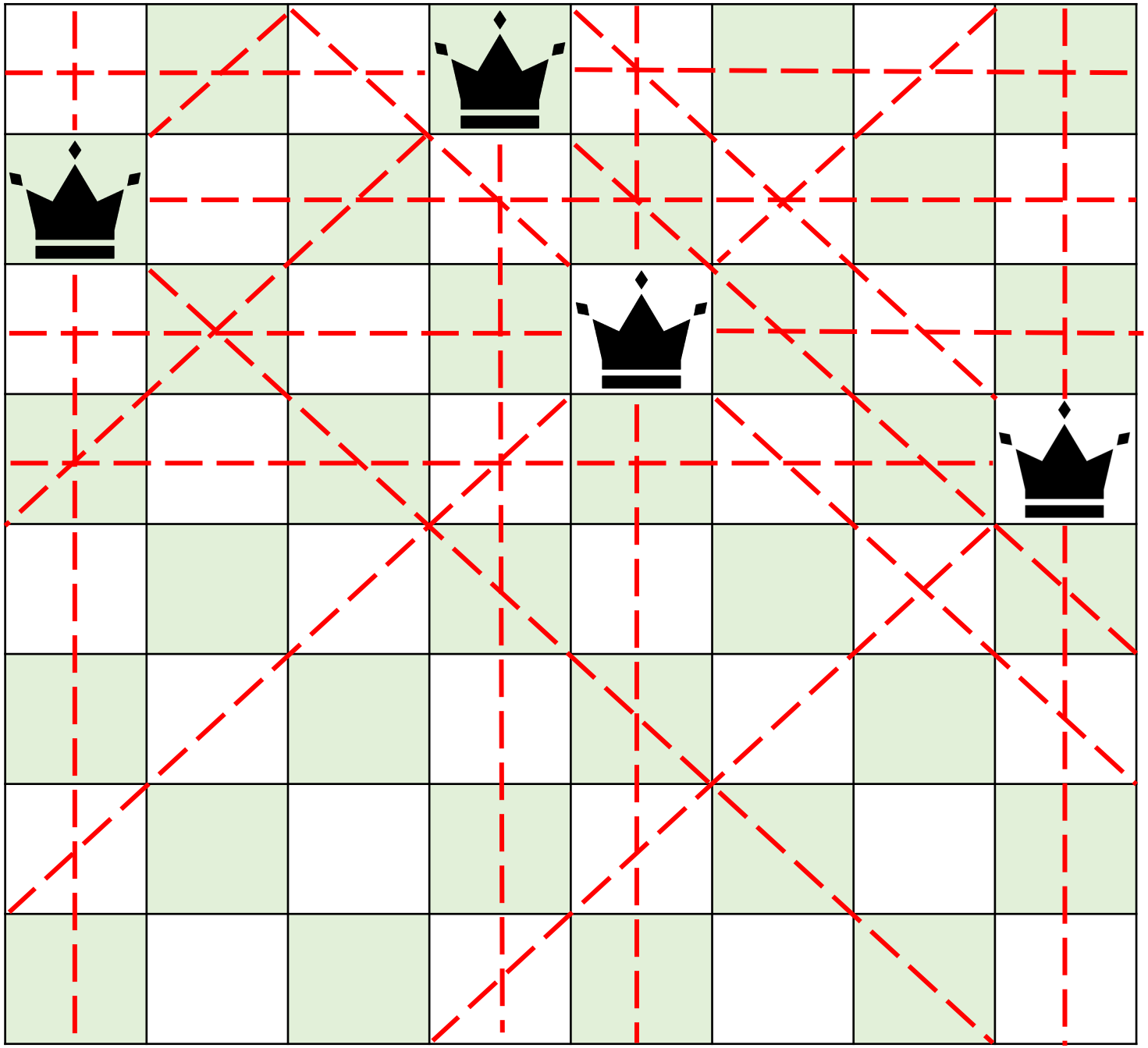
Let’s try again

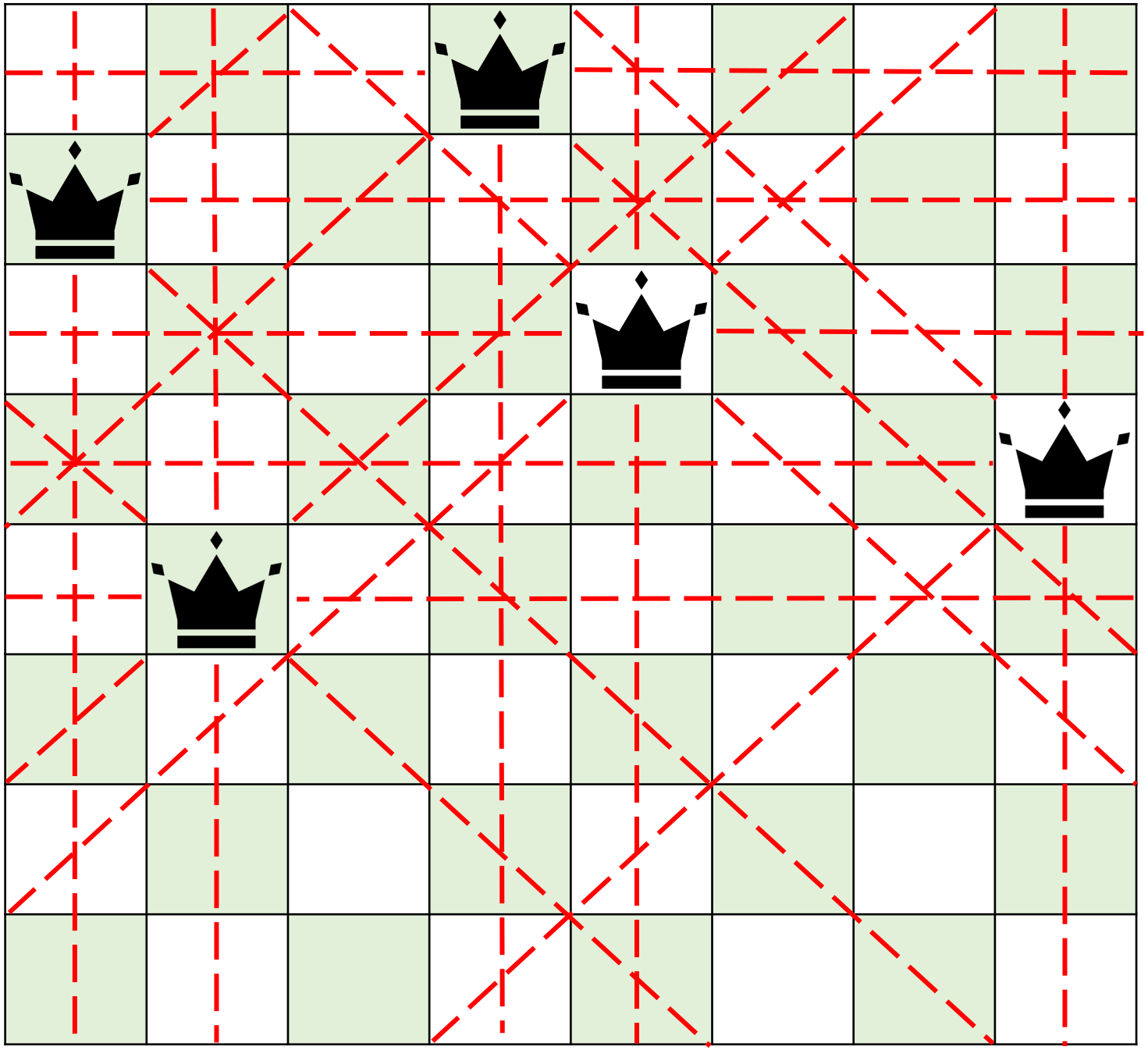


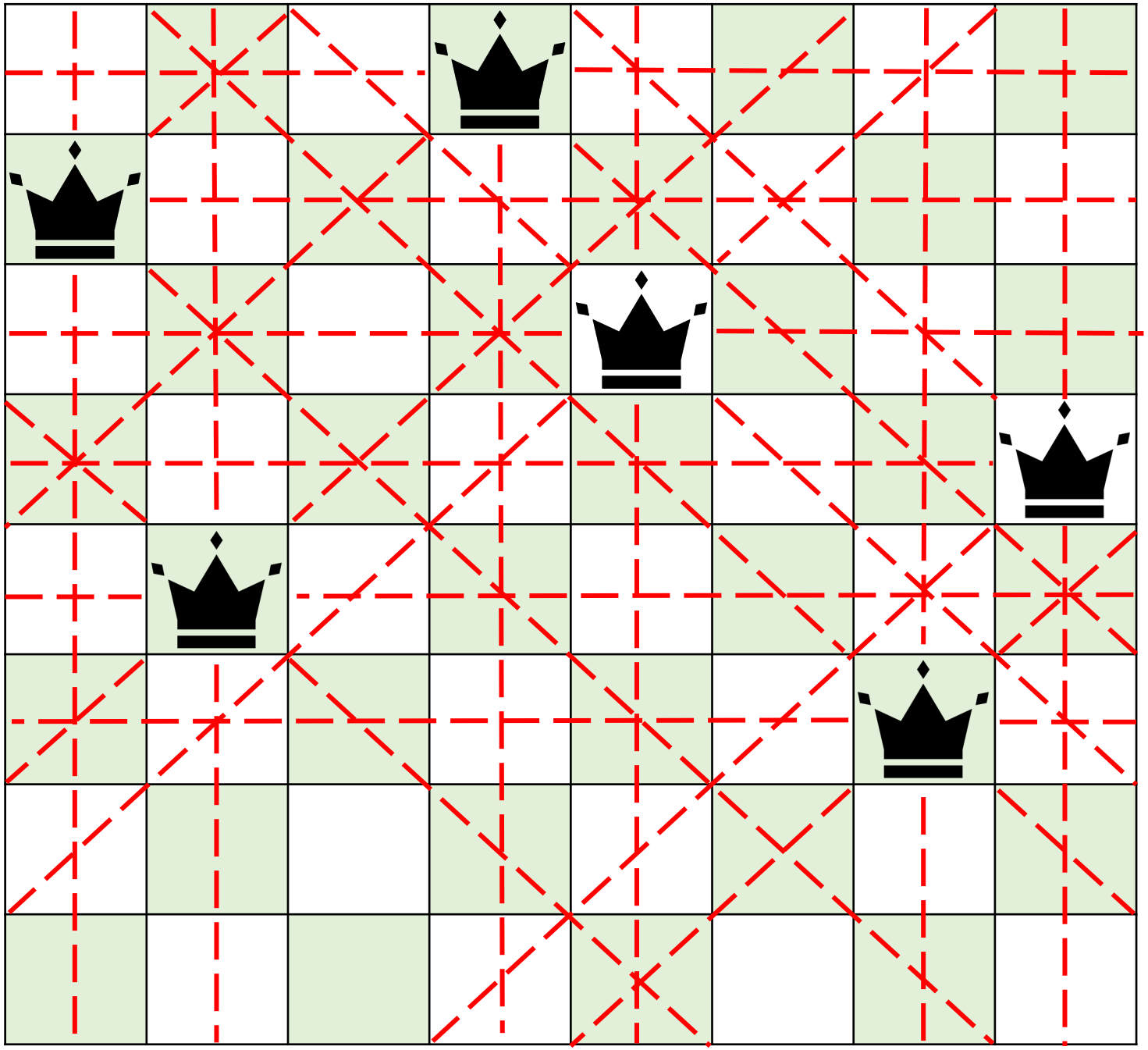


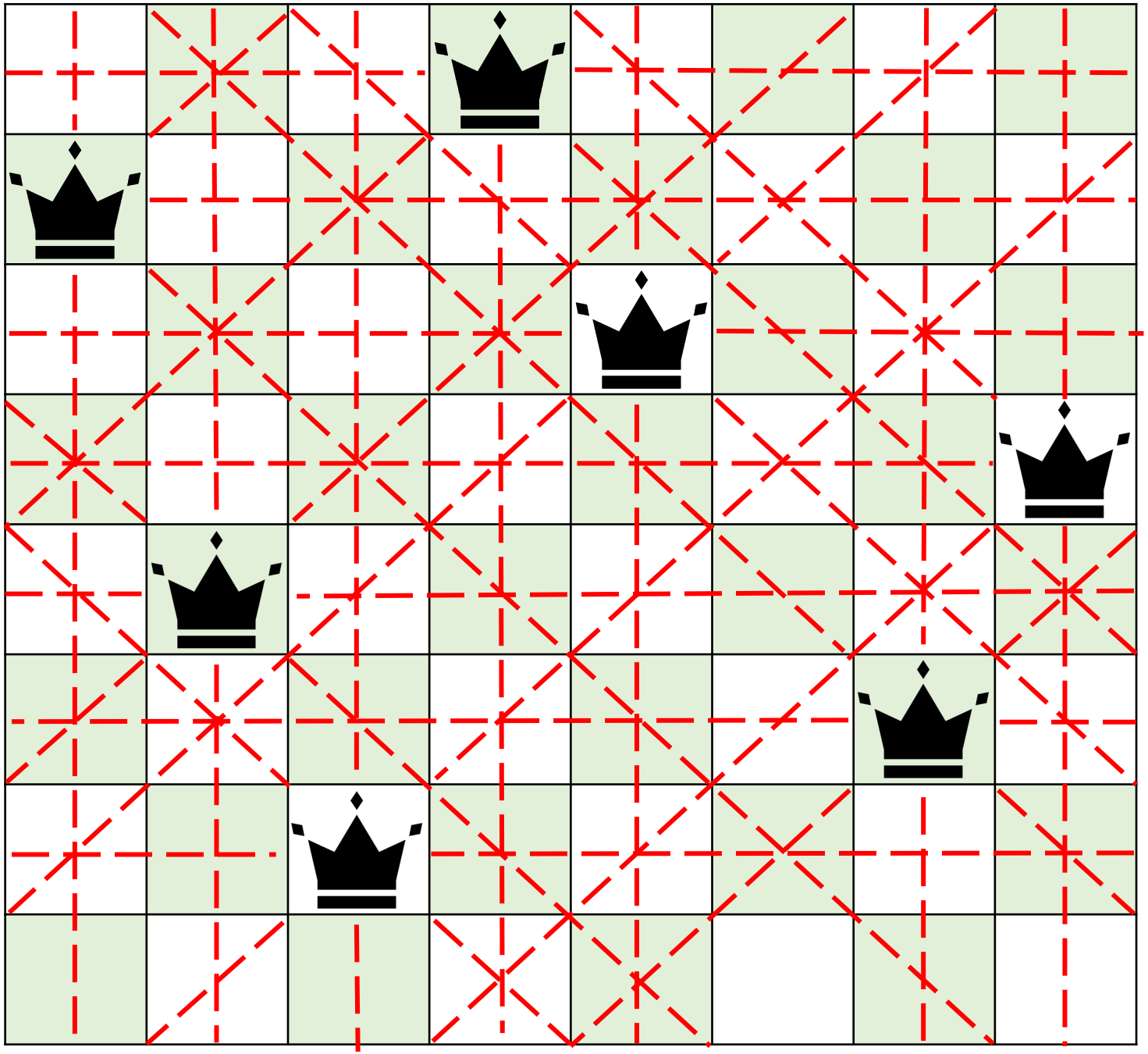


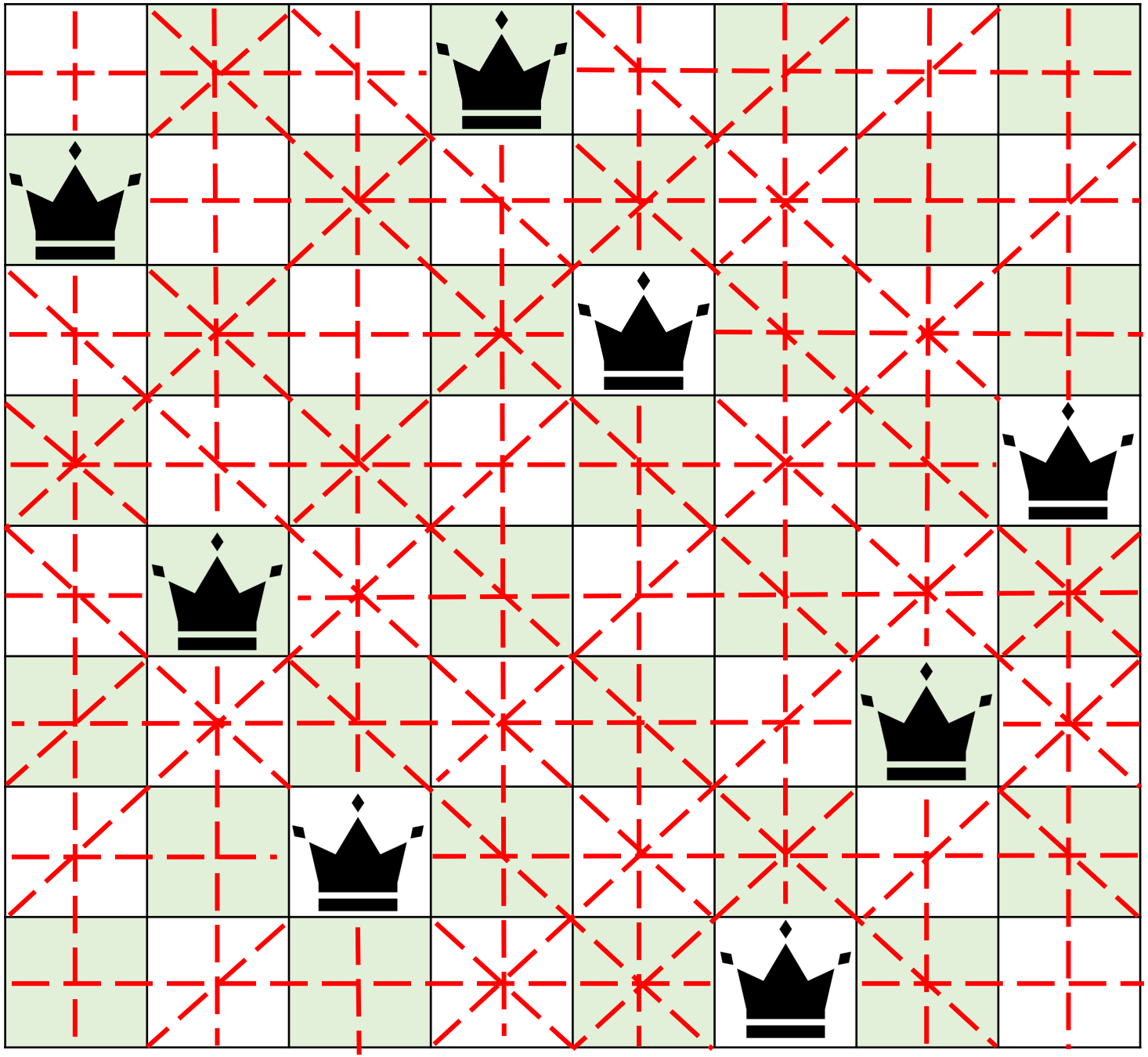


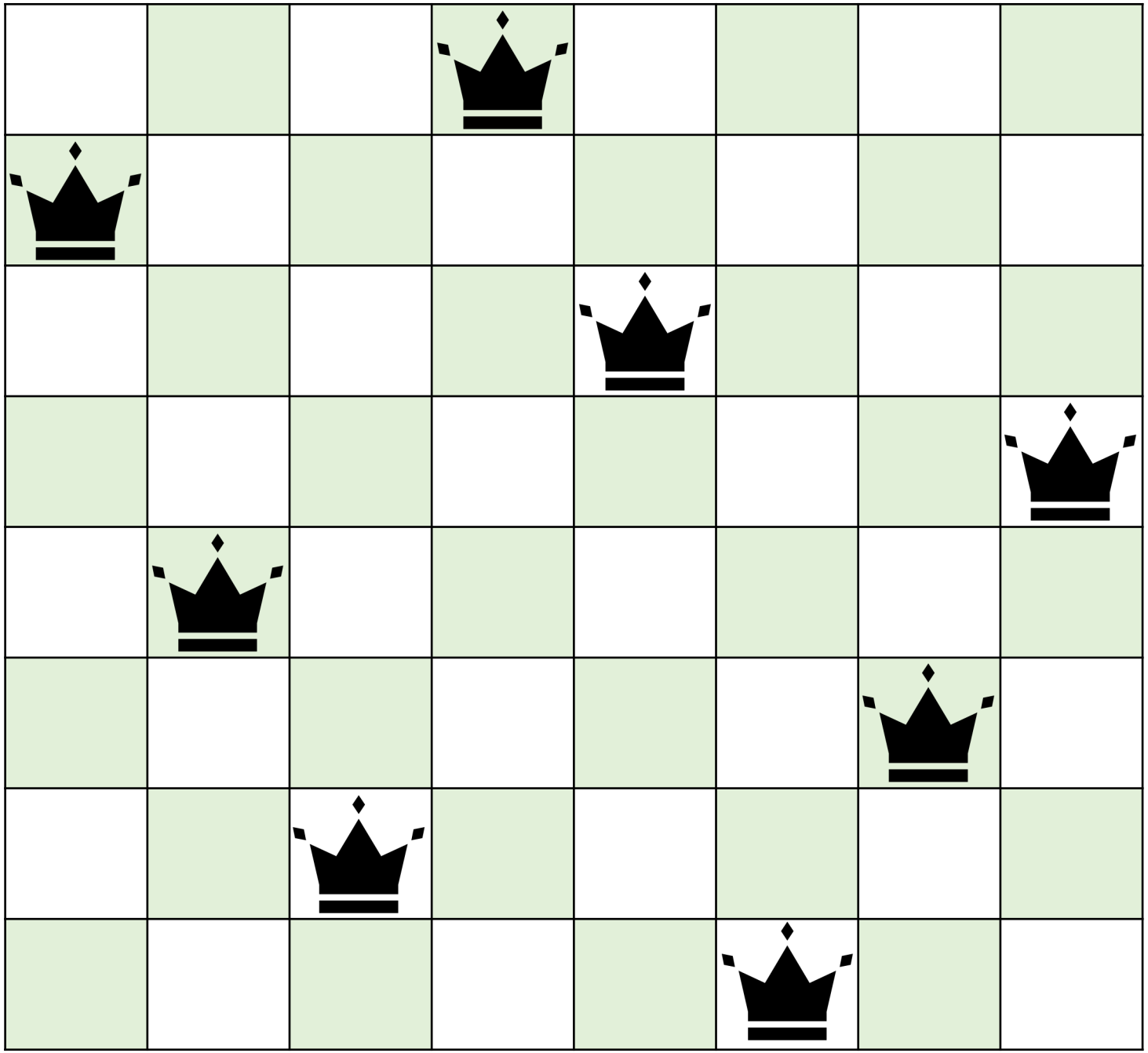












We found a solution

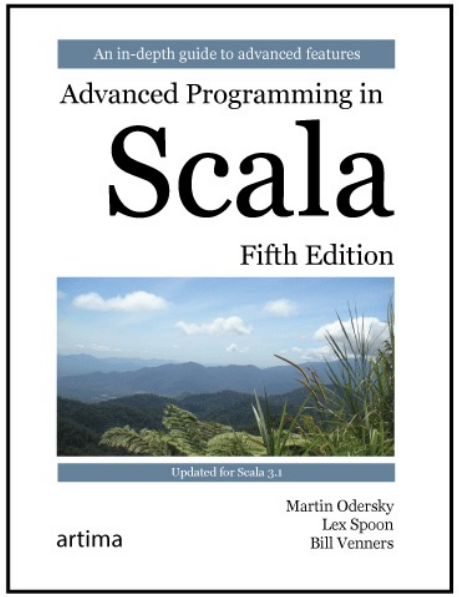
Martin Odersky

Lex Spoon

Bill Venners

A particularly suitable application area of **for expressions** are **combinatorial puzzles**.

An example of such a puzzle is the **8-queens problem**: **Given a standard chess-board, place eight queens such that no queen is in check from any other (a queen can check another piece if they are on the same column, row, or diagonal)**.



Martin Odersky

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

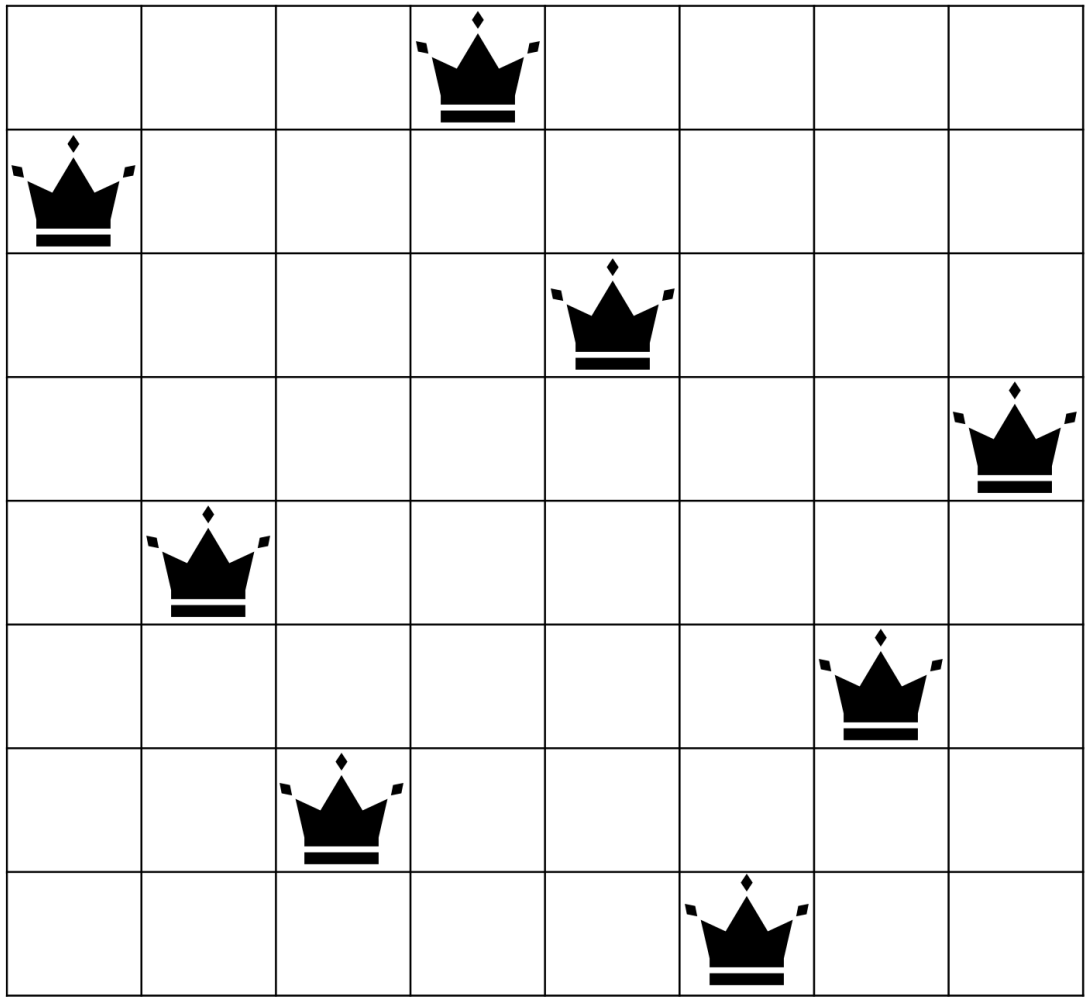
Once we have placed k - 1 queens, one must place the kth queen in a column where it's not "in check" with any other queen on the board.

Functional Programming Principles in Scala
1.18K subscribers

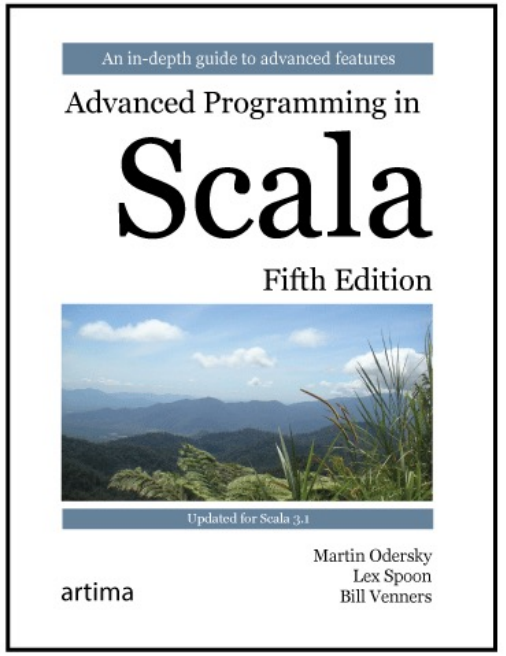
6.3 Combinatorial Search Example
830 views • 11 Sept 2017

YouTube

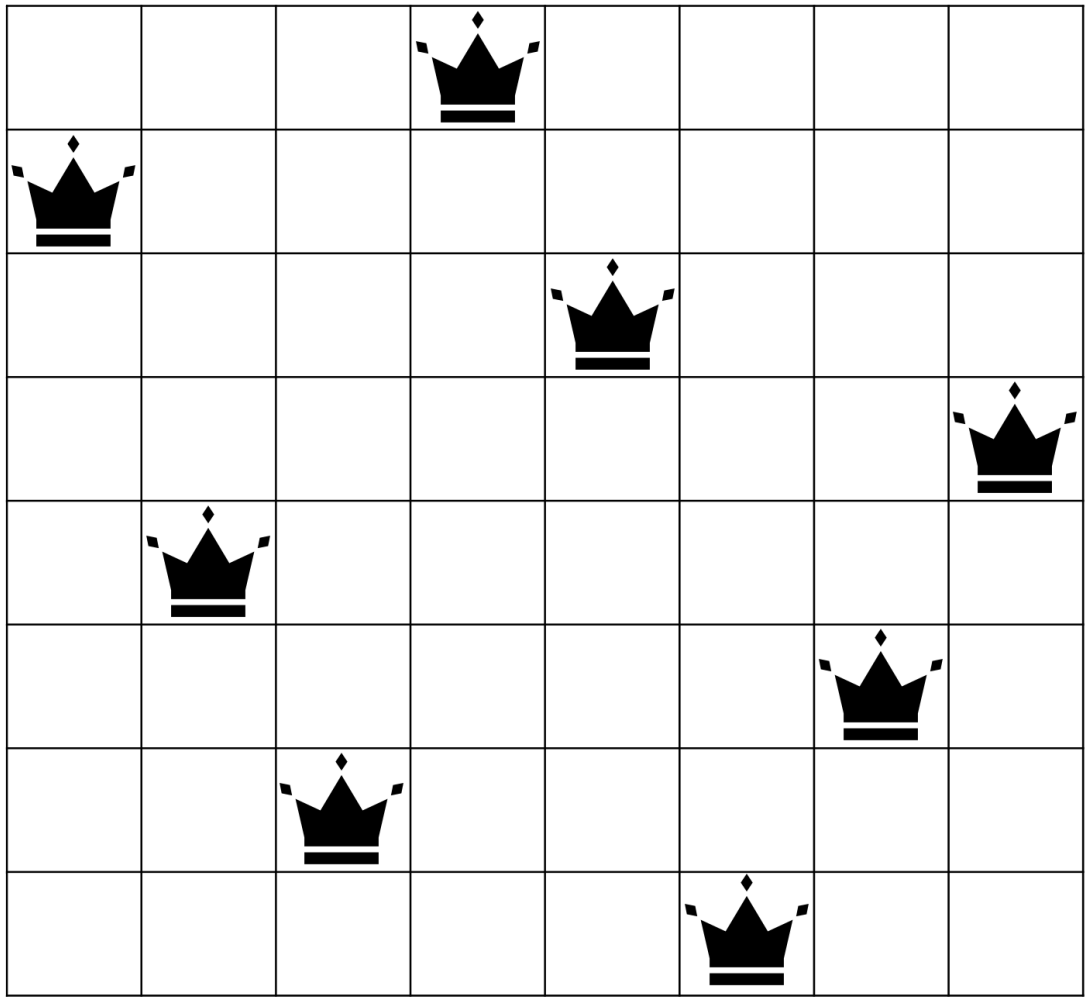
The eight queens puzzle is a special case of the more general ***n* queens problem** of placing *n* non-attacking queens on an *n*×*n* chessboard. Solutions exist for all natural numbers *n* with the exception of *n* = 2 and = 3.



ROW	COL
1	4
2	1
3	5
4	8
5	2
6	7
7	3
8	6




```
List((8, 6), (7, 3), (6, 7), (5, 2), (4, 8), (3, 5), (2, 1), (1, 4))
```



R C
O O
W L

(0, 3)

(1, 0)

(2, 4)

(3, 7)

(4, 1)

(5, 6)

(6, 2)

(7, 5)

```
List(5, 2, 6, 1, 7, 4, 0, 3)
```

Functional Programming Principles in Scala
1.18K subscribers

YouTube

6.3 Combinatorial Search Example

830 views • 11 Sept 2017

Functional Programming Principles in Scala

1.18K subscribers

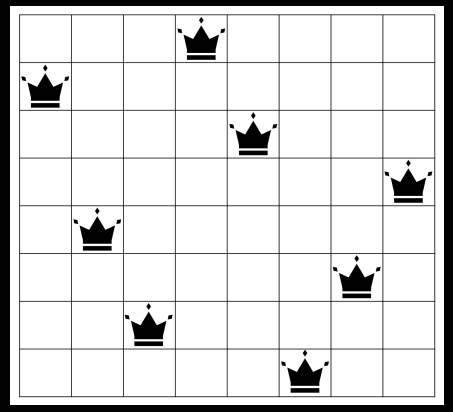
6.3 Combinatorial Search Example

830 views • 11 Sept 2017

```
def show(queens: List[Int]) = {
  val lines =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("* ").updated(col, "X ").mkString
  "\n" + (lines mkString "\n")
}                                                 > show: (queens: List[Int])java.lang.Stri

queens(4) map show                                > res0: scala.collection.immutable.Set[ja
                                                  | * * X *
                                                  | X * * *
                                                  | * * * X
                                                  | * X * * ", "
                                                  | * X * *
                                                  | * * * X
                                                  | X * * *
                                                  | * * X * ")
```

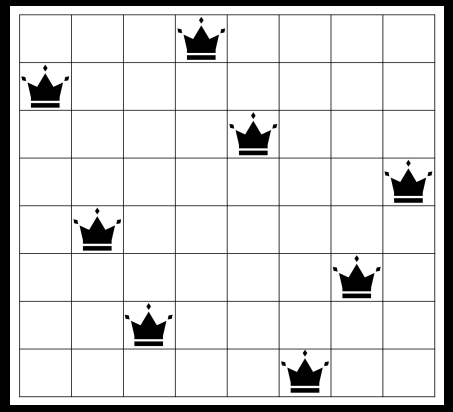
```scala
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```

```scala
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```



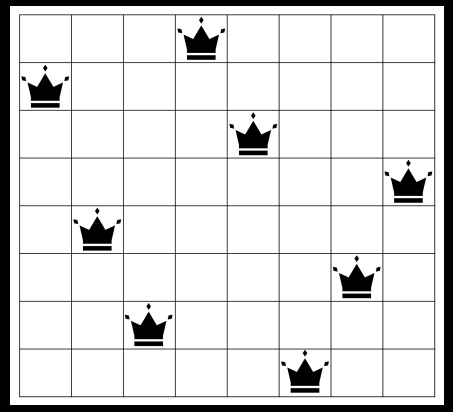
```scala
List(5, 2, 6, 1, 7, 4, 0, 3)
```

```scala
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```



```scala
println( show( List(5, 2, 6, 1, 7, 4, 0, 3) ) )
```

```
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```



Compositional Vector Graphics

```
val redSquare = Image.square(100).fillColor(Color.red)
```

```
val redSquare = Image.square(100).fillColor(Color.red)
```

```
val blueSquare = Image.square(100).fillColor(Color.blue)
```

```
val redSquare = Image.square(100).fillColor(Color.red)
```

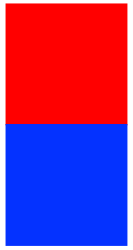
```
val blueSquare = Image.square(100).fillColor(Color.blue)
```

```
val redBesideBlue = redSquare.beside(blueSquare)
```

```
val redSquare = Image.square(100).fillColor(Color.red)
```

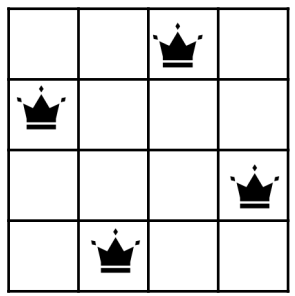
```
val blueSquare = Image.square(100).fillColor(Color.blue)
```

```
val redBesideBlue = redSquare.beside(blueSquare)
```

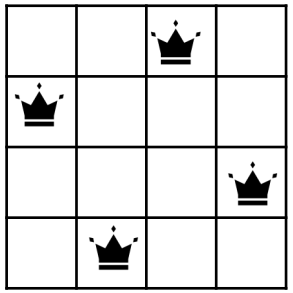


```
val redAboveBlue = redSquare.above(blueSquare)
```

Solution



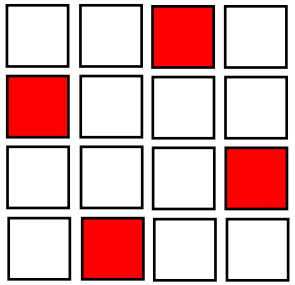
Solution



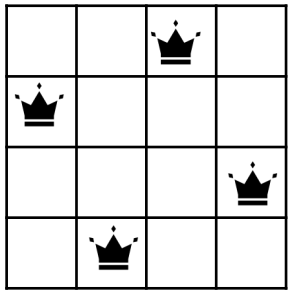
```
Image.square
Image.fillColor
```

create

red and white
square images



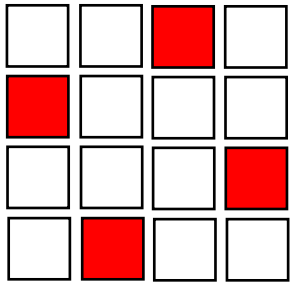
Solution



`Image.square`
`Image.fillColor`

create

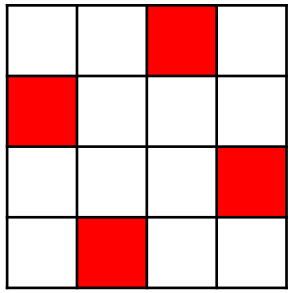
red and white
square images



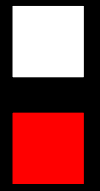
`Image.beside`
`Image.above`

compose

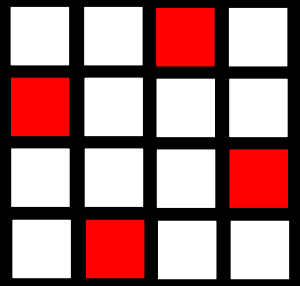
composite
solution image



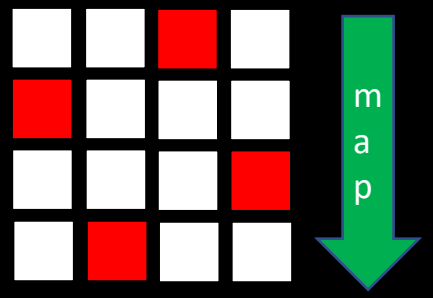
```
val square: Image = Image.square(100).strokeColor(Color.black)
val emptySquare: Image = square.fillColor(Color.white)
val fullSquare: Image = square.fillColor(Color.orangeRed)
```



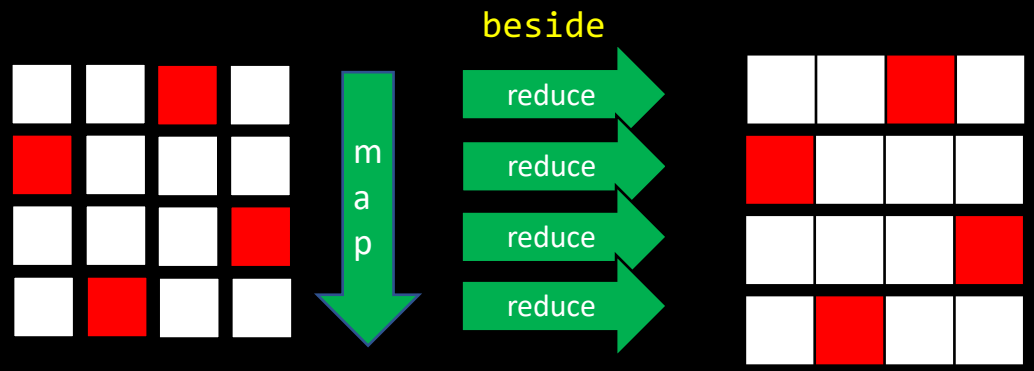
```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```



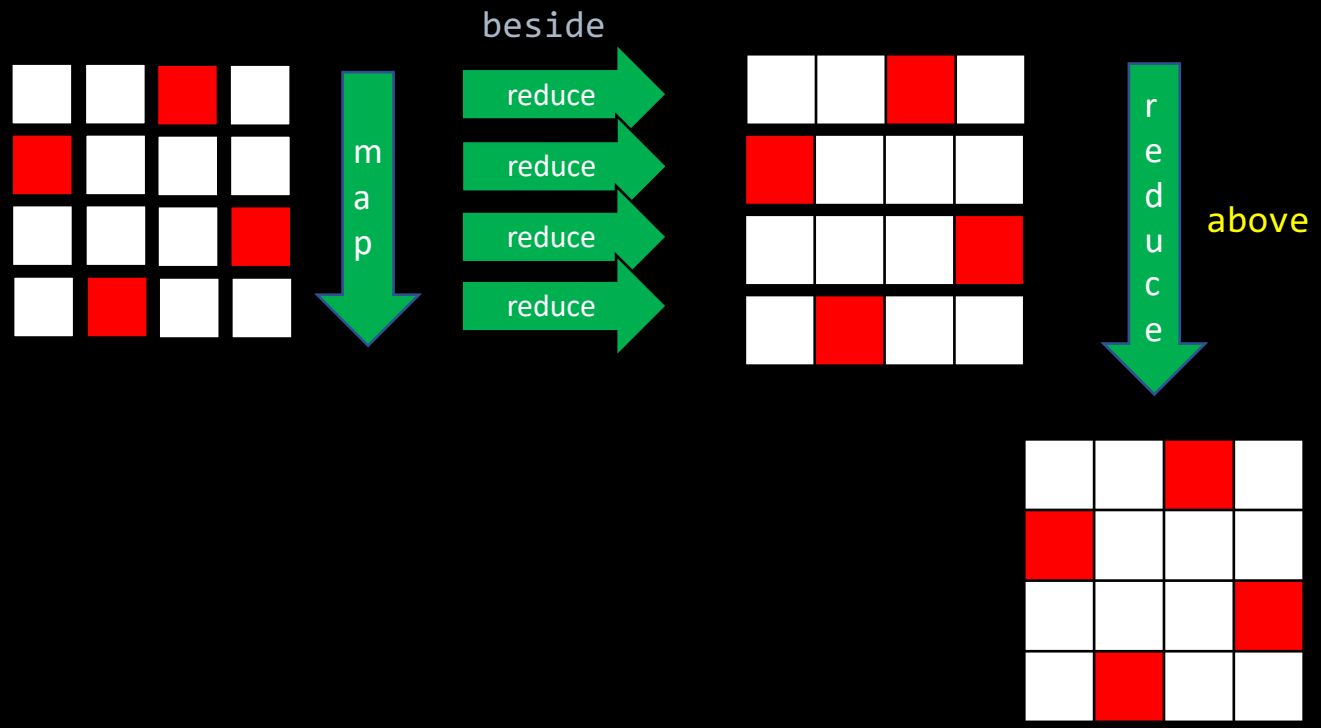
```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```



```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```

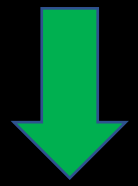


```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```



```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```

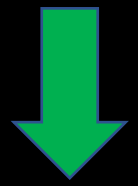
```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```



safer to use fold, in case any of the lists is empty

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.fold(Image.empty)(_ beside _))
    .fold(Image.empty)(_ above _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```



safer to use fold, in case any of the lists is empty

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.fold(Image.empty)(_ beside _))
    .fold(Image.empty)(_ above _)
```

fold is just an alias for foldLeft

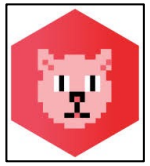
```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.foldLeft(Image.empty)(_ beside _))
    .foldLeft(Image.empty)(_ above _)
```


```scala
trait Foldable[F[_]] {

 def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

 def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def toList[A](as: F[A]): List[A] =
    foldRight(as)(List[A]())(_ :: _)
}
```

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```


```scala
trait Foldable[F[_]] {

 def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

 def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def toList[A](as: F[A]): List[A] =
    foldRight(as)(List[A]())(_ :: _)
}
```

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```

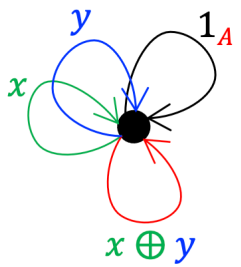


concatenate	fold,combineAll
foldMap	foldMap
foldLeft	foldLeft
foldRight	foldRight

Monoid

A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

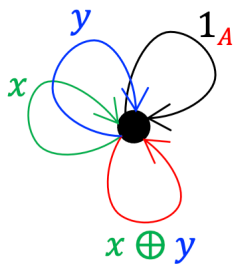
Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



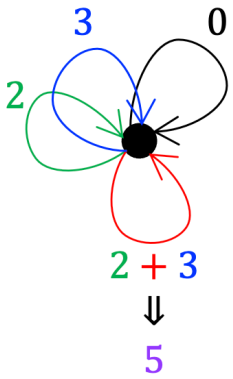
Monoid

A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



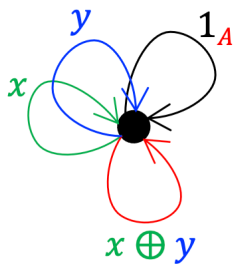
$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$



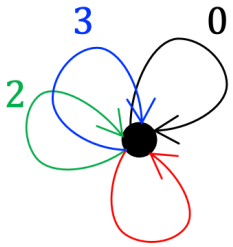
Monoid

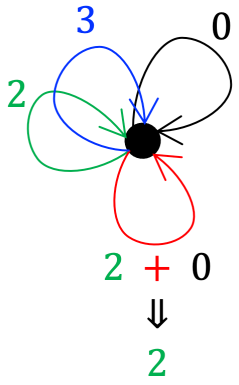
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$

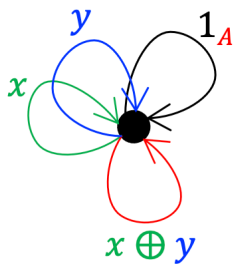




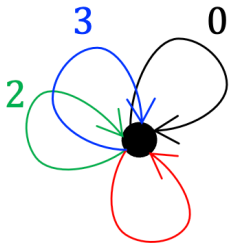
Monoid

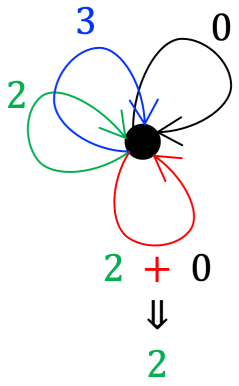
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

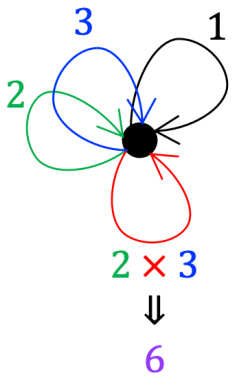


$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$





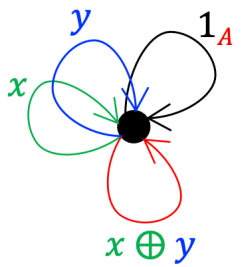
$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$



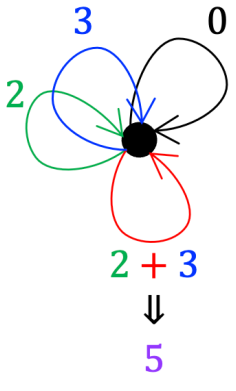
Monoid

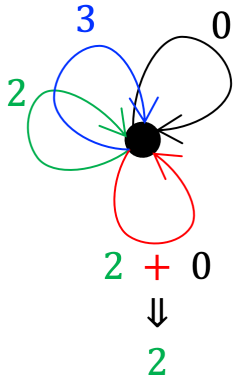
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

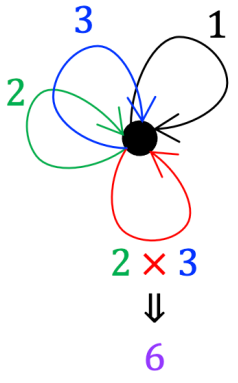


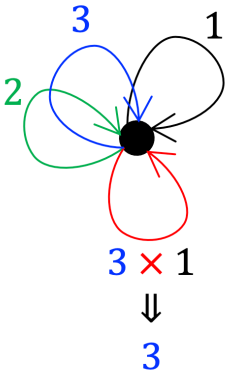
$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$





$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$

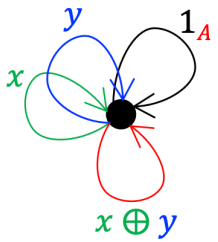




Monoid

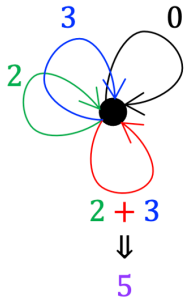
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

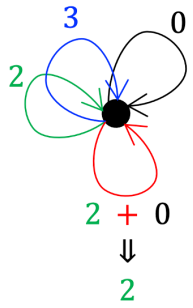
Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$




```
import cats.Monoid
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$



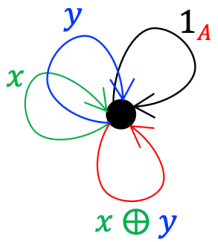



```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)
```

Monoid

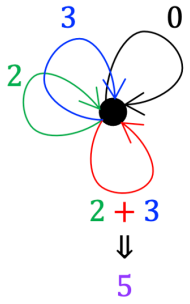
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

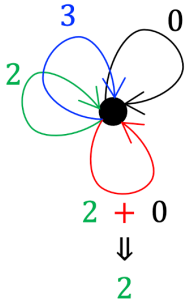
Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$




```
import cats.Monoid
import cats.syntax.monoid.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$



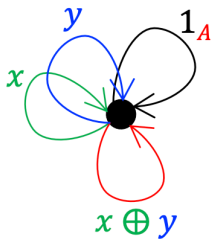



```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)
```

Monoid

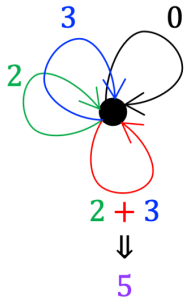
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

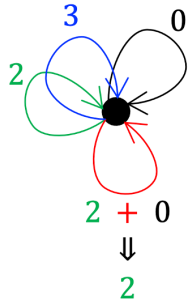
Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$




```
import cats.Monoid
import cats.syntax.monoid.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$



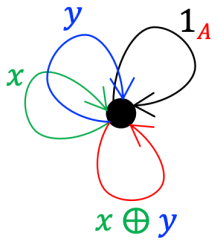



```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)
```

Monoid

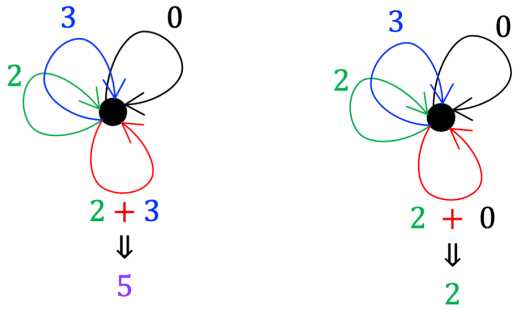
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

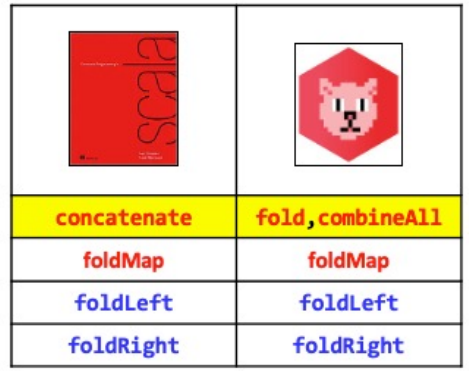


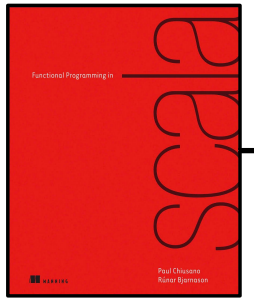

```
import cats.Monoid
import cats.syntax.monoid.*
import cats.syntax.foldable.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$




```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)

assert(List.empty[Int].combineAll == 0)
assert(List(1, 2, 3, 4).combineAll == 10)
```



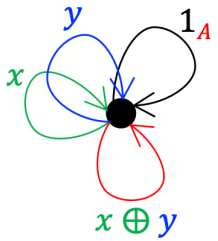


```
trait Foldable[F[_]] {
  …
  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
  …
}
```

Monoid

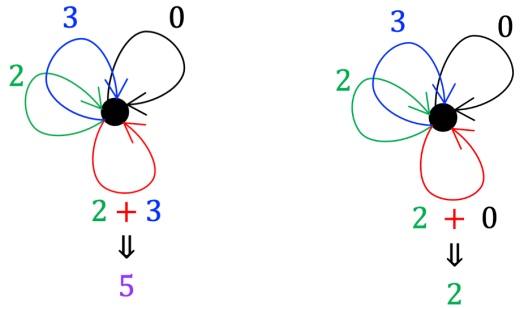
A: type (set of values)
$\oplus: A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



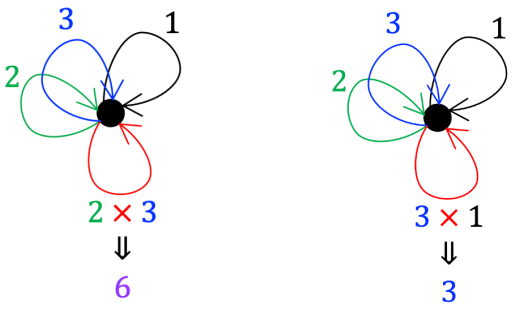

```
import cats.Monoid
import cats.syntax.monoid.*
import cats.syntax.foldable.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$




```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)

assert(List.empty[Int].combineAll == 0)
assert(List(1, 2, 3, 4).combineAll == 10)
```

$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$

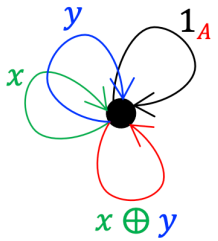



```
val prodMonoid = cats.Monoid.instance[Int](emptyValue = 1, cmb = _ * _)
```

Monoid

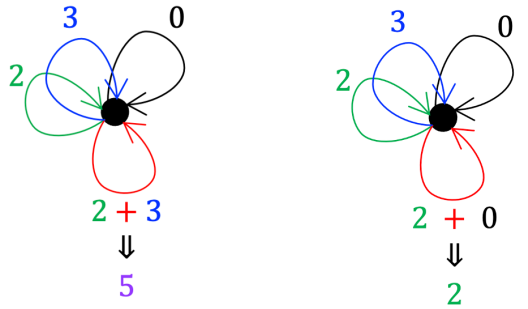
A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



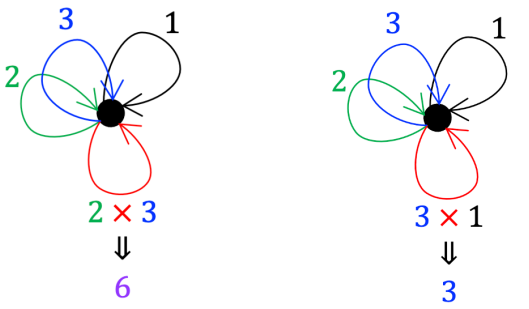

```
import cats.Monoid
import cats.syntax.monoid.*
import cats.syntax.foldable.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$




```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)

assert(List.empty[Int].combineAll == 0)
assert(List(1, 2, 3, 4).combineAll == 10)
```

$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$

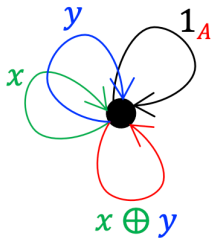



```
val prodMonoid = cats.Monoid.instance[Int](emptyValue = 1, cmb = _ * _)

assert(prodMonoid.combine(2, 3) == 6)
assert(prodMonoid.combine(3, prodMonoid.empty) == 3)
```

Monoid

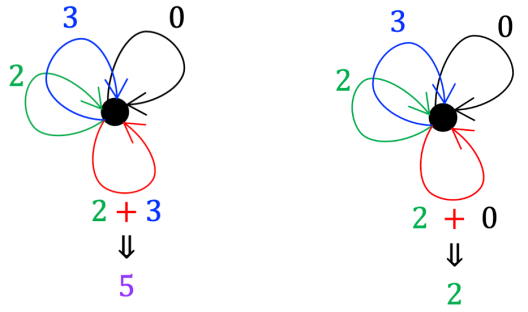
A: type (set of values)
$\oplus: A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



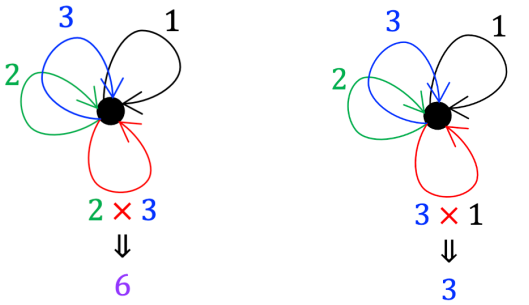

```
import cats.Monoid
import cats.syntax.monoid.*
import cats.syntax.foldable.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$




```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)

assert(List.empty[Int].combineAll == 0)
assert(List(1, 2, 3, 4).combineAll == 10)
```

$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$

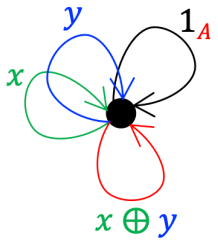



```
val prodMonoid = cats.Monoid.instance[Int](emptyValue = 1, cmb = _ * _)

assert(prodMonoid.combine(2, 3) == 6)
assert(prodMonoid.combine(3, prodMonoid.empty) == 3)

assert(prodMonoid.empty == 1)
```

Monoid

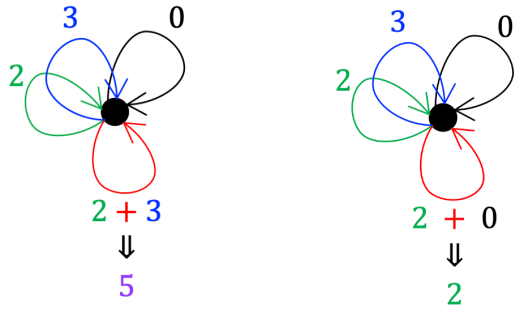
A: type (set of values)
$\oplus: A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



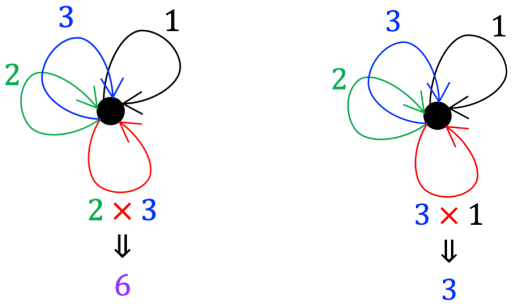

```
import cats.Monoid
import cats.syntax.monoid.*
import cats.syntax.foldable.*
```

$\{A = \mathbb{N}, \oplus = +, 1_A = 0\}$




```
assert(Monoid[Int].combine(2,3) == 5)
assert(Monoid[Int].combine(2, Monoid[Int].empty) == 2)

assert((2 |+| 3) == 5)
assert((2 |+| Monoid[Int].empty) == 2)

assert(Monoid[Int].empty == 0)

assert(List.empty[Int].combineAll == 0)
assert(List(1, 2, 3, 4).combineAll == 10)
```

$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$



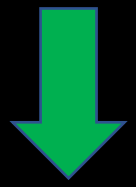

```
val prodMonoid = cats.Monoid.instance[Int](emptyValue = 1, cmb = _ * _)

assert(prodMonoid.combine(2, 3) == 6)
assert(prodMonoid.combine(3, prodMonoid.empty) == 3)

assert(prodMonoid.empty == 1)

assert(List.empty[Int].combineAll(prodMonoid) == 1)
assert(List(1, 2, 3, 4).combineAll(prodMonoid) == 24)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.foldLeft(Image.empty)(_ beside _))
    .foldLeft(Image.empty)(_ above _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.foldLeft(Image.empty)(_ beside _))
    .foldLeft(Image.empty)(_ above _)
```

```scala
import cats.Monoid
import cats.implicits.*
val beside = Monoid.instance[Image](Image.empty, _ beside _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.foldLeft(Image.empty)(_ beside _))
    .foldLeft(Image.empty)(_ above _)
```

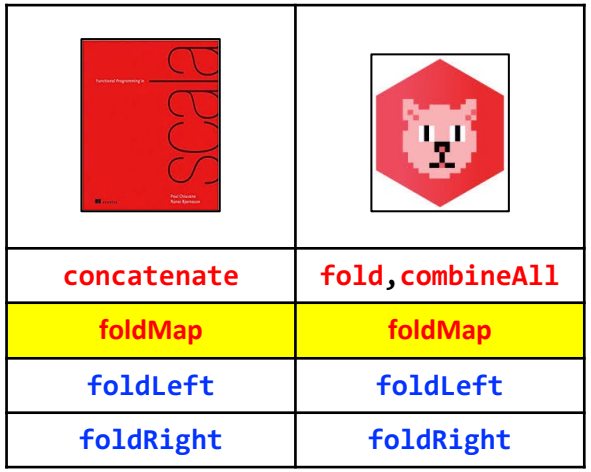


```scala
import cats.Monoid
import cats.implicits.*
val beside = Monoid.instance[Image](Image.empty, _ beside _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .foldLeft(Image.empty)(_ above _)
```


```scala
trait Foldable[F[_]] {

 def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

 def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def toList[A](as: F[A]): List[A] =
    foldRight(as)(List[A]())(_ :: _)
}
```

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



The marketing buzzword for **foldMap** is **MapReduce**.

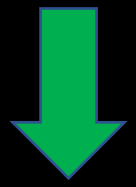

```scala
object ListFoldable extends Foldable[List] {

 override def foldMap[A, B](as:List[A])(f: A => B)(mb: Monoid[B]): B =
    foldLeft(as)(mb.zero)((b, a) => mb.op(b, f(a)))

  override def foldRight[A, B](as: List[A])(z: B)(f: (A,B) => B) =
    as match {
      case Nil => z
      case Cons(h, t) => f(h, foldRight(t, z)(f))
    }

  override def foldLeft[A, B](as: List[A])(z: B)(f: (B,A) => B) =
    as match {
      case Nil => z
      case Cons(h, t) => foldLeft(t, f(z,h))(f)
    }
}
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .foldLeft(Image.empty)(_ above _)
```

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .foldLeft(Image.empty)(_ above _)
```

purely to make the next step easier to understand,
let's revert to using reduce rather than foldLeft.

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .foldLeft(Image.empty)(_ above _)
```



purely to make the next step easier to understand,
let's revert to using reduce rather than foldLeft.

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .reduce(_ above _)
```

```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .reduce(_ above _)
```

MapReduce is the marketing buzzword for foldMap

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .reduce(_ above _)
```

MapReduce is the marketing buzzword for foldMap

```scala
val above = Monoid.instance[Image](Image.empty, _ above _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.combineAll(beside))
    .reduce(_ above _)
```

MapReduce is the marketing buzzword for foldMap

```scala
val above = Monoid.instance[Image](Image.empty, _ above _)
```

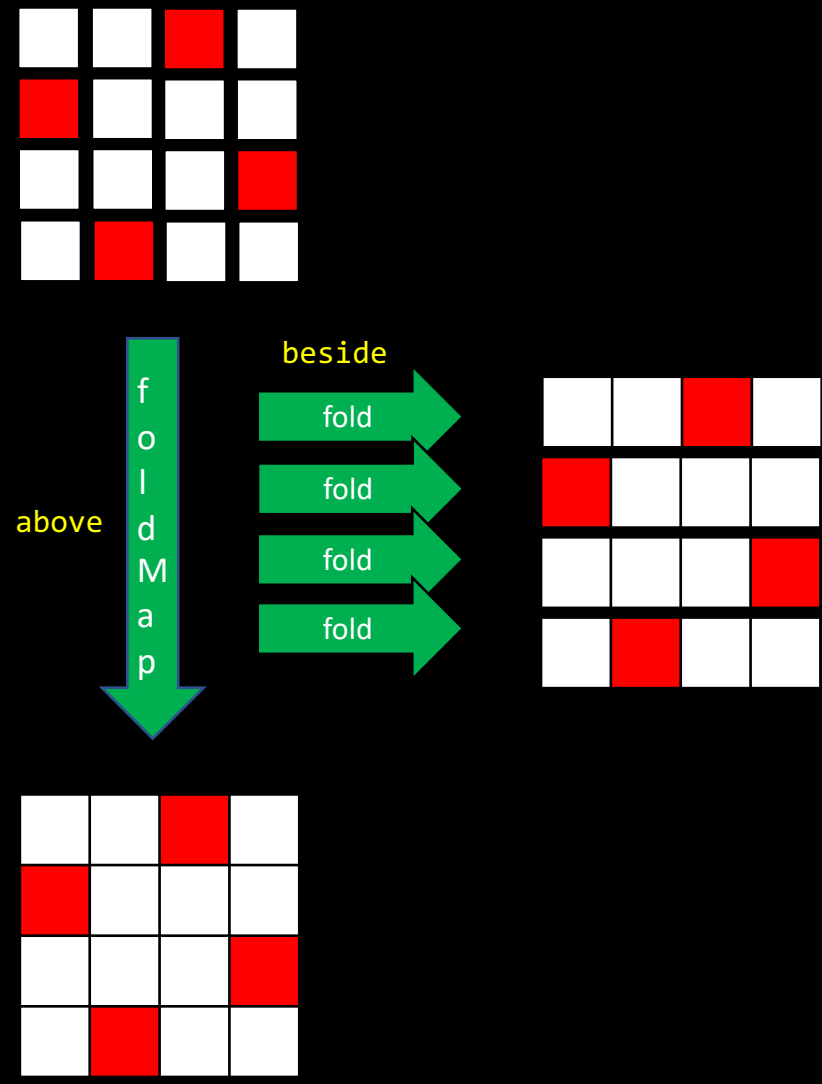
```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_.combineAll(beside))(above)
```

RECAP

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_.combineAll(beside))(above)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_.fold(beside))(above)
```



```scala
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```

❎ ❎ ❎ 👑 ❎ ❎ ❎ ❎
👑 ❎ ❎ ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ 👑 ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ ❎ ❎ 👑
❎ 👑 ❎ ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ ❎ 👑 ❎
❎ ❎ 👑 ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ 👑 ❎ ❎

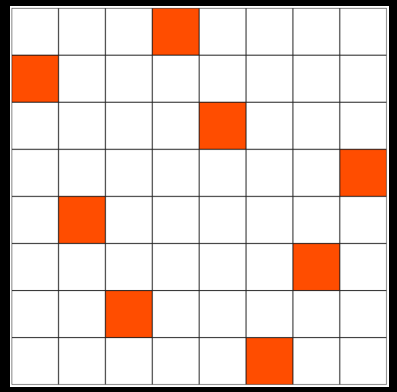
```scala
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```

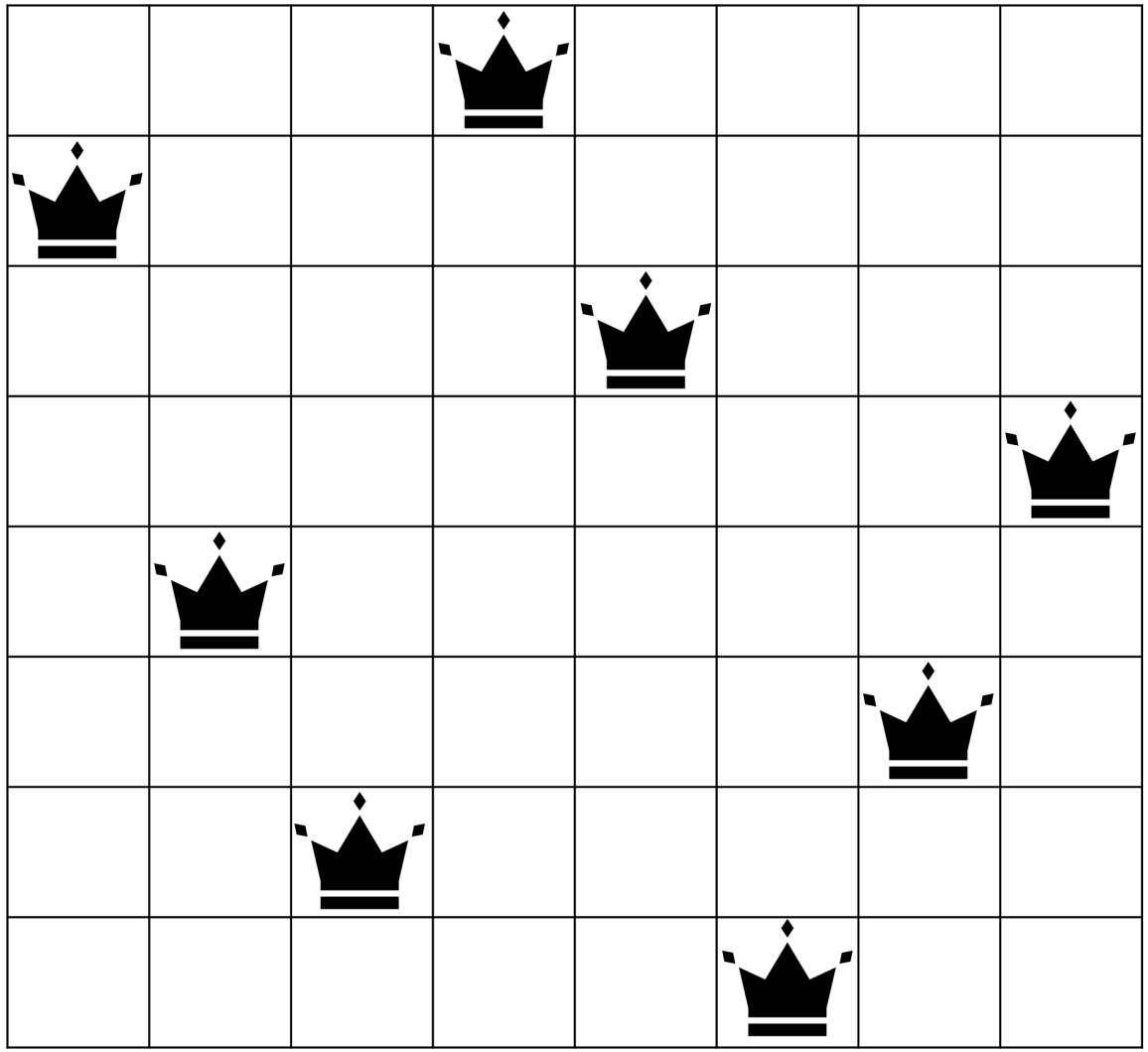
```scala
def show(queens: List[Int]): Image =
  val squareImageGrid: List[List[Image]] =
    for col <- queens.reverse
    yield List.fill(queens.length)(emptySquare)
                .updated(col,fullSquare)
  combine(squareImageGrid)
```

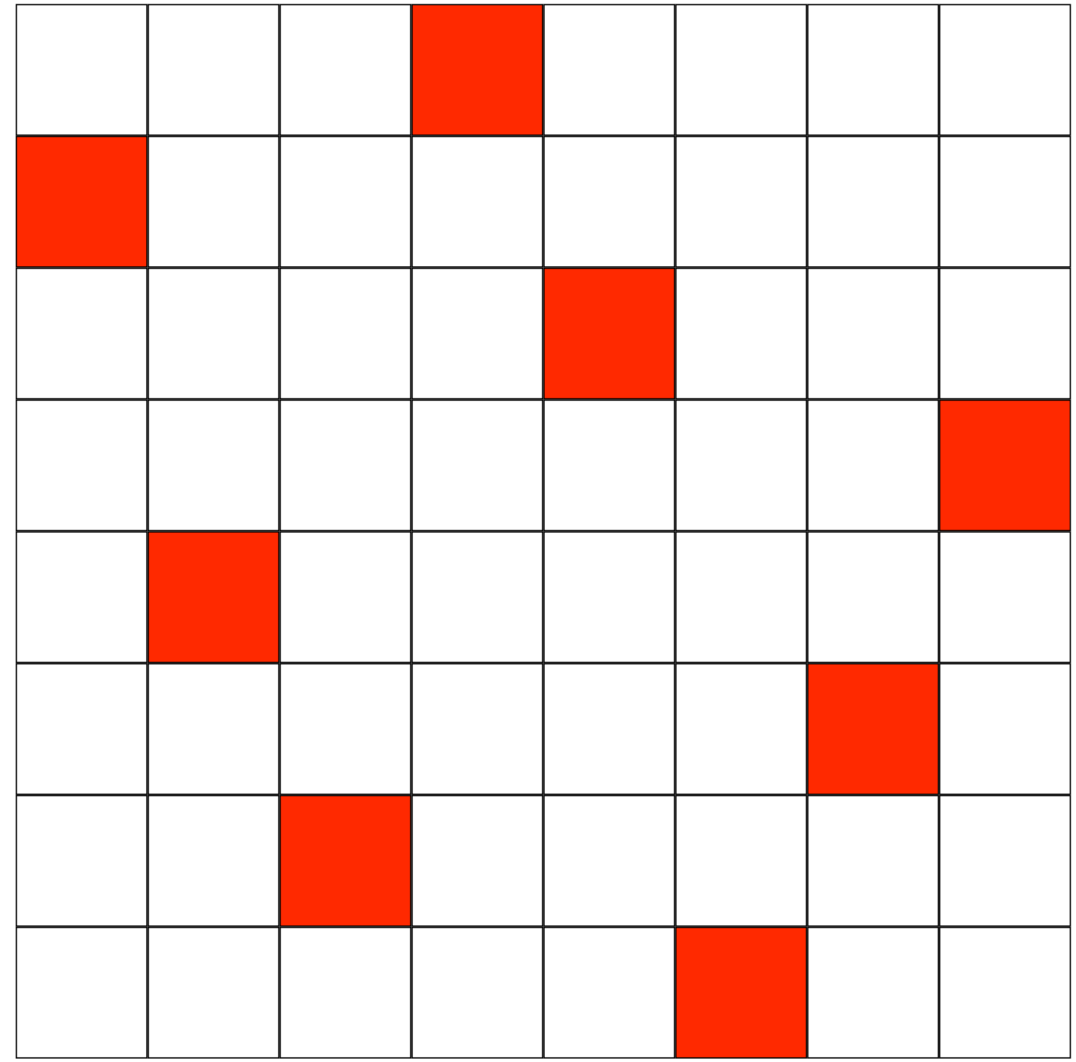
```
def show(queens: List[Int]): String =
  val lines: List[String] =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("❎ ")
                .updated(col, s"👑 ")
                .mkString
  "\n" + lines.mkString("\n")
```

❎ ❎ ❎ 👑 ❎ ❎ ❎ ❎
👑 ❎ ❎ ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ 👑 ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ ❎ ❎ 👑
❎ 👑 ❎ ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ ❎ 👑 ❎
❎ ❎ 👑 ❎ ❎ ❎ ❎ ❎
❎ ❎ ❎ ❎ ❎ 👑 ❎ ❎

```
def show(queens: List[Int]): Image =
  val squareImageGrid: List[List[Image]] =
    for col <- queens.reverse
    yield List.fill(queens.length)(emptySquare)
              .updated(col,fullSquare)
  combine(squareImageGrid)
```

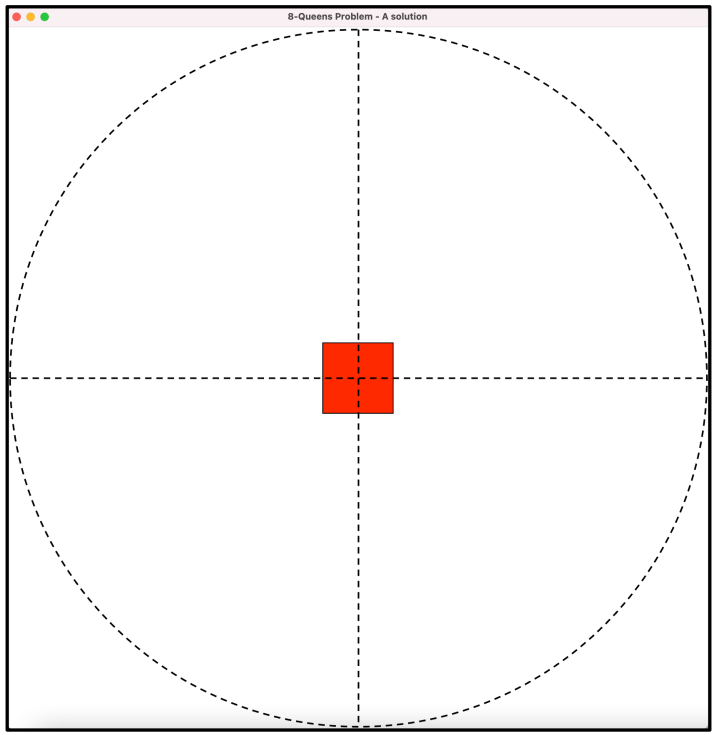




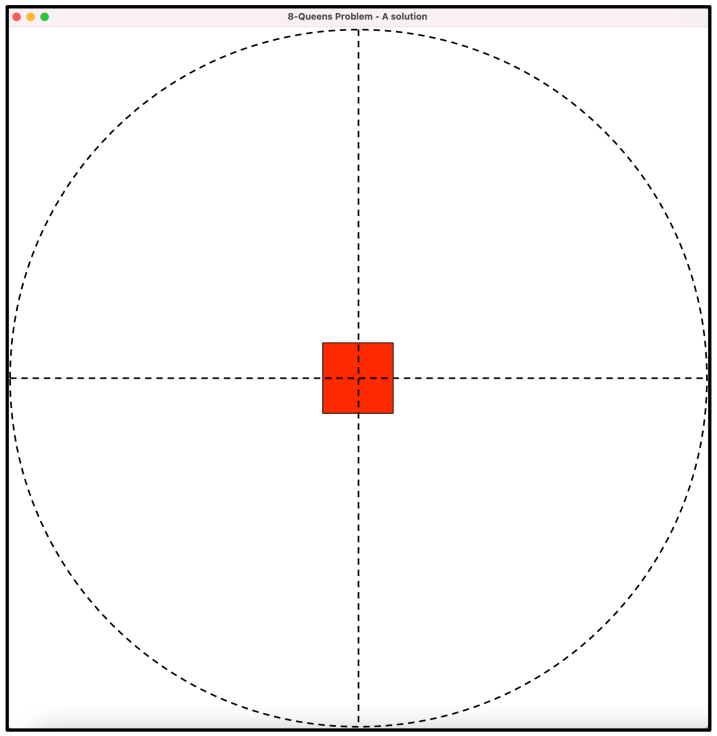


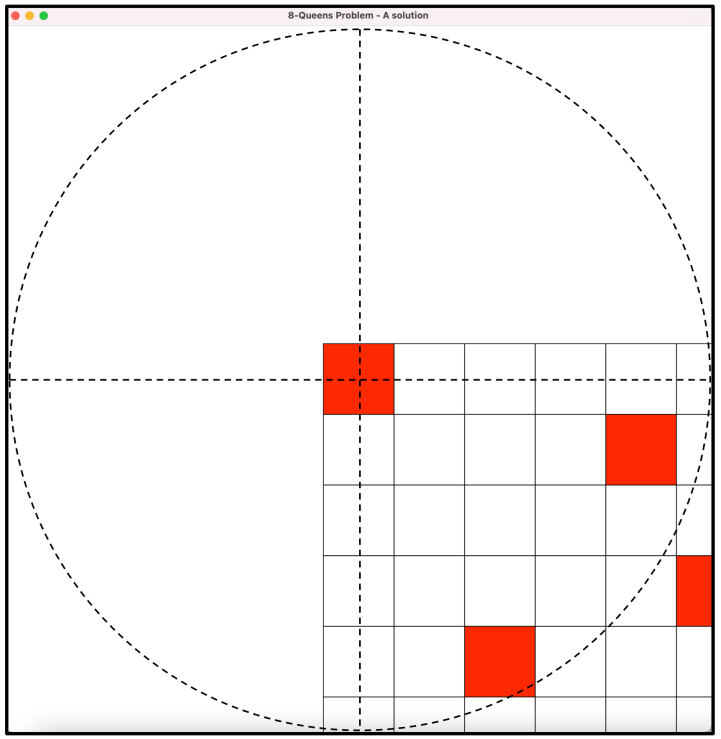
We can do a bit more though

The images that we have been creating up to now have been **centered** on the **origin** of the **coordinate system**

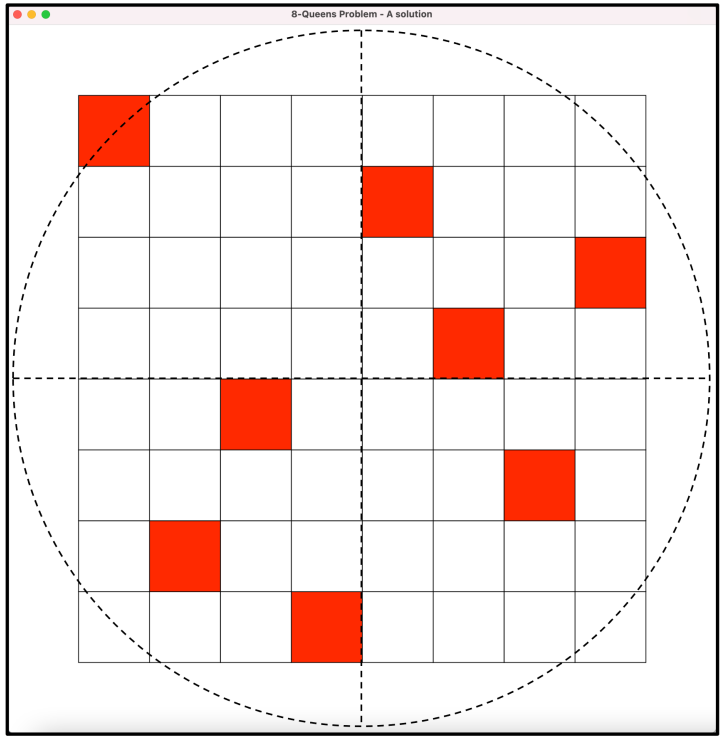


But we can use an image's **at** function to specify a desired position for the image



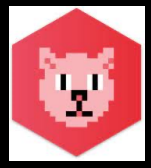



```
val (x,y) = …
squareImage = squareImageAtOrigin.at(x,y)
```




```
val (x,y) = …
boardImage = boardImageAtOrigin.at(x,y)
```

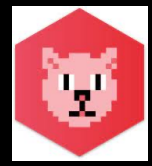
And if we are prepared to do that, we can further simplify the way we combine images



```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

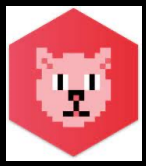
```scala
val on = Monoid.instance[Image](Image.empty, _ on _)
```



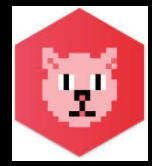
```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll beside)(above)
```

```scala
val on = Monoid.instance[Image](Image.empty, _ on _)
```

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll on)(on)
```



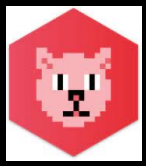
```scala
val on = Monoid.instance[Image](Image.empty, _ on _)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll on)(on)
```



```scala
val on = Monoid.instance[Image](Image.empty, _ on _)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll on)(on)
```

```scala
given Monoid.instance[Image] = Monoid.instance[Image](Image.empty, _ on _)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

FULL RECAP



```
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```

FULL RECAP

```scala
def combine(imageGrid: List[List[Image]]): Image =
  imageGrid
    .map(_.reduce(_ beside _))
    .reduce(_ above _)
```

```scala
given Monoid.instance[Image] = Monoid.instance[Image](Image.empty, _ on _)

def combine(imageGrid: List[List[Image]]): Image =
  imageGrid.foldMap(_ combineAll)
```

Earlier we looked at the number of ways of placing 8 queens on the board, one queen per row

$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16{,}777{,}216$ i.e. # of permutations of 8 queens (repetition allowed)

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$ " " " " " without repetition

Earlier we looked at the number of ways of placing 8 queens on the board, one queen per row

$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 = 8^8 = 16{,}777{,}216$ i.e. # of permutations of 8 queens (repetition allowed)

$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 8! = 40{,}320$ " " " " " without repetition

So the formulas for arbitrary N are the following:

N^N	# of permutations of N queens (repetition allowed)
$N!$	# of permutations of N queens without repetition

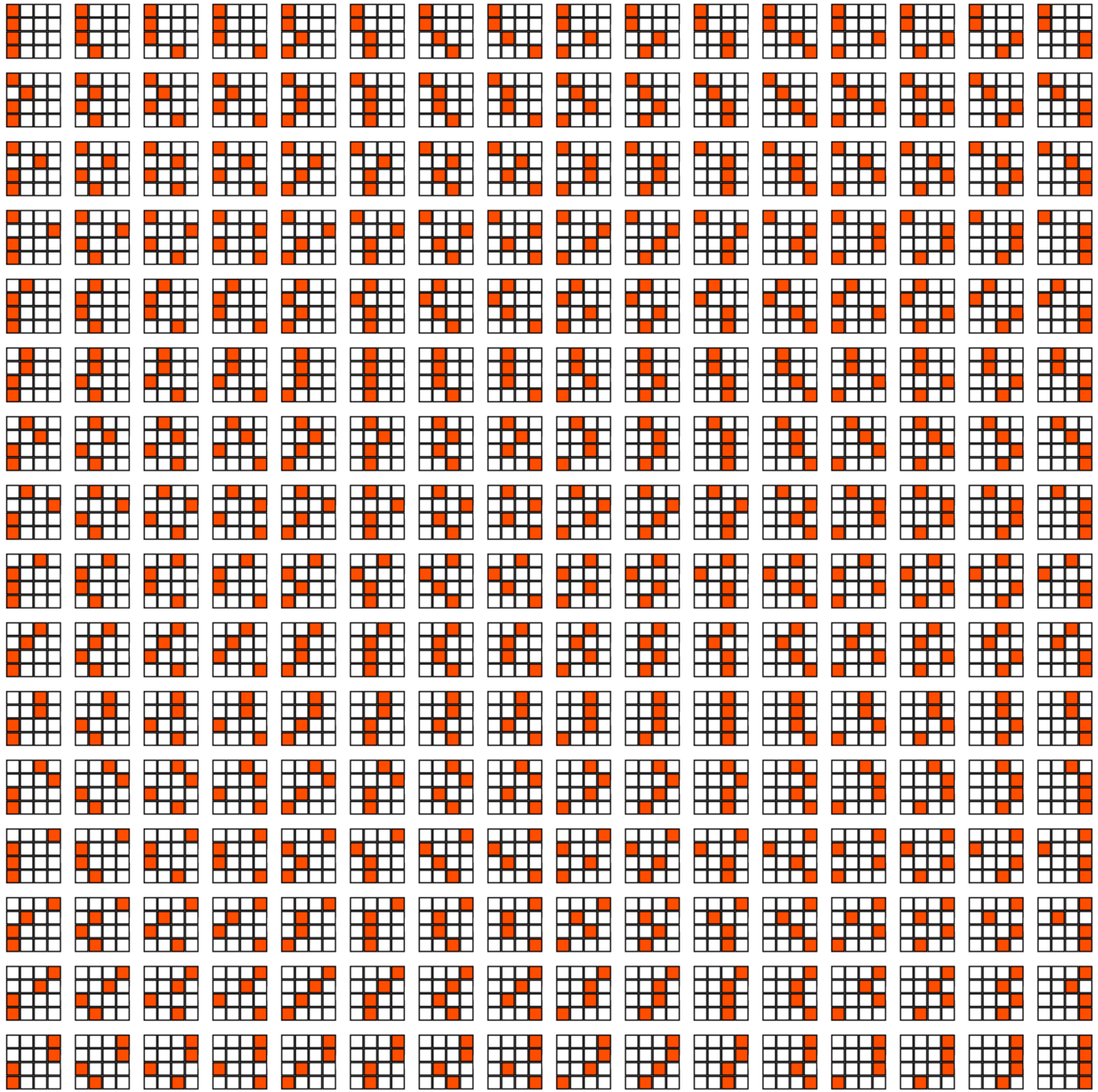
How do we compute the permutations of **N** queens, e.g. for **N**=**4**?

```
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

```
List(
  List(1, 1, 1, 1), List(1, 1, 1, 2), List(1, 1, 1, 3), List(1, 1, 1, 4), List(1, 1, 2, 1), List(1, 1, 2, 2),
  List(1, 1, 2, 3), List(1, 1, 2, 4), List(1, 1, 3, 1), List(1, 1, 3, 2), List(1, 1, 3, 3), List(1, 1, 3, 4),
  List(1, 1, 4, 1), List(1, 1, 4, 2), List(1, 1, 4, 3), List(1, 1, 4, 4), List(1, 2, 1, 1), List(1, 2, 1, 2),
  List(1, 2, 1, 3), List(1, 2, 1, 4), List(1, 2, 2, 1), List(1, 2, 2, 2), List(1, 2, 2, 3), List(1, 2, 2, 4),
  List(1, 2, 3, 1), List(1, 2, 3, 2), List(1, 2, 3, 3), List(1, 2, 3, 4), List(1, 2, 4, 1), List(1, 2, 4, 2),
  List(1, 2, 4, 3), List(1, 2, 4, 4), List(1, 3, 1, 1), List(1, 3, 1, 2), List(1, 3, 1, 3), List(1, 3, 1, 4),
  List(1, 3, 2, 1), List(1, 3, 2, 2), List(1, 3, 2, 3), List(1, 3, 2, 4), List(1, 3, 3, 1), List(1, 3, 3, 2),
  List(1, 3, 3, 3), List(1, 3, 3, 4), List(1, 3, 4, 1), List(1, 3, 4, 2), List(1, 3, 4, 3), List(1, 3, 4, 4),
  List(1, 4, 1, 1), List(1, 4, 1, 2), List(1, 4, 1, 3), List(1, 4, 1, 4), List(1, 4, 2, 1), List(1, 4, 2, 2),
  List(1, 4, 2, 3), List(1, 4, 2, 4), List(1, 4, 3, 1), List(1, 4, 3, 2), List(1, 4, 3, 3), List(1, 4, 3, 4),
  List(1, 4, 4, 1), List(1, 4, 4, 2), List(1, 4, 4, 3), List(1, 4, 4, 4), List(2, 1, 1, 1), List(2, 1, 1, 2),
  List(2, 1, 1, 3), List(2, 1, 1, 4), List(2, 1, 2, 1), List(2, 1, 2, 2), List(2, 1, 2, 3), List(2, 1, 2, 4),
  List(2, 1, 3, 1), List(2, 1, 3, 2), List(2, 1, 3, 3), List(2, 1, 3, 4), List(2, 1, 4, 1), List(2, 1, 4, 2),
  List(2, 1, 4, 3), List(2, 1, 4, 4), List(2, 2, 1, 1), List(2, 2, 1, 2), List(2, 2, 1, 3), List(2, 2, 1, 4),
  List(2, 2, 2, 1), List(2, 2, 2, 2), List(2, 2, 2, 3), List(2, 2, 2, 4), List(2, 2, 3, 1), List(2, 2, 3, 2),
  List(2, 2, 3, 3), List(2, 2, 3, 4), List(2, 2, 4, 1), List(2, 2, 4, 2), List(2, 2, 4, 3), List(2, 2, 4, 4),
  List(2, 3, 1, 1), List(2, 3, 1, 2), List(2, 3, 1, 3), List(2, 3, 1, 4), List(2, 3, 2, 1), List(2, 3, 2, 2),
  List(2, 3, 2, 3), List(2, 3, 2, 4), List(2, 3, 3, 1), List(2, 3, 3, 2), List(2, 3, 3, 3), List(2, 3, 3, 4),
  List(2, 3, 4, 1), List(2, 3, 4, 2), List(2, 3, 4, 3), List(2, 3, 4, 4), List(2, 4, 1, 1), List(2, 4, 1, 2),
  List(2, 4, 1, 3), List(2, 4, 1, 4), List(2, 4, 2, 1), List(2, 4, 2, 2), List(2, 4, 2, 3), List(2, 4, 2, 4),
  List(2, 4, 3, 1), List(2, 4, 3, 2), List(2, 4, 3, 3), List(2, 4, 3, 4), List(2, 4, 4, 1), List(2, 4, 4, 2),
  List(2, 4, 4, 3), List(2, 4, 4, 4), List(3, 1, 1, 1), List(3, 1, 1, 2), List(3, 1, 1, 3), List(3, 1, 1, 4),
  List(3, 1, 2, 1), List(3, 1, 2, 2), List(3, 1, 2, 3), List(3, 1, 2, 4), List(3, 1, 3, 1), List(3, 1, 3, 2),
  List(3, 1, 3, 3), List(3, 1, 3, 4), List(3, 1, 4, 1), List(3, 1, 4, 2), List(3, 1, 4, 3), List(3, 1, 4, 4),
  List(3, 2, 1, 1), List(3, 2, 1, 2), List(3, 2, 1, 3), List(3, 2, 1, 4), List(3, 2, 2, 1), List(3, 2, 2, 2),
  List(3, 2, 2, 3), List(3, 2, 2, 4), List(3, 2, 3, 1), List(3, 2, 3, 2), List(3, 2, 3, 3), List(3, 2, 3, 4),
  List(3, 2, 4, 1), List(3, 2, 4, 2), List(3, 2, 4, 3), List(3, 2, 4, 4), List(3, 3, 1, 1), List(3, 3, 1, 2),
  List(3, 3, 1, 3), List(3, 3, 1, 4), List(3, 3, 2, 1), List(3, 3, 2, 2), List(3, 3, 2, 3), List(3, 3, 2, 4),
  List(3, 3, 3, 1), List(3, 3, 3, 2), List(3, 3, 3, 3), List(3, 3, 3, 4), List(3, 3, 4, 1), List(3, 3, 4, 2),
  List(3, 3, 4, 3), List(3, 3, 4, 4), List(3, 4, 1, 1), List(3, 4, 1, 2), List(3, 4, 1, 3), List(3, 4, 1, 4),
  List(3, 4, 2, 1), List(3, 4, 2, 2), List(3, 4, 2, 3), List(3, 4, 2, 4), List(3, 4, 3, 1), List(3, 4, 3, 2),
  List(3, 4, 3, 3), List(3, 4, 3, 4), List(3, 4, 4, 1), List(3, 4, 4, 2), List(3, 4, 4, 3), List(3, 4, 4, 4),
  List(4, 1, 1, 1), List(4, 1, 1, 2), List(4, 1, 1, 3), List(4, 1, 1, 4), List(4, 1, 2, 1), List(4, 1, 2, 2),
  List(4, 1, 2, 3), List(4, 1, 2, 4), List(4, 1, 3, 1), List(4, 1, 3, 2), List(4, 1, 3, 3), List(4, 1, 3, 4),
  List(4, 1, 4, 1), List(4, 1, 4, 2), List(4, 1, 4, 3), List(4, 1, 4, 4), List(4, 2, 1, 1), List(4, 2, 1, 2),
  List(4, 2, 1, 3), List(4, 2, 1, 4), List(4, 2, 2, 1), List(4, 2, 2, 2), List(4, 2, 2, 3), List(4, 2, 2, 4),
  List(4, 2, 3, 1), List(4, 2, 3, 2), List(4, 2, 3, 3), List(4, 2, 3, 4), List(4, 2, 4, 1), List(4, 2, 4, 2),
  List(4, 2, 4, 3), List(4, 2, 4, 4), List(4, 3, 1, 1), List(4, 3, 1, 2), List(4, 3, 1, 3), List(4, 3, 1, 4),
  List(4, 3, 2, 1), List(4, 3, 2, 2), List(4, 3, 2, 3), List(4, 3, 2, 4), List(4, 3, 3, 1), List(4, 3, 3, 2),
  List(4, 3, 3, 3), List(4, 3, 3, 4), List(4, 3, 4, 1), List(4, 3, 4, 2), List(4, 3, 4, 3), List(4, 3, 4, 4),
  List(4, 4, 1, 1), List(4, 4, 1, 2), List(4, 4, 1, 3), List(4, 4, 1, 4), List(4, 4, 2, 1), List(4, 4, 2, 2),
  List(4, 4, 2, 3), List(4, 4, 2, 4), List(4, 4, 3, 1), List(4, 4, 3, 2), List(4, 4, 3, 3), List(4, 4, 3, 4),
  List(4, 4, 4, 1), List(4, 4, 4, 2), List(4, 4, 4, 3), List(4, 4, 4, 4))
```

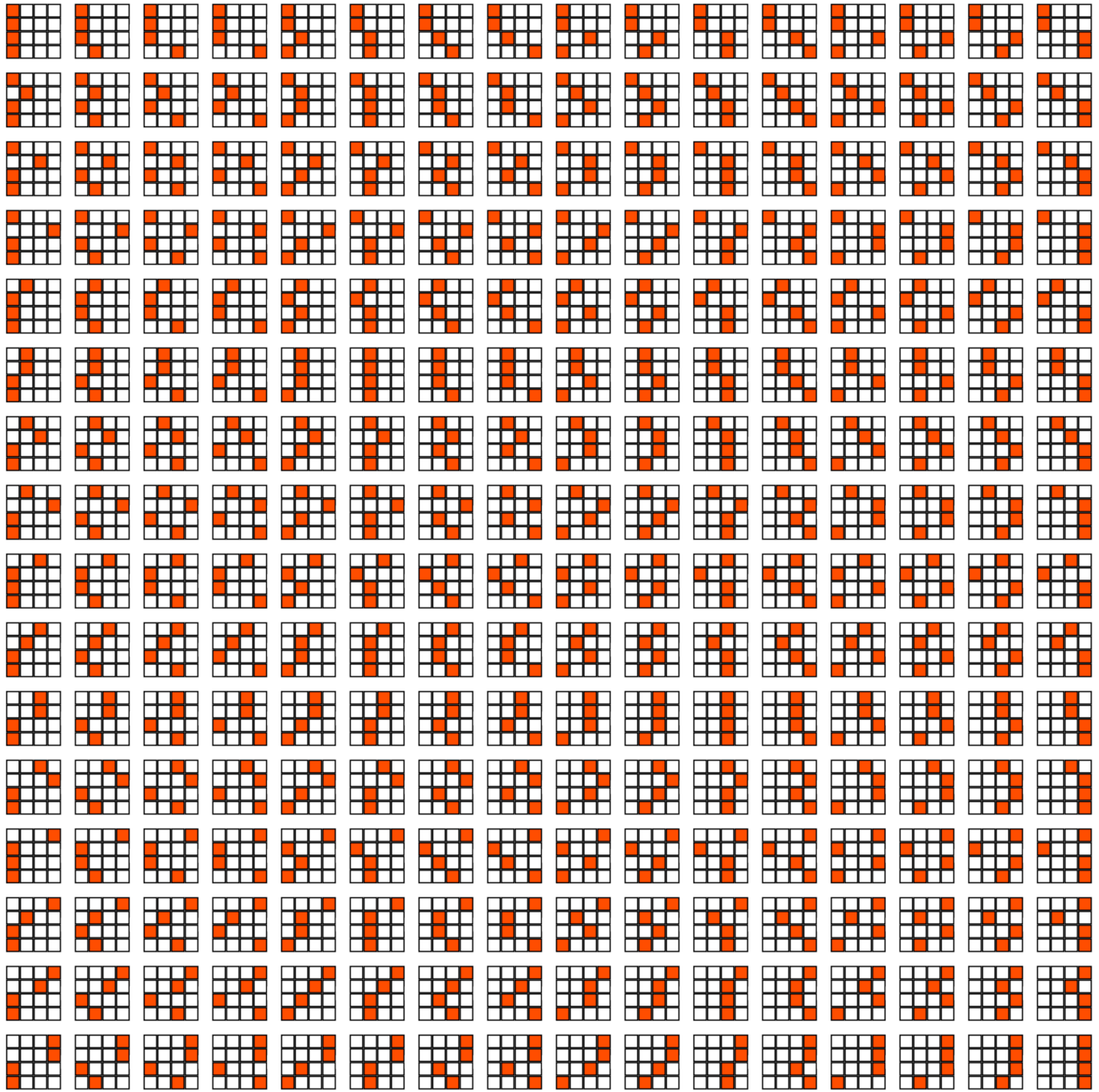
```scala
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList

println(s"# of results = ${results.size}")
println(s"# of permutations with repetition allowed = ${4 * 4 * 4 * 4}")    // N^N
println(s"# of permutations with repetition disallowed = ${4 * 3 * 2 * 1}") // N!
```

```
# of results = 256 (size of result)
# of permutations with repetition allowed = 256 (4⁴)
# of permutations with repetition disallowed = 24 (4!)
```



N = 4

# of permutations = 4^4 = 256



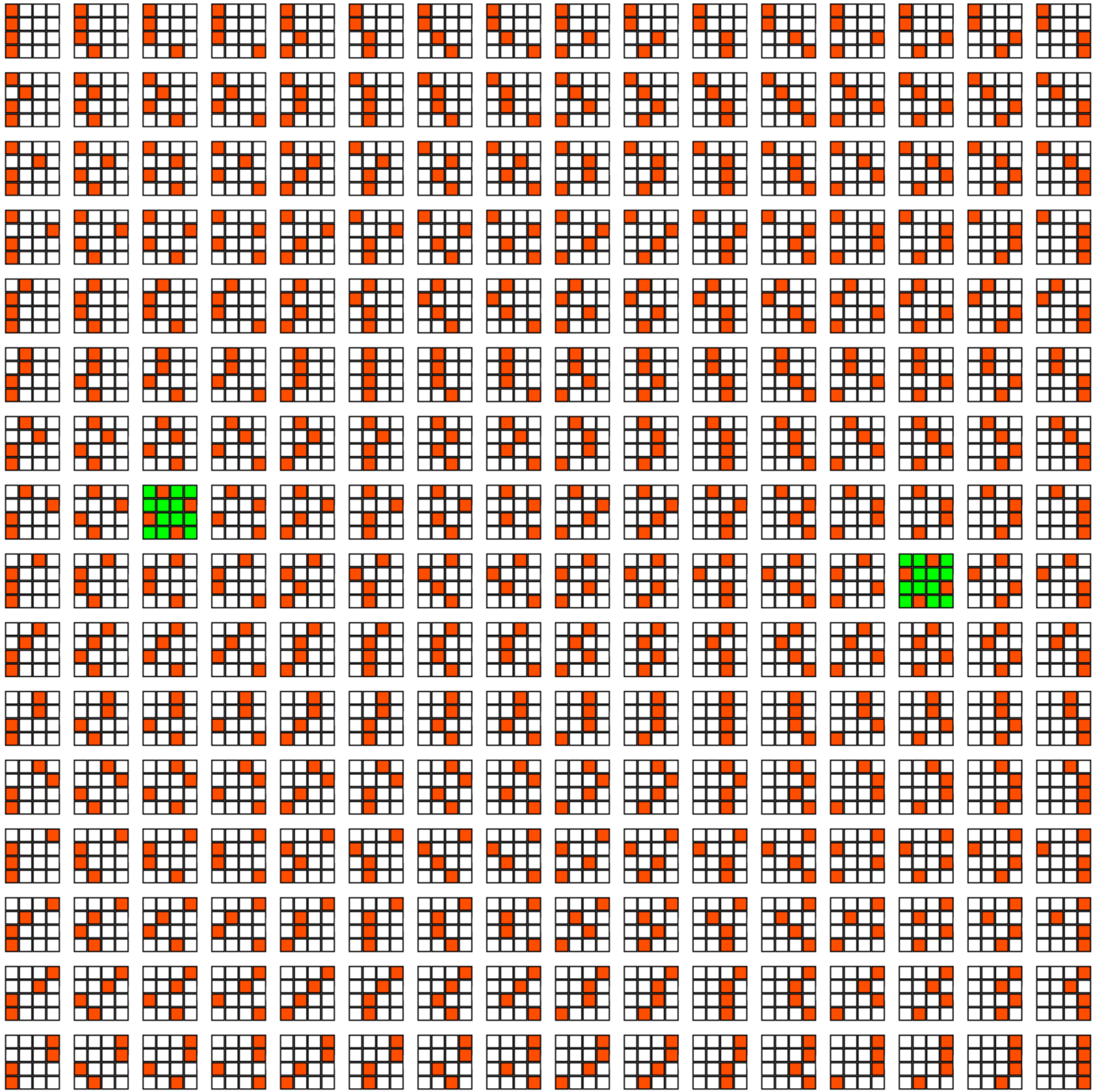
N = 4

# of permutations = $4^4 = 256$

How many of these are solutions?

N	N-queens solution count
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2,680
12	14,200
13	73,712
14	365,596
15	2,279,184
16	14,772,512
17	95,815,104
18	666,090,624
19	4,968,057,848
20	39,029,188,884
21	314,666,222,712
22	2,691,008,701,644
23	24,233,937,684,440
24	227,514,171,973,736
25	2,207,893,435,808,352
26	22,317,699,616,364,044
27	234,907,967,154,122,528

data source: https://en.wikipedia.org/wiki/Eight_queens_puzzle



N = 4
# of permutations = 4^4 = 256
# of solutions = 2

How do we find the solutions?

Martin Odersky

A full solution can not be found in a single step.

It needs to be built up gradually, by occupying successive rows with queens.

This suggests a **recursive algorithm**.

Martin Odersky

Assume you have already generated all solutions of placing k queens on a board of size N x N, where k is less than N.

…

Now, to place the next queen in row $k + 1$, generate all possible extensions of each previous solution by one more queen.

Martin Odersky

Assume you have already generated all solutions of placing k queens on a board of size N x N, where k is less than N.

…

Now, to place the next queen in row $k + 1$, generate all possible extensions of each previous solution by one more queen.

This yields another list of solutions lists, this time of length $k + 1$.

Continue the process until you have obtained all solutions of the size of the chess-board N.

```scala
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

```
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

simplify by using recursion

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

```
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

simplify by using recursion

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

```scala
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

simplify by using recursion

```scala
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

```
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

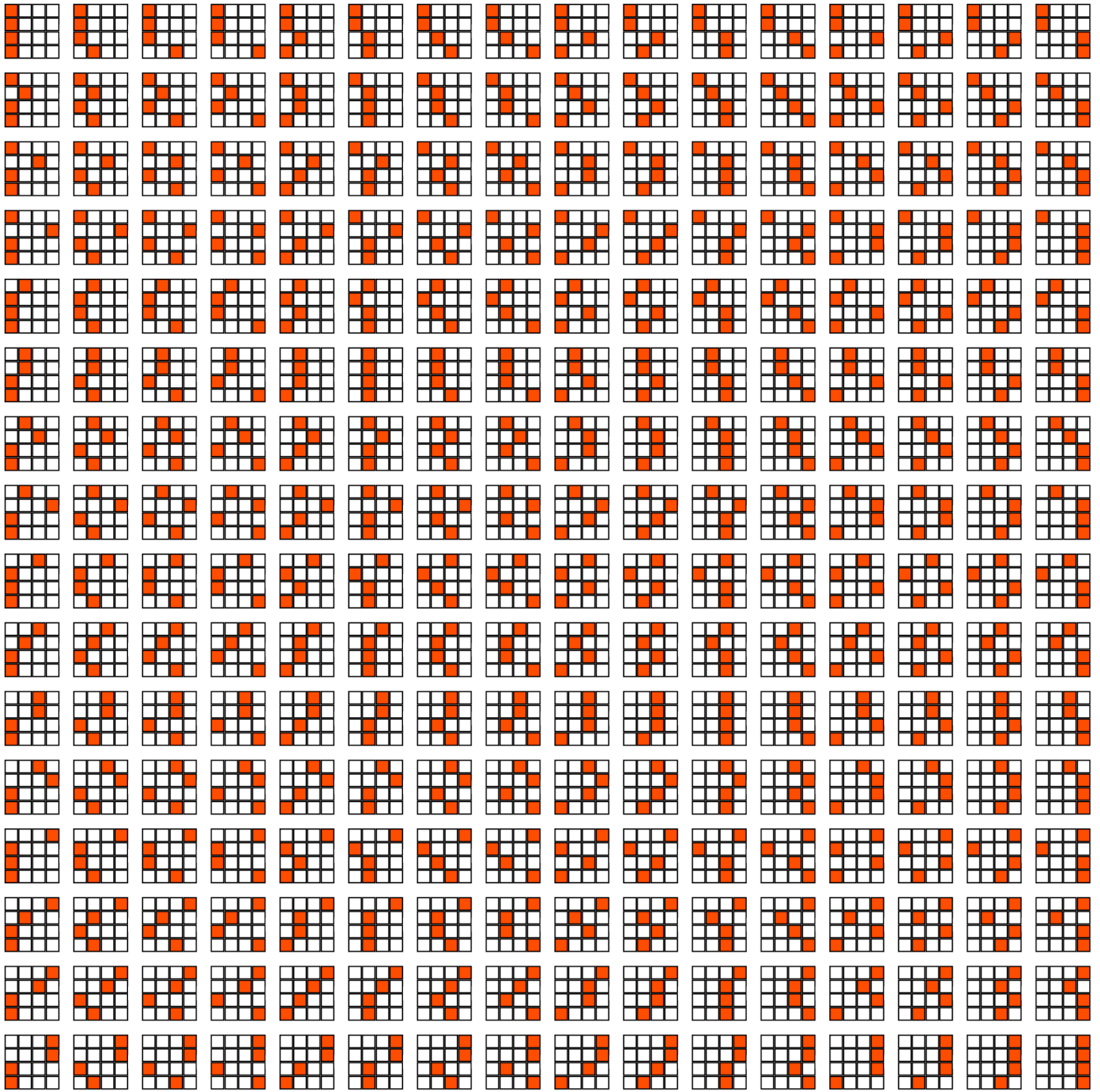
simplify by using recursion

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

```scala
def permutations(): List[List[Int]] =
  { for
      firstQueen  <- 1 to 4
      secondQueen <- 1 to 4
      thirdQueen  <- 1 to 4
      fourthQueen <- 1 to 4
      queens = List(firstQueen, secondQueen, thirdQueen, fourthQueen)
    yield queens
  }.toList
```

simplify by using recursion

```scala
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

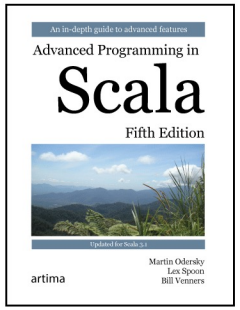


N = 4
# of permutations = 4^4 = 256

Same result as before using recursion

Martin Odersky

This **algorithmic idea** is embodied in function **placeQueens**.



Functional Programming Principles in Scala

1.18K subscribers

```scala
def queens(n: Int): List[List[(Int,Int)]] = {
  def placeQueens(k: Int): List[List[(Int,Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
    } yield queen :: queens
  placeQueens(n)
}
```

```scala
def queens(n: Int): Set[List[Int]] = {
  def placeQueens(k: Int): Set[List[Int]] =
    if (k == 0)
      Set(List())
    else
      for {
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
    } yield col :: queens
  placeQueens(n)
}
```

```scala
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // same row
  q1._2 == q2._2 || // same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

```scala
def isSafe(col: Int, queens: List[Int]): Boolean = {
  val row = queens.length
  val queensWithRow = (row - 1 to 0 by -1) zip queens
  queensWithRow forall
    { case (r, c) => col != c && math.abs(col - c) != row - r }
}
```

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen  <- 1 to 4
    yield queen :: queens
```

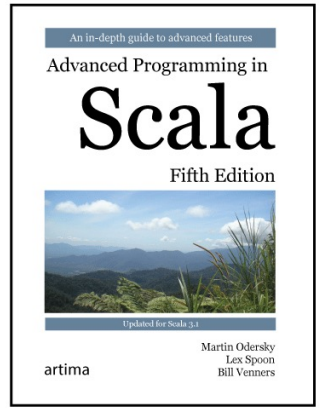
```
def queens(n: Int): List[List[(Int,Int)]] = {
  def placeQueens(k: Int): List[List[(Int,Int)]] =
    if (k == 0) List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
    } yield queen :: queens
  placeQueens(n)
}
```

Martin Odersky

The only remaining bit is the **isSafe** method, which is used to check whether a given queen is **in check** from any other element in a list of queens.

Here is the definition:

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // same row
  q1._2 == q2._2 || // same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```



The **isSafe** method expresses that a queen is safe with respect to some other queens if it is not **in check** from any other queen.

The **inCheck** method expresses that queens q1 and q2 are mutually **in check**.

It returns true in one of three cases:

1. If the two queens have the **same row coordinate**.
2. If the two queens have the **same column coordinate**.
3. If the two queens are on **the same diagonal** (*i.e.*, the difference between their rows and the difference between their columns are the same).

The first case – that the two queens have the same row coordinate – cannot happen in the application because **placeQueens** already takes care to place each queen in a different row. So you could remove the test without changing the functionality of the program.

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen <- 1 to 4
    yield queen :: queens
```

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen <- 1 to 4
    yield queen :: queens
```

use isSafe function

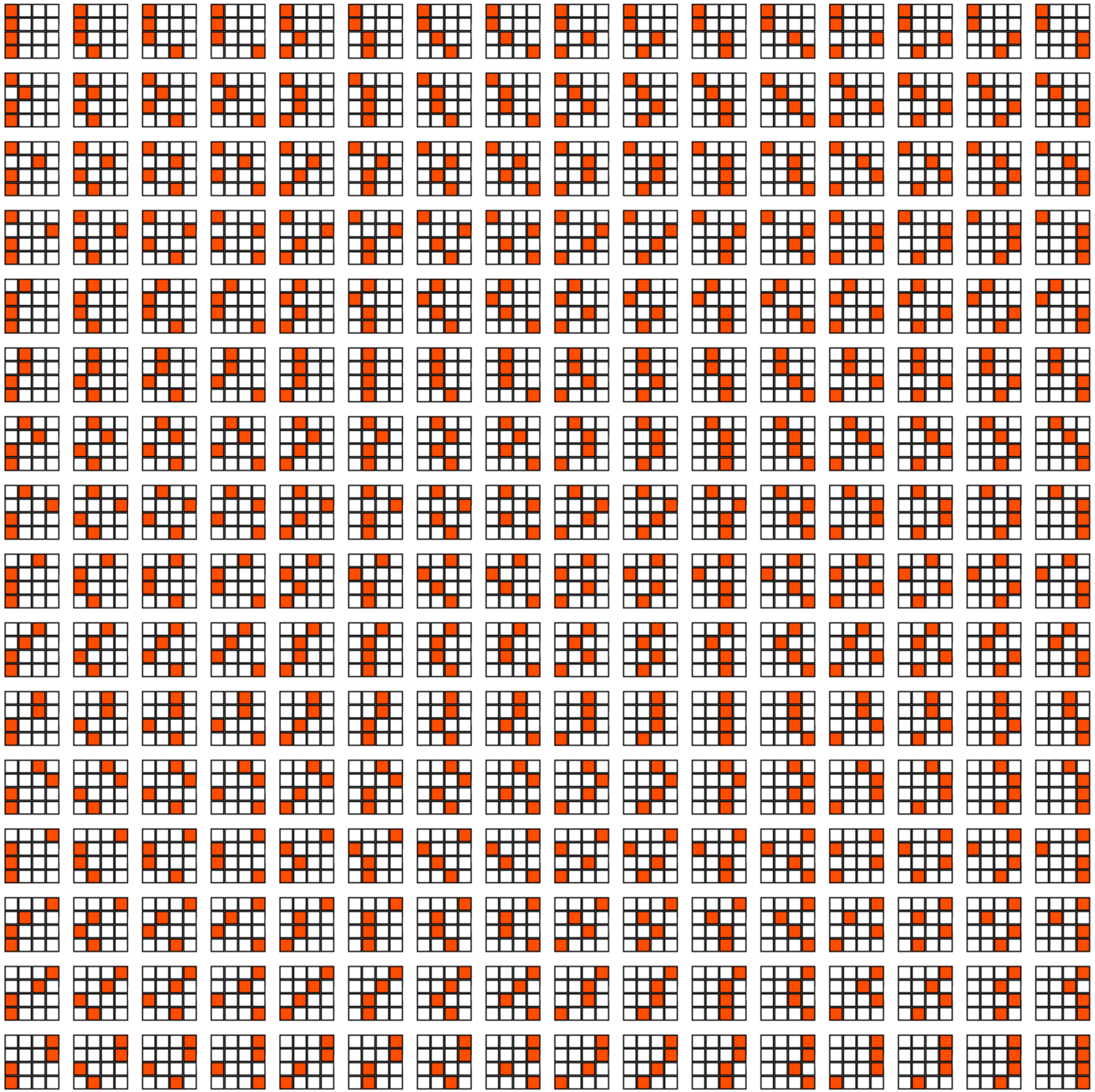
```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen <- 1 to 4
      if isSafe(queen,queens)
    yield queen :: queens
```

```
List(
  List(1, 1, 1, 1), List(1, 1, 1, 2), List(1, 1, 1, 3), List(1, 1, 1, 4), List(1, 1, 2, 1), List(1, 1, 2, 2),
  List(1, 1, 2, 3), List(1, 1, 2, 4), List(1, 1, 3, 1), List(1, 1, 3, 2), List(1, 1, 3, 3), List(1, 1, 3, 4),
  List(1, 1, 4, 1), List(1, 1, 4, 2), List(1, 1, 4, 3), List(1, 1, 4, 4), List(1, 2, 1, 1), List(1, 2, 1, 2),
  List(1, 2, 1, 3), List(1, 2, 1, 4), List(1, 2, 2, 1), List(1, 2, 2, 2), List(1, 2, 2, 3), List(1, 2, 2, 4),
  List(1, 2, 3, 1), List(1, 2, 3, 2), List(1, 2, 3, 3), List(1, 2, 3, 4), List(1, 2, 4, 1), List(1, 2, 4, 2),
  List(1, 2, 4, 3), List(1, 2, 4, 4), List(1, 3, 1, 1), List(1, 3, 1, 2), List(1, 3, 1, 3), List(1, 3, 1, 4),
  List(1, 3, 2, 1), List(1, 3, 2, 2), List(1, 3, 2, 3), List(1, 3, 2, 4), List(1, 3, 3, 1), List(1, 3, 3, 2),
  List(1, 3, 3, 3), List(1, 3, 3, 4), List(1, 3, 4, 1), List(1, 3, 4, 2), List(1, 3, 4, 3), List(1, 3, 4, 4),
  List(1, 4, 1, 1), List(1, 4, 1, 2), List(1, 4, 1, 3), List(1, 4, 1, 4), List(1, 4, 2, 1), List(1, 4, 2, 2),
  List(1, 4, 2, 3), List(1, 4, 2, 4), List(1, 4, 3, 1), List(1, 4, 3, 2), List(1, 4, 3, 3), List(1, 4, 3, 4),
  List(1, 4, 4, 1), List(1, 4, 4, 2), List(1, 4, 4, 3), List(1, 4, 4, 4), List(2, 1, 1, 1), List(2, 1, 1, 2),
  List(2, 1, 1, 3), List(2, 1, 1, 4), List(2, 1, 2, 1), List(2, 1, 2, 2), List(2, 1, 2, 3), List(2, 1, 2, 4),
  List(2, 1, 3, 1), List(2, 1, 3, 2), List(2, 1, 3, 3), List(2, 1, 3, 4), List(2, 1, 4, 1), List(2, 1, 4, 2),
  List(2, 1, 4, 3), List(2, 1, 4, 4), List(2, 2, 1, 1), List(2, 2, 1, 2), List(2, 2, 1, 3), List(2, 2, 1, 4),
  List(2, 2, 2, 1), List(2, 2, 2, 2), List(2, 2, 2, 3), List(2, 2, 2, 4), List(2, 2, 3, 1), List(2, 2, 3, 2),
  List(2, 2, 3, 3), List(2, 2, 3, 4), List(2, 2, 4, 1), List(2, 2, 4, 2), List(2, 2, 4, 3), List(2, 2, 4, 4),
  List(2, 3, 1, 1), List(2, 3, 1, 2), List(2, 3, 1, 3), List(2, 3, 1, 4), List(2, 3, 2, 1), List(2, 3, 2, 2),
  List(2, 3, 2, 3), List(2, 3, 2, 4), List(2, 3, 3, 1), List(2, 3, 3, 2), List(2, 3, 3, 3), List(2, 3, 3, 4),
  List(2, 3, 4, 1), List(2, 3, 4, 2), List(2, 3, 4, 3), List(2, 3, 4, 4), List(2, 4, 1, 1), List(2, 4, 1, 2),
  List(2, 4, 1, 3), List(2, 4, 1, 4), List(2, 4, 2, 1), List(2, 4, 2, 2), List(2, 4, 2, 3), List(2, 4, 2, 4),
  List(2, 4, 3, 1), List(2, 4, 3, 2), List(2, 4, 3, 3), List(2, 4, 3, 4), List(2, 4, 4, 1), List(2, 4, 4, 2),
  List(2, 4, 4, 3), List(2, 4, 4, 4), List(3, 1, 1, 1), List(3, 1, 1, 2), List(3, 1, 1, 3), List(3, 1, 1, 4),
  List(3, 1, 2, 1), List(3, 1, 2, 2), List(3, 1, 2, 3), List(3, 1, 2, 4), List(3, 1, 3, 1), List(3, 1, 3, 2),
  List(3, 1, 3, 3), List(3, 1, 3, 4), List(3, 1, 4, 1), List(3, 1, 4, 2), List(3, 1, 4, 3), List(3, 1, 4, 4),
  List(3, 2, 1, 1), List(3, 2, 1, 2), List(3, 2, 1, 3), List(3, 2, 1, 4), List(3, 2, 2, 1), List(3, 2, 2, 2),
  List(3, 2, 2, 3), List(3, 2, 2, 4), List(3, 2, 3, 1), List(3, 2, 3, 2), List(3, 2, 3, 3), List(3, 2, 3, 4),
  List(3, 2, 4, 1), List(3, 2, 4, 2), List(3, 2, 4, 3), List(3, 2, 4, 4), List(3, 3, 1, 1), List(3, 3, 1, 2),
  List(3, 3, 1, 3), List(3, 3, 1, 4), List(3, 3, 2, 1), List(3, 3, 2, 2), List(3, 3, 2, 3), List(3, 3, 2, 4),
  List(3, 3, 3, 1), List(3, 3, 3, 2), List(3, 3, 3, 3), List(3, 3, 3, 4), List(3, 3, 4, 1), List(3, 3, 4, 2),
  List(3, 3, 4, 3), List(3, 3, 4, 4), List(3, 4, 1, 1), List(3, 4, 1, 2), List(3, 4, 1, 3), List(3, 4, 1, 4),
  List(3, 4, 2, 1), List(3, 4, 2, 2), List(3, 4, 2, 3), List(3, 4, 2, 4), List(3, 4, 3, 1), List(3, 4, 3, 2),
  List(3, 4, 3, 3), List(3, 4, 3, 4), List(3, 4, 4, 1), List(3, 4, 4, 2), List(3, 4, 4, 3), List(3, 4, 4, 4),
  List(4, 1, 1, 1), List(4, 1, 1, 2), List(4, 1, 1, 3), List(4, 1, 1, 4), List(4, 1, 2, 1), List(4, 1, 2, 2),
  List(4, 1, 2, 3), List(4, 1, 2, 4), List(4, 1, 3, 1), List(4, 1, 3, 2), List(4, 1, 3, 3), List(4, 1, 3, 4),
  List(4, 1, 4, 1), List(4, 1, 4, 2), List(4, 1, 4, 3), List(4, 1, 4, 4), List(4, 2, 1, 1), List(4, 2, 1, 2),
  List(4, 2, 1, 3), List(4, 2, 1, 4), List(4, 2, 2, 1), List(4, 2, 2, 2), List(4, 2, 2, 3), List(4, 2, 2, 4),
  List(4, 2, 3, 1), List(4, 2, 3, 2), List(4, 2, 3, 3), List(4, 2, 3, 4), List(4, 2, 4, 1), List(4, 2, 4, 2),
  List(4, 2, 4, 3), List(4, 2, 4, 4), List(4, 3, 1, 1), List(4, 3, 1, 2), List(4, 3, 1, 3), List(4, 3, 1, 4),
  List(4, 3, 2, 1), List(4, 3, 2, 2), List(4, 3, 2, 3), List(4, 3, 2, 4), List(4, 3, 3, 1), List(4, 3, 3, 2),
  List(4, 3, 3, 3), List(4, 3, 3, 4), List(4, 3, 4, 1), List(4, 3, 4, 2), List(4, 3, 4, 3), List(4, 3, 4, 4),
  List(4, 4, 1, 1), List(4, 4, 1, 2), List(4, 4, 1, 3), List(4, 4, 1, 4), List(4, 4, 2, 1), List(4, 4, 2, 2),
  List(4, 4, 2, 3), List(4, 4, 2, 4), List(4, 4, 3, 1), List(4, 4, 3, 2), List(4, 4, 3, 3), List(4, 4, 3, 4),
  List(4, 4, 4, 1), List(4, 4, 4, 2), List(4, 4, 4, 3), List(4, 4, 4, 4))
```

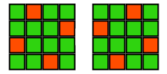
adding filtering
(isSafe function)

BEFORE AFTER

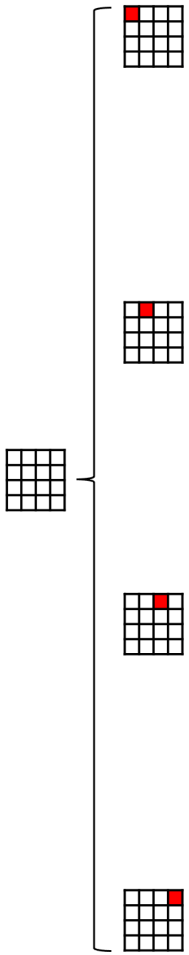
```
List(
  List(3,1,4,2),
  List(2,4,1,3))
)
```

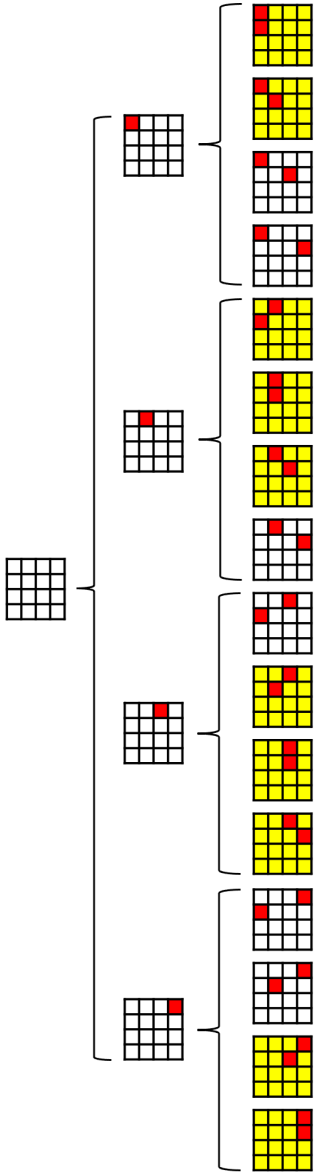


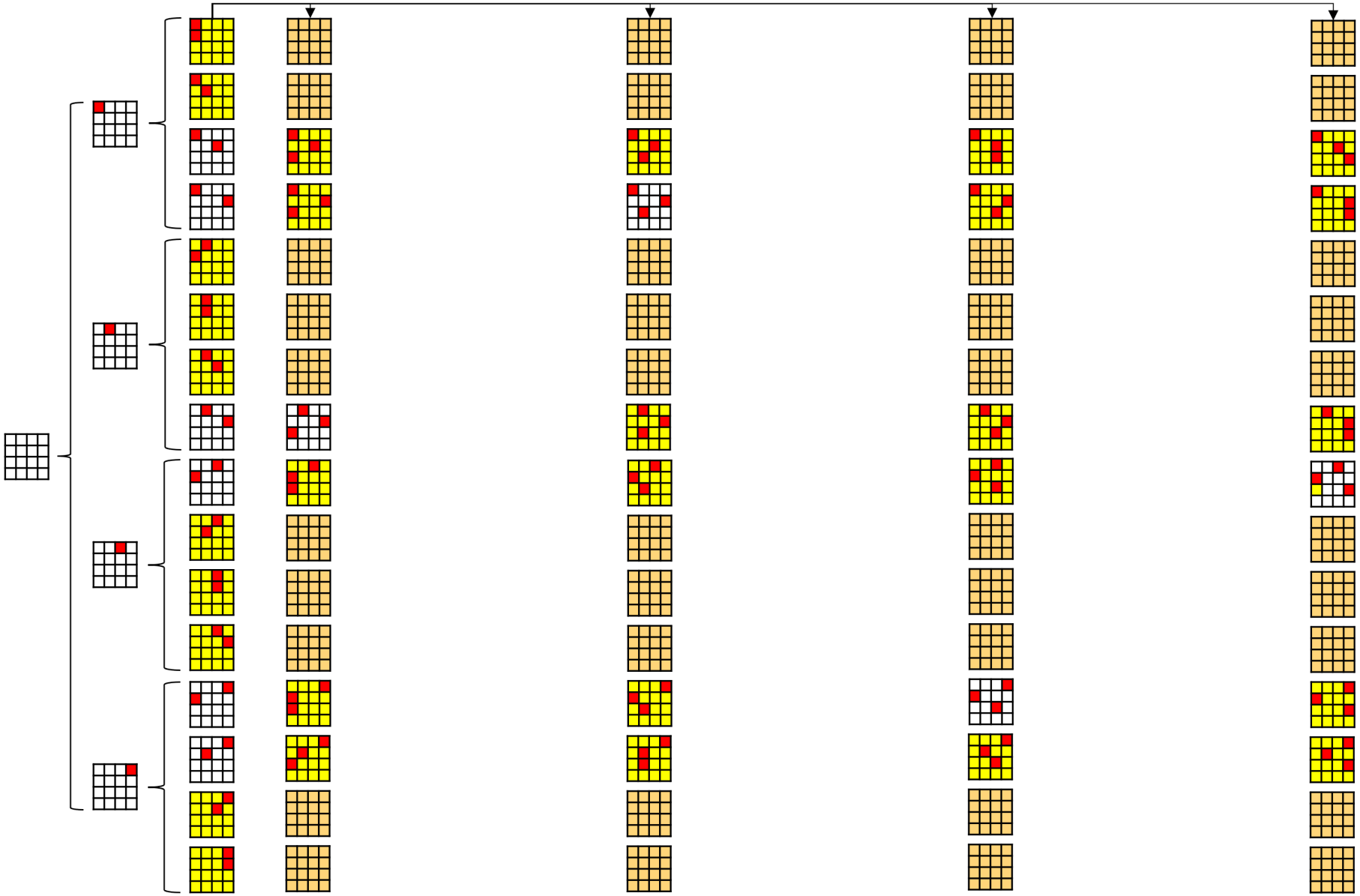
adding filtering (**isSafe** function)

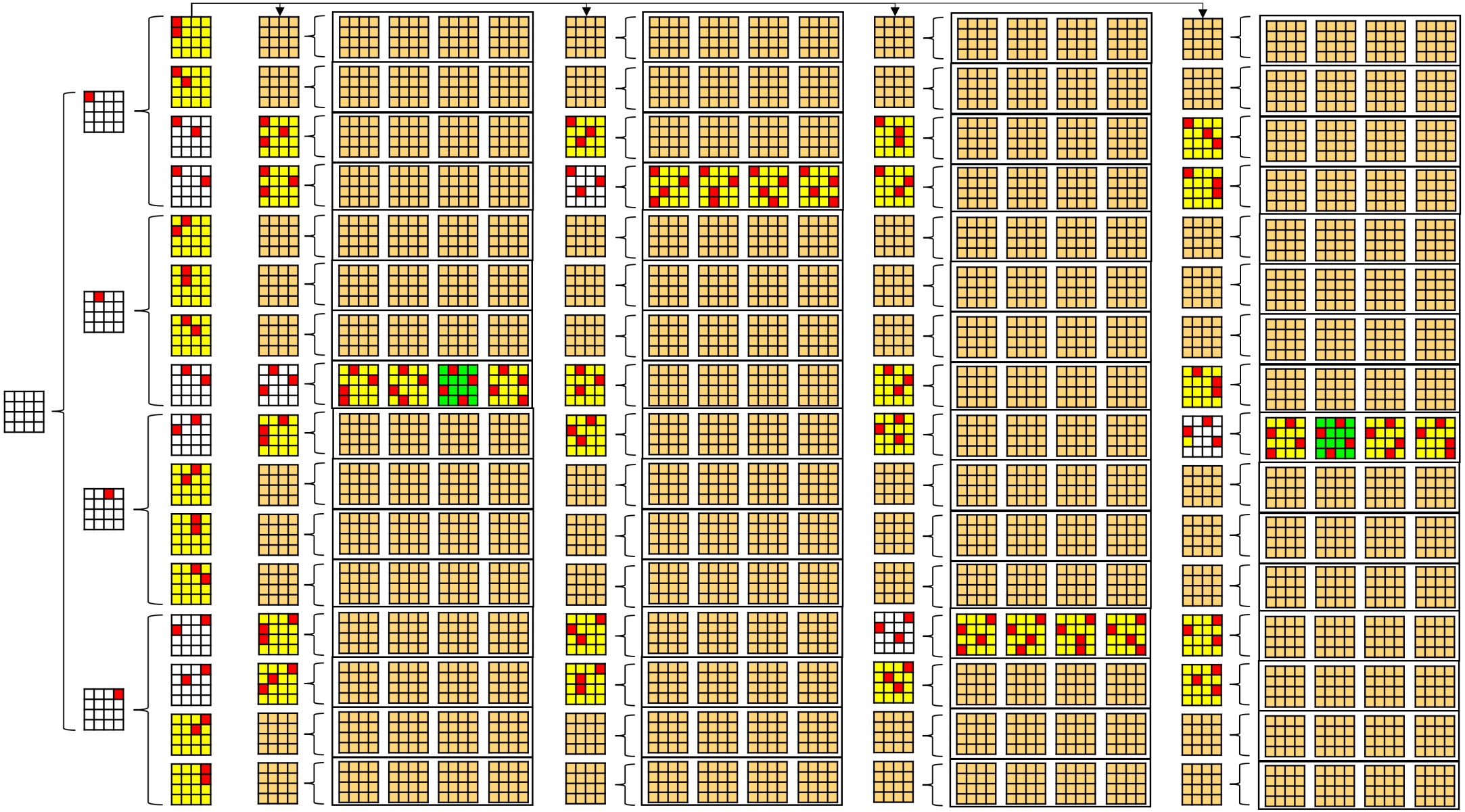


Let’s visualise how the filtering reduces the problem space



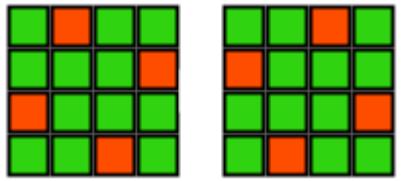




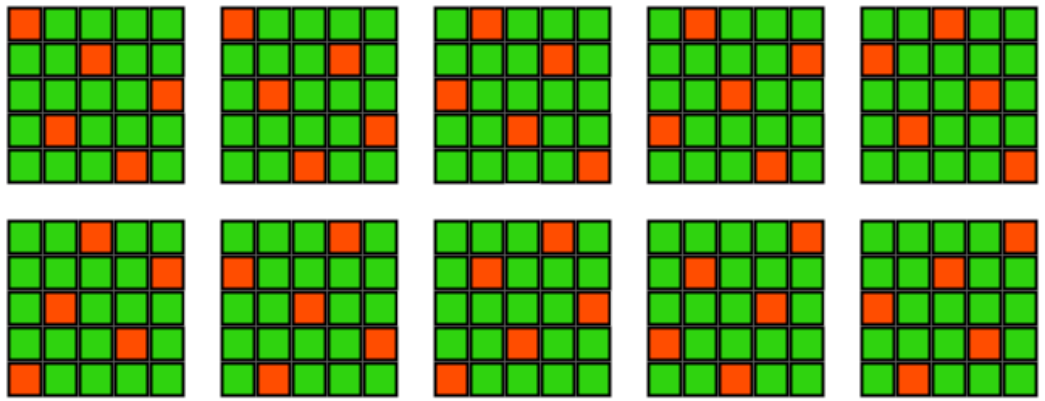


Candidate boards generated and tested: 60 (out of 256)

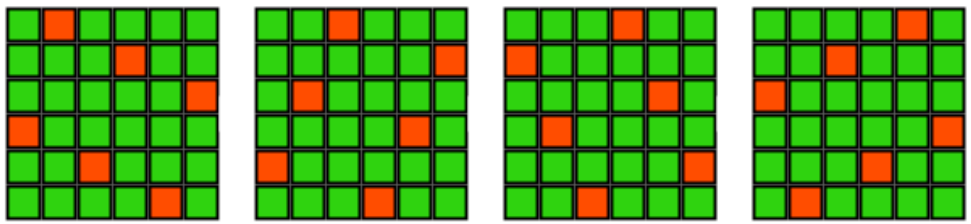
N = 4
2 solutions



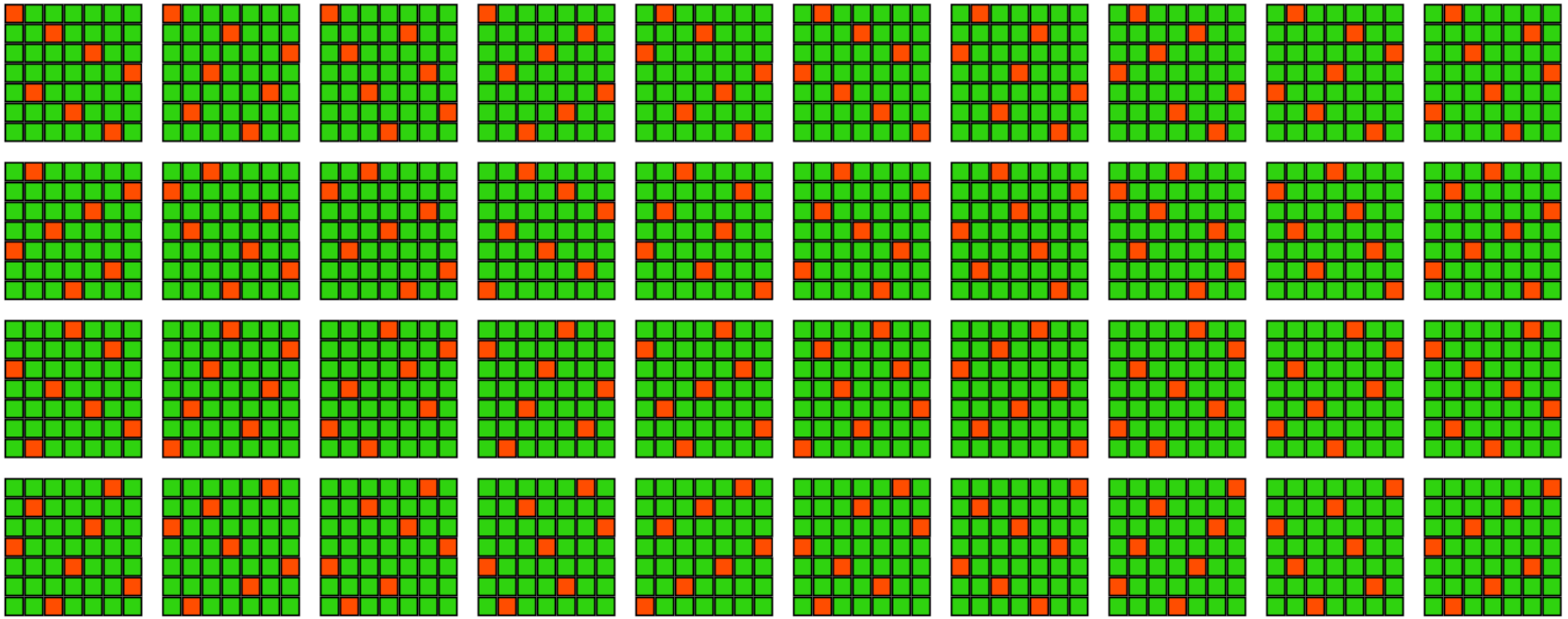
N = 5
10 solutions

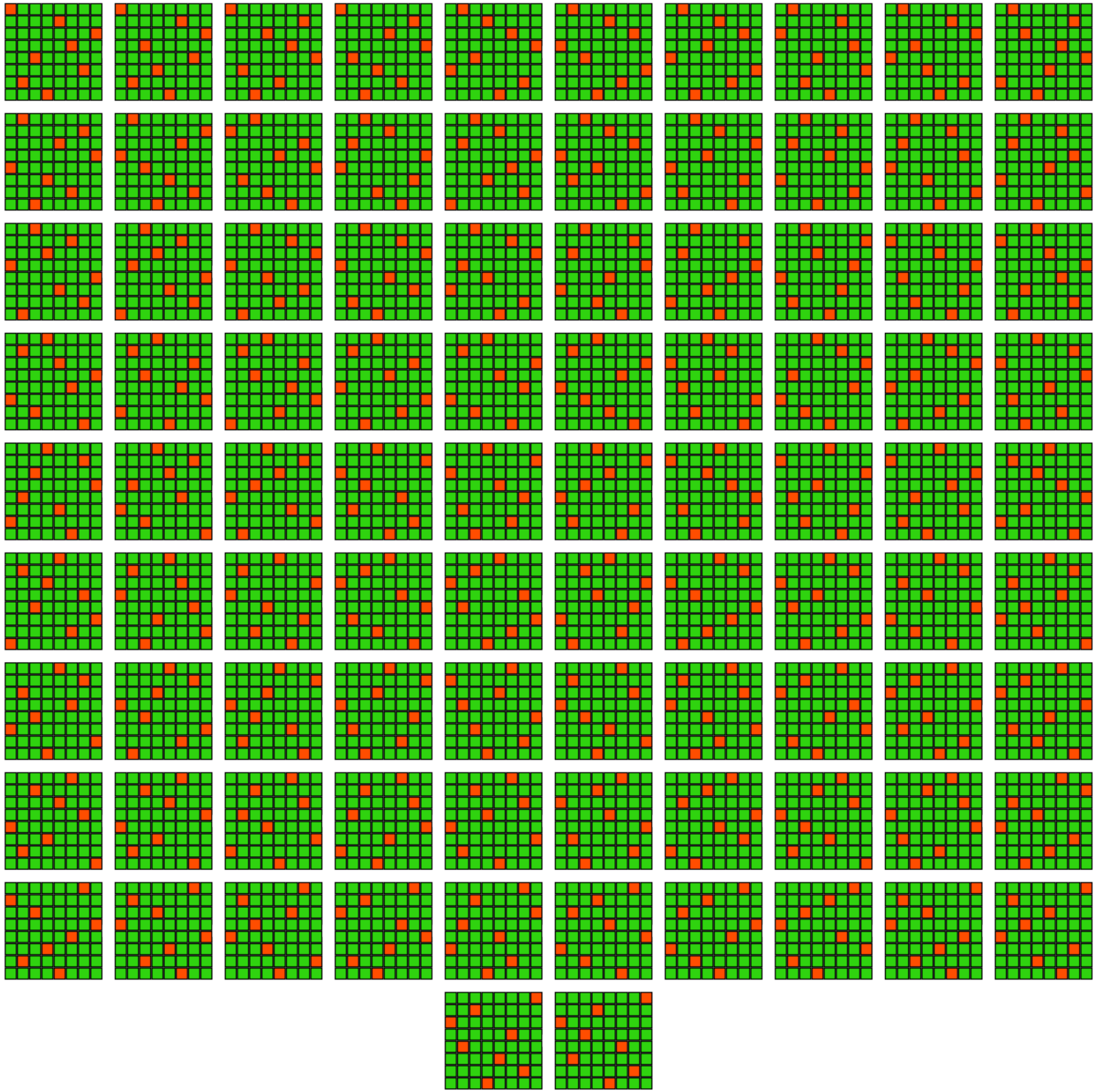


N = 6
4 solutions

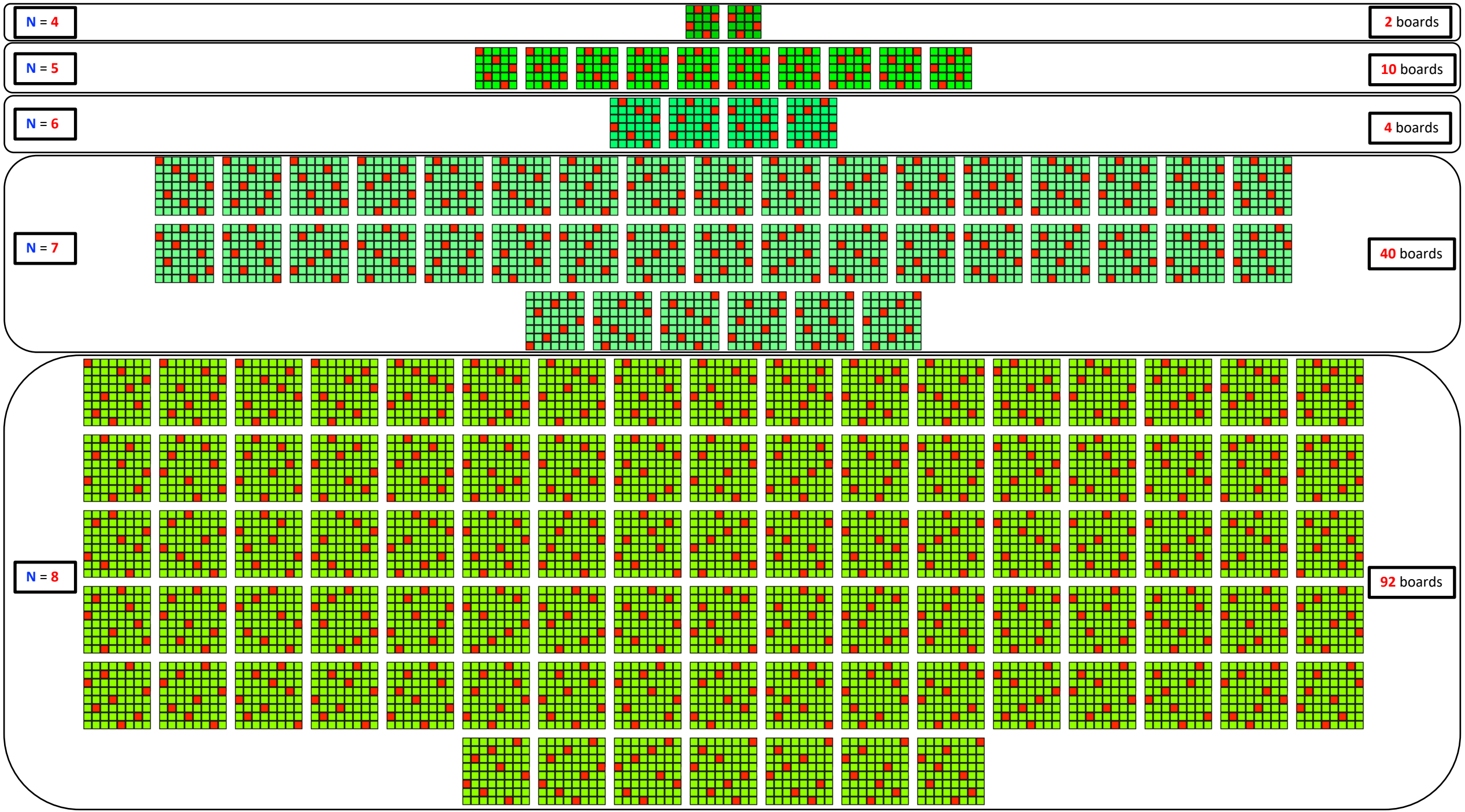


N = 7
40 solutions





N = 8
92 solutions


N = 4
2 boards
N = 5
10 boards
N = 6
4 boards
N = 7
40 boards
N = 8
92 boards

Scala

Haskell

Recursive Algorithm

```scala
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if safe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```

```haskell
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k = [queen:queens |
                     queens <- placeQueens(k-1),
                     queen <- [1..n],
                     safe queen queens]
```

```scala
def safe(queen: Int, queens: List[Int]): Boolean =
  val (row, column) = (queens.length, queen)
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)
  zipWithRows(queens) forall safe
```

```haskell
safe queen queens = all safe (zipWithRows queens)
  where
    safe (r,c) = c /= col && not (onDiagonal col row c r)
    row = length queens
    col = queen
```

```scala
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =
  math.abs(row - otherRow) ==  math.abs(column - otherColumn)
```

```haskell
onDiagonal row column otherRow otherColumn =
  abs (row - otherRow) == abs (column - otherColumn)
```

```scala
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =
  val rowCount = queens.length
  val rowNumbers = rowCount - 1 to 0 by -1
  rowNumbers zip queens
```

```haskell
zipWithRows queens = zip rowNumbers queens
  where
    rowCount = length queens
    rowNumbers = [rowCount-1,rowCount-2..0]
```

https://rosettacode.org/wiki/

On the **Rosetta Code** site, I came across a **Haskell N-Queens puzzle** program which

- does not use recursion
- uses a function called **foldM**
- it is succinct, but relies on comments to help understand how it works

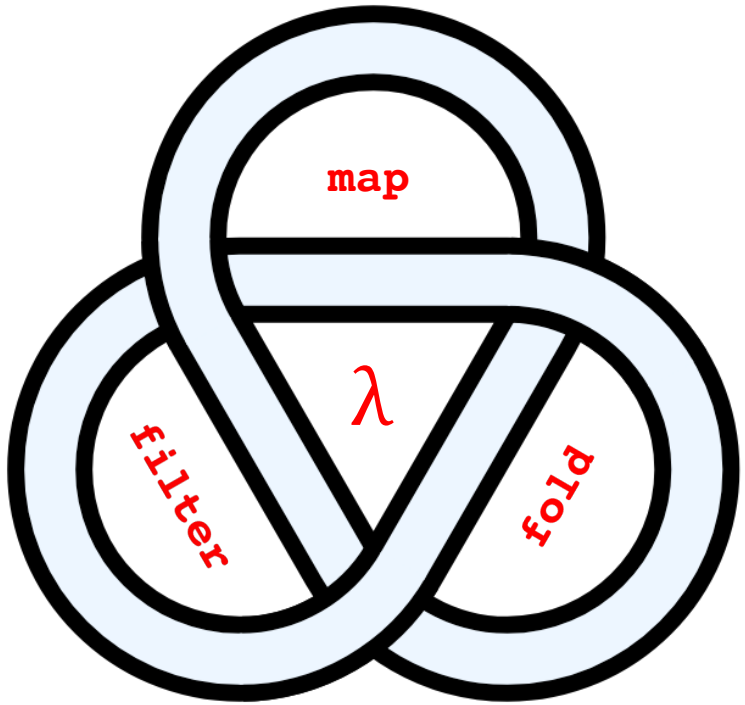
https://rosettacode.org/wiki/N-queens_problem#Haskell

In the rest of this talk we shall

- gain an understanding of the **foldM** function
- see how it can be used to write an **iterative solution** to the puzzle

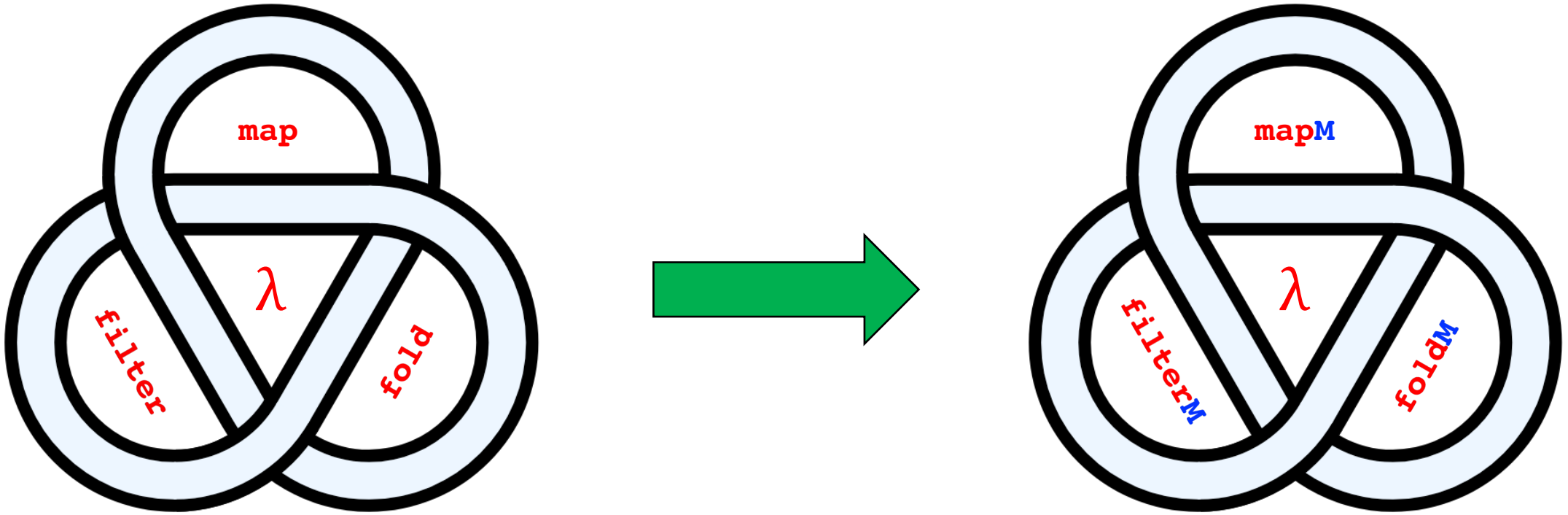
We are all very familiar with the **3 functions** that are the **bread**, **butter**, and **jam** of **Functional Programming**

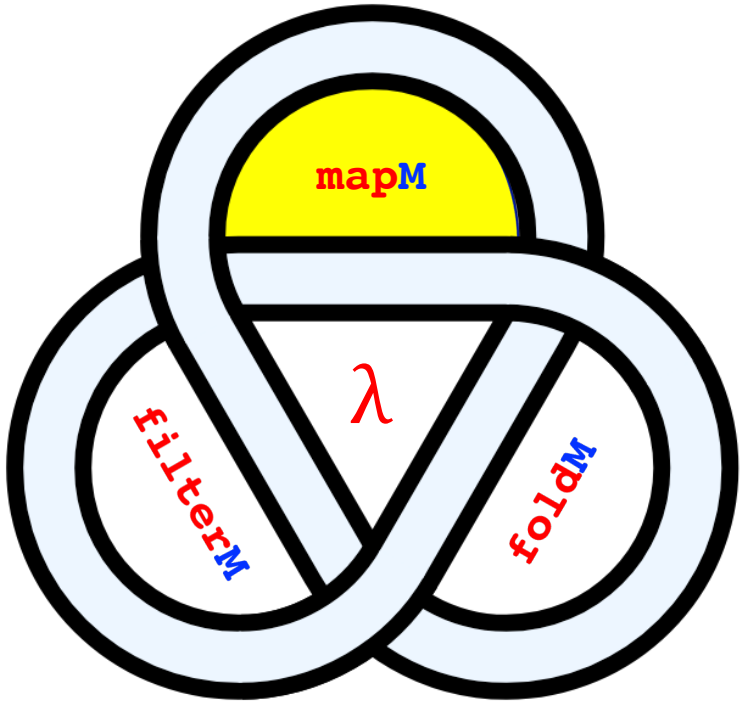
We are all very familiar with the **3 functions** that are the **bread**, **butter**, and **jam** of **Functional Programming**



I am (of course) referring to the triad of **map**, **filter** and **fold**

Let’s gain an understanding of the **foldM** function by looking at the **monadic** variant of the **triad**




mapM
λ
filterM
foldM

Graham Hutton

@haskellhutt

Generic functions

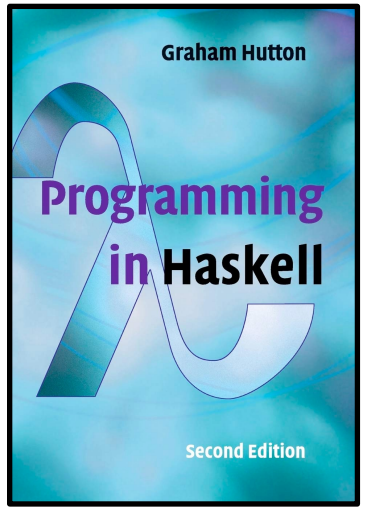
An important benefit of abstracting out the concept of monads is the ability to define generic functions that can be used with any monad.

…

For example, **a monadic version of the map function on list** can be defined as follows:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

Note that mapM has the same type as map, except that the argument function and the function itself now have monadic return types.



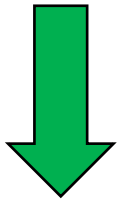
```
map :: (a -> b) -> [a] -> [b]
```

```haskell
map :: (a -> b) -> [a] -> [b]
```

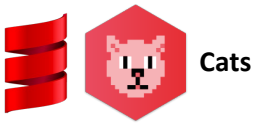
```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

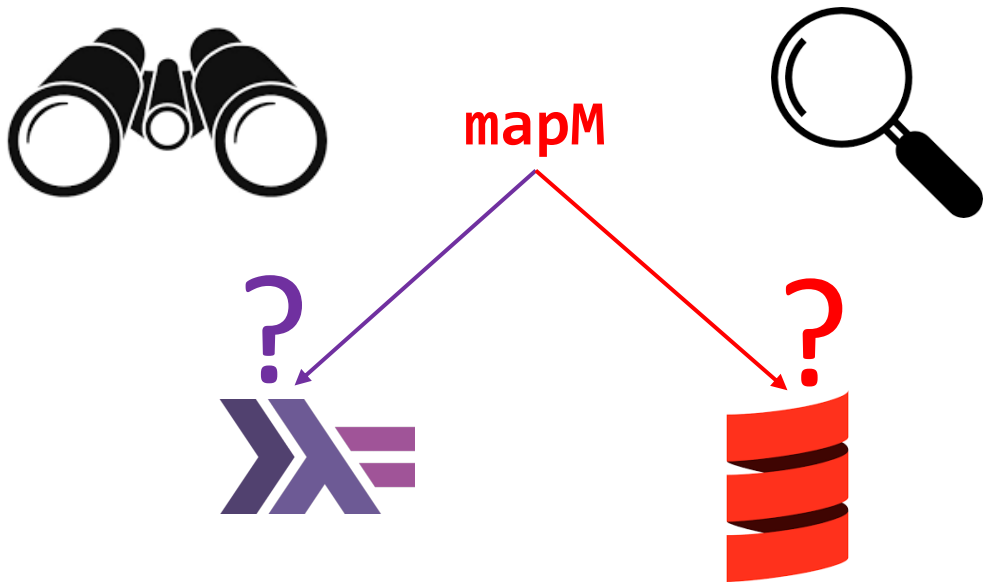
```haskell
map :: (a -> b) -> [a] -> [b]
```

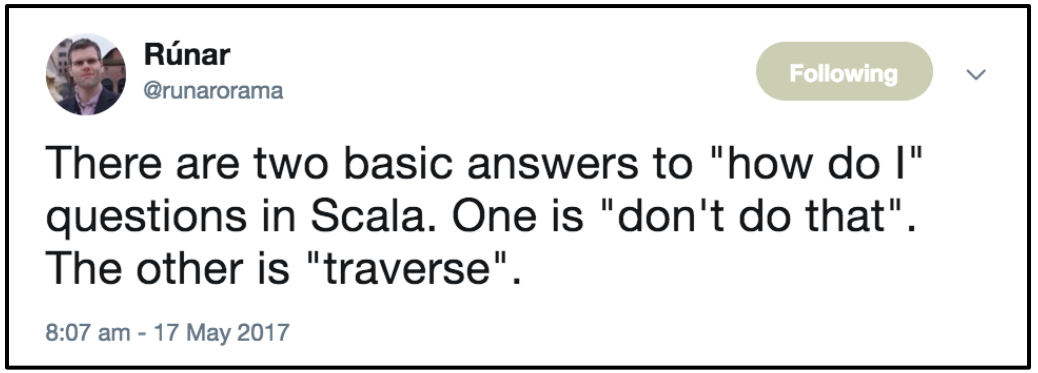
```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```



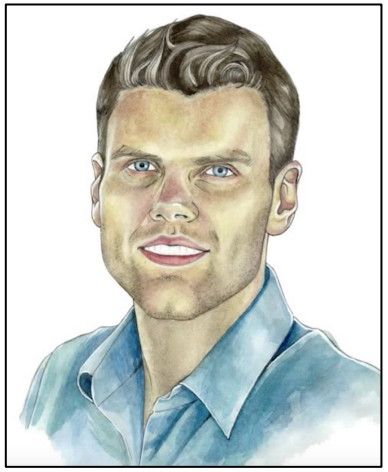
```scala
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```




mapM
?
?

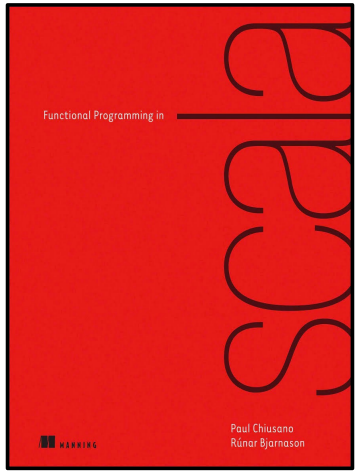


There turns out to be a startling number of operations that can be defined in the most general possible way in terms of **sequence** and/or **traverse**



Runar Bjarnason

@runarorama



FP in Scala

Paul Chiusano

@pchiusano

A quick refresher on the **traverse** function

While there *is* a **traverse** function in **Scala**'s standard library, it is specific to **Future**, and **IterableOnce**.

```
final def traverse[A, B, M <: (IterabelOnce)]
  (in: M[A])
  (fn: A => Future[B])
  (implicit bf: BuildFrom[M[A], B, M[B]],
            executor: ExecutionContext)
: Future[M[B]]
```

Asynchronously and **non-blockingly** transforms a `IterableOnce[A]` into a `Future[IterableOnce[B]]` using the provided function `A => Future[B]`.

This is useful for performing a **parallel map**. For example, to apply a function to all items of a list in parallel.

```
final def sequence[A, CC <: (IterabelOnce), To]
  (in: CC[Future[A]])
  (implicit bf: BuildFrom[CC[Future[A]], A, To],
            executor: ExecutionContext)
: Future[To]
```

Simple version of **Future.traverse**.

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)
```

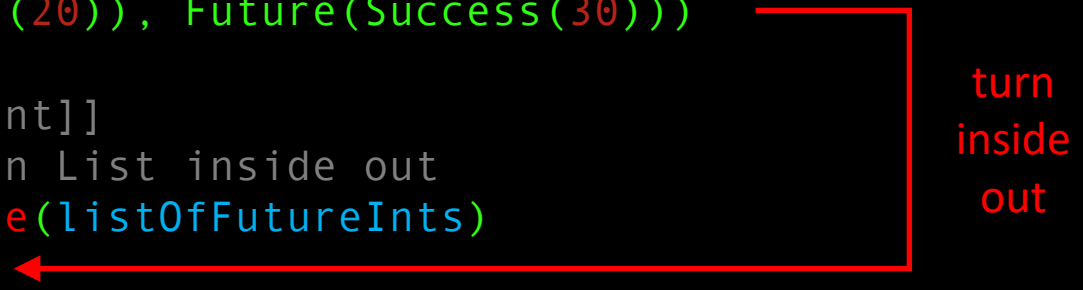
```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))
```



```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))

// first map List[Int] to List[Future[Int]]
// and then turn List[Future[Int]] into Future[List[Int]]
scala> val futureListOfInts = Future.traverse(listOfInts) { n => Future(n*10) }
```

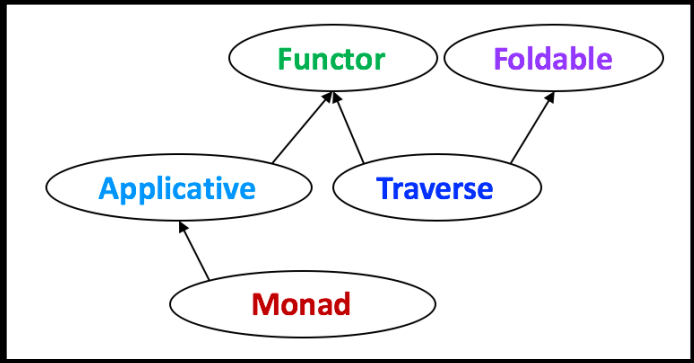
```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))

// first map List[Int] to List[Future[Int]]
// and then turn List[Future[Int]] into Future[List[Int]]
scala> val futureListOfInts = Future.traverse(listOfInts) { n => Future(n*10) }
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))
```

```
scala> import scala.concurrent.Future
     | import concurrent.ExecutionContext.Implicits.global

// list of integers
scala> val listOfInts = List(1,2,3)
val listOfInts: List[Int] = List(1, 2, 3)

// list of future integers
scala> val listOfFutureInts = listOfInts.map{ n => Future(n*10) }
val listOfFutureInts: List[Future[Int]] =
    List(Future(Success(10)), Future(Success(20)), Future(Success(30)))

// turn List[Future[Int]] into Future[List[Int]]
// i.e. it turns the nesting of Future within List inside out
scala> val futureListOfInts = Future.sequence(listOfFutureInts)
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))

// first map List[Int] to List[Future[Int]]
// and then turn List[Future[Int]] into Future[List[Int]]
scala> val futureListOfInts = Future.traverse(listOfInts) { n => Future(n*10) }
val futureListOfInts: Future[List[Int]] =
    Future(Success(List(10, 20, 30)))
```

same result

That was the quick refresher on the **traverse** function

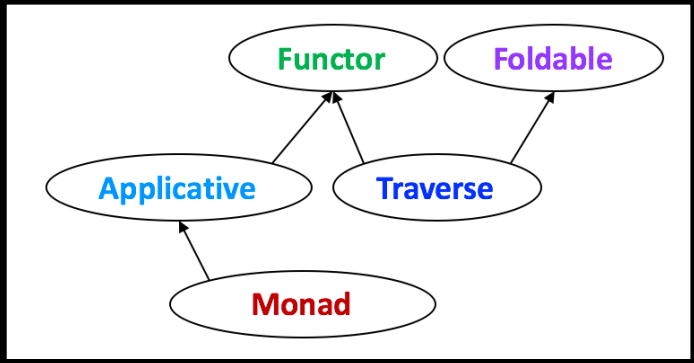
```haskell
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```



```
map :: (a -> b) -> [a] -> [b]
```

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

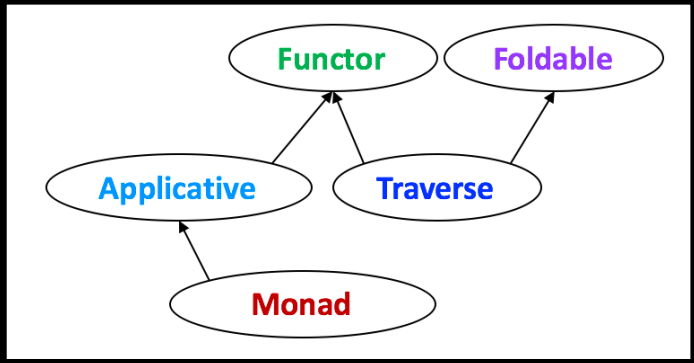
```
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```



```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

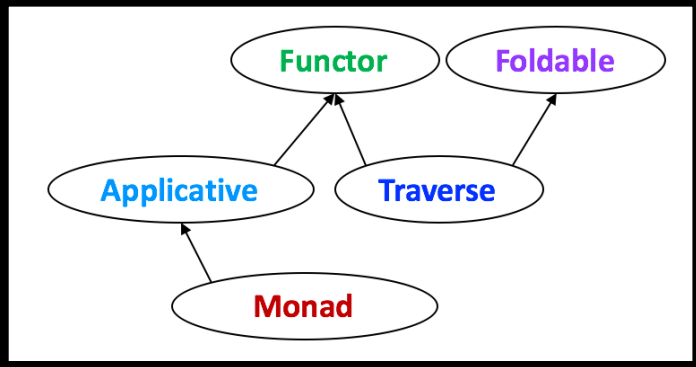
```haskell
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```



```
map :: (a -> b) -> [a] -> [b]
```

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

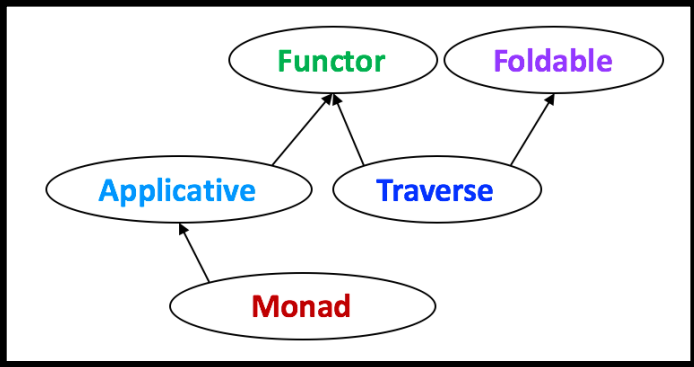
```
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

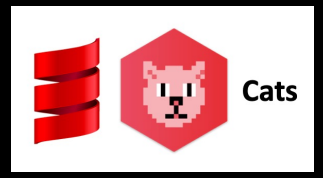


```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```haskell
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  …
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  mapM = traverse
```





Traverse, also known as Traversable.

Traversal over a structure with an effect.

Traversing with the cats.Id effect is equivalent to Functor#map. Traversing with the cats.data.Const effect where the first type parameter has a cats.Monoid instance is equivalent to Foldable#fold.

See: The Essence of the Iterator Pattern↗

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] with UnorderedTraverse[F] { self =>
```

Given a function which returns a G effect, thread this effect through the running of this function on all the values in F, returning an F[B] in a G context.

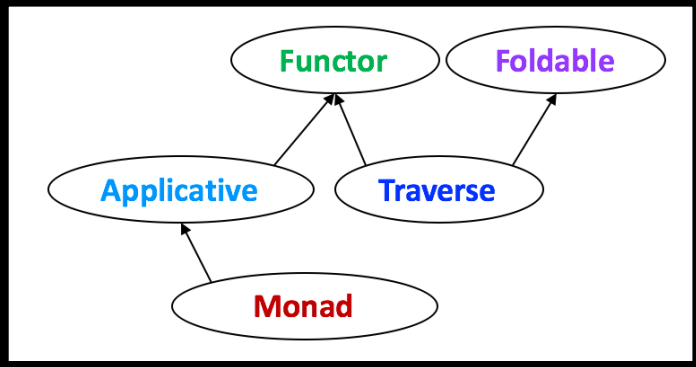
```
import cats.implicits._

def parseInt(s: String): Option[Int] = s.toIntOption

assert(List("1", "2", "3").traverse(parseInt) == Some(List(1, 2, 3)))

assert(List("1", "two", "3").traverse(parseInt) == None)
```

```
def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
```

```
map :: (a -> b) -> [a] -> [b]
```

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```
class (Functor t, Foldable t) => Traversable t where

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  …
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  mapM = traverse
```



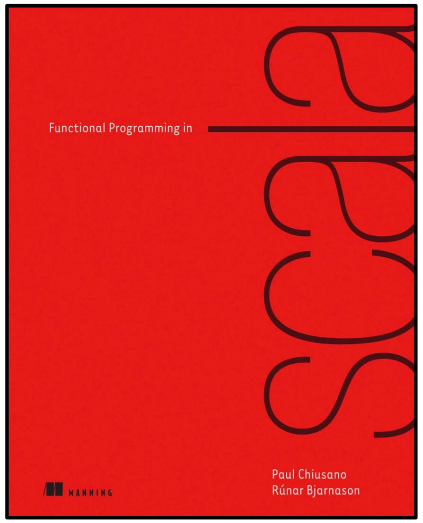
```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```scala
def mapM [A, B, M[_]: Monad](as: List[A])(f: A => M[B]): M[List[B]]
```

```scala
import cats.{Applicative, Monad}
import cats.syntax.traverse.*
import cats.Traverse

extension[A, B, M[_] : Monad] (as: List[A])
  def mapM(f: A => M[B]): M[List[B]] =
    as.traverse(f)
```

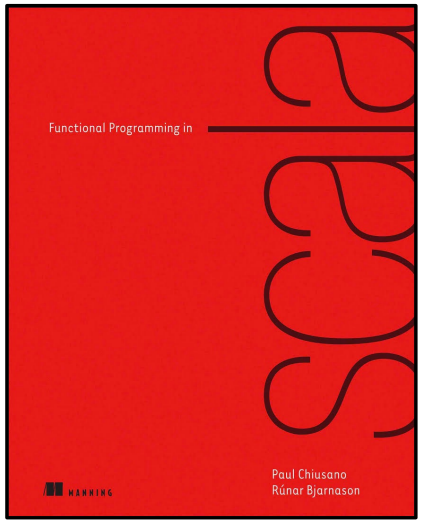
I am not really surprised
that **mapM** is just **traverse**.

As seen in the red book…



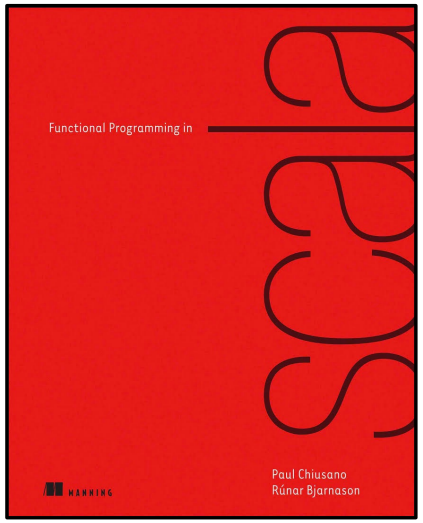
```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  …


  def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]]
  …
}
```

… **traverse** is so general that it can be used to define **map**…



```
G = Id Monad

trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  …
  def map[A,B](fa: F[A])(f: A => B): F[B] = traverse…

  def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]]
  …
}
```

… and **foldMap**



G = Id Monad

G = Applicative Monoid

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  …
  def map[A,B](fa: F[A])(f: A => B): F[B] = traverse…

  def foldMap[A,M](as: F[A])(f: A => M)(mb: Monoid[M]): M = traverse…

  def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]]
  …
}
```

A quick mention of the **Symmetry** that exists in the interrelation of **flatMap**/**foldMap**/**traverse** and **flatten**/**fold**/**sequence**

```scala
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]                    =  flatten(map(ma)(f))
```

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]          =  flatten(map(ma)(f))

def flatten[A](mma: F[F[A]]): F[A]                     =  flatMap(mma)(x ⇒ x)
```

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]          =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B         =     fold(map(fa)(f))


def flatten[A](mma: F[F[A]]): F[A]                     =  flatMap(mma)(x ⇒ x)
```

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]          =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B         =    fold(map(fa)(f))


def flatten[A](mma: F[F[A]]): F[A]                     =  flatMap(mma)(x ⇒ x)
def fold[A:Monoid](fa: F[A]): A                        =   foldMap(fa)(x ⇒ x)
```

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]                          =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B                         =     fold(map(fa)(f))
def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A ⇒ M[B]): M[F[B]] = sequence(map(fa)(f))

def flatten[A](mma: F[F[A]]): F[A]                                     =  flatMap(mma)(x ⇒ x)
def fold[A:Monoid](fa: F[A]): A                                        =   foldMap(fa)(x ⇒ x)
```

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]                        =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B                       =     fold(map(fa)(f))
def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A ⇒ M[B]): M[F[B]] = sequence(map(fa)(f))

def flatten[A](mma: F[F[A]]): F[A]                                   =  flatMap(mma)(x ⇒ x)
def fold[A:Monoid](fa: F[A]): A                                      =   foldMap(fa)(x ⇒ x)
def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]]              = traverse(fma)(x ⇒ x)
```

Symmetry in the interrelation of flatMap/foldMap/traverse and flatten/fold/sequence

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]                          =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B                         =     fold(map(fa)(f))
def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A ⇒ M[B]): M[F[B]] = sequence(map(fa)(f))

def flatten[A](mma: F[F[A]]): F[A]                                     =  flatMap(mma)(x ⇒ x)
def fold[A:Monoid](fa: F[A]): A                                        =   foldMap(fa)(x ⇒ x)
def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]]                = traverse(fma)(x ⇒ x)
```

Examples of **mapM** usage

Example	Monadic Context	How the result of **mapM** is affected
1		
2		
3		

Examples of **mapM** usage

Example	Monadic Context	How the result of **mapM** is affected
1	**Option**	There may or may not be a result.
2		
3		

Graham Hutton
@haskellhutt

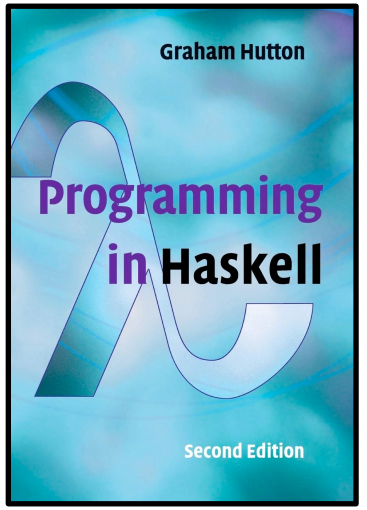
To illustrate how it might be used, consider a function that converts a digit character to its numeric value, provided that the character is indeed a digit:

```
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing
```

(The functions isDigit and digitToInt are provided in Data.Char.) Then applying mapM to the conv function gives a means of converting a string of digits into the corresponding list of numeric values, which succeeds if every character in the string is a digit, and fails otherwise:

```
> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a"
Nothing
```

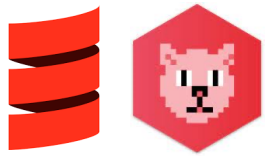
```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```




```
def mapM [A, B, M[_]: Monad](as: List[A])(f: A => M[B]): M[List[B]]
```

```
M = Option
```

```
def convert(c: Char): Option[Int] =
  Option.when(c.isDigit)(c.asDigit)

assert("12a4".toList.mapM(convert) == None)

assert("1234".toList.mapM(convert) == Some(List(1, 2, 3, 4)))
```



Let’s look at three examples of **mapM** usage:

Example	Monadic Context	How the result of **mapM** is affected
1	**Option**	There may or may not be a result.
2		
3		

Let’s look at three examples of **mapM** usage:

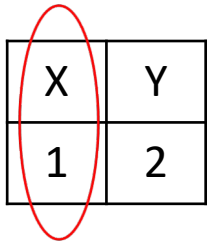
Example	Monadic Context	How the result of **mapM** is affected
1	**Option**	There may or may not be a result.
2	**List**	There may be zero, one, or more results.
3		

```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```

X	Y
1	2

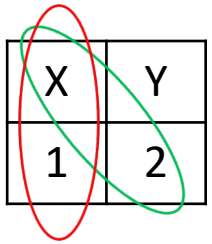
- X is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```



- X is combined first with 1, and then with 2, and the results are paired

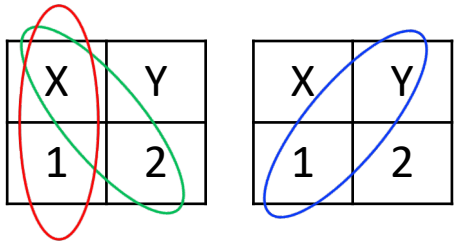
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"))
```

- X is combined first with 1, and then with 2, and the results are paired

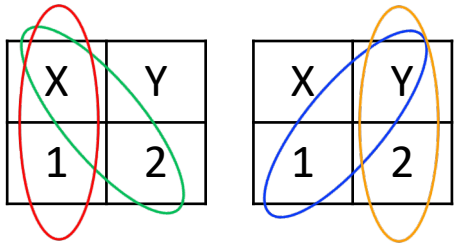
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"))
```

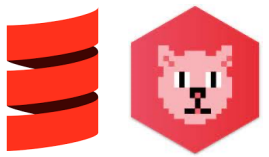
- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```

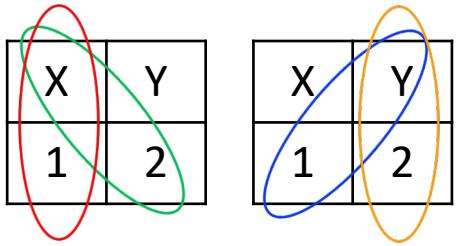



```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

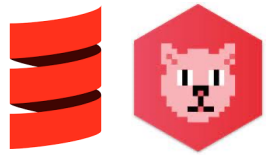
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

X	Y
1	2

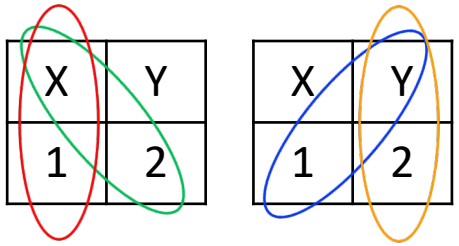
- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

`M = List`



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

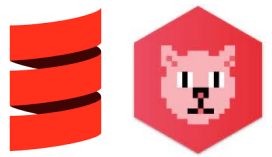
- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

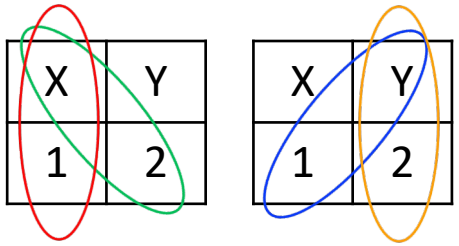
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

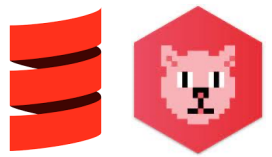
- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

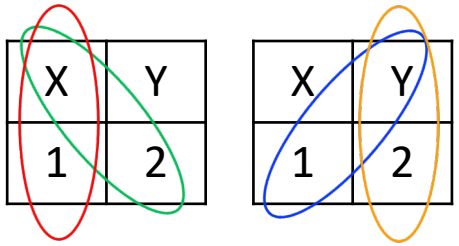
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2"))
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

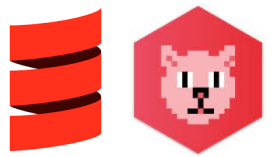
- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

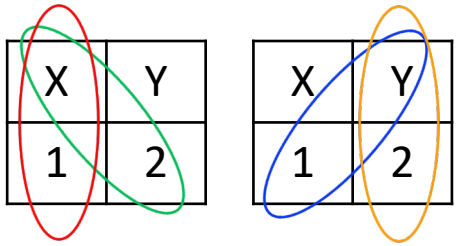
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2"))
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

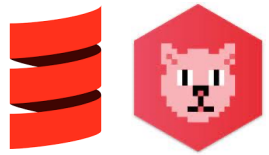
- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

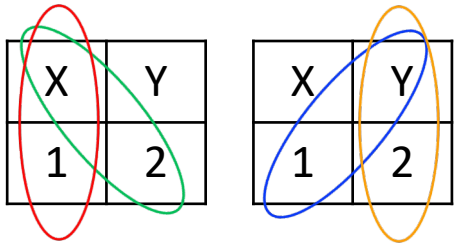
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2"))
      bs <- mapM(as)(f) // ??
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

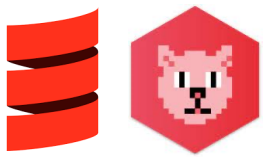
→ recursive call for `List("Y")`

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

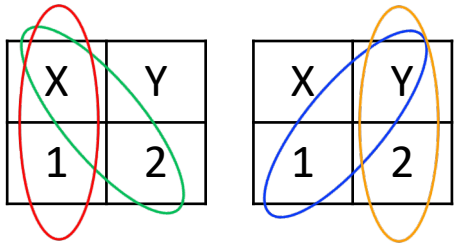
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

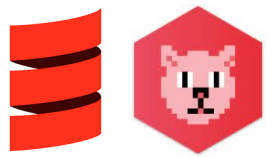
→ recursive call for `List("Y")`

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

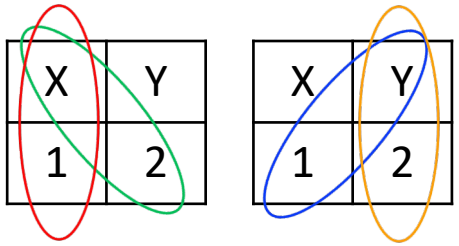
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

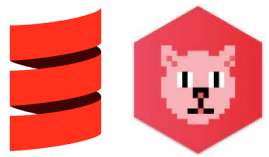
→ recursive call for `List("Y")`

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

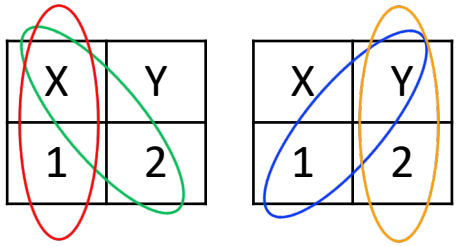
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

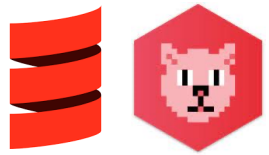
→ recursive call for `List("Y")`

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

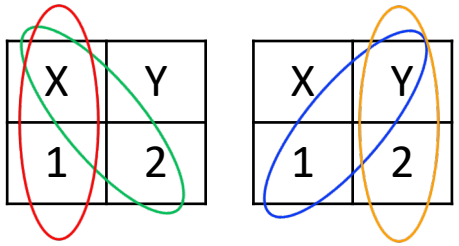
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // ??
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

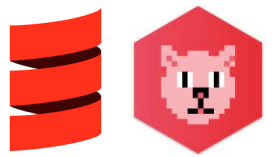
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

X	Y
1	2

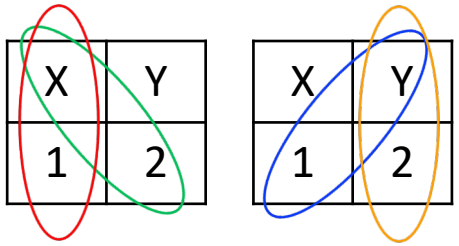
- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

→ recursive call for `List("Y")`
→ recursive call for `List()`

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure // List(List())
  case a::as =>
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

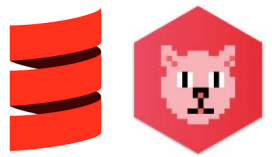
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

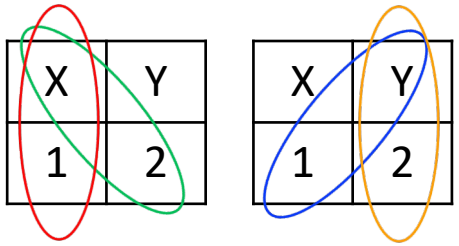
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

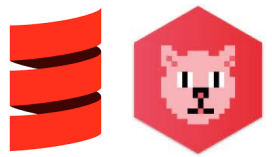
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

X	Y
1	2

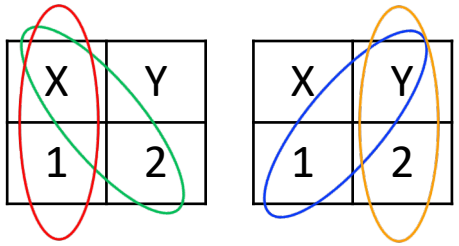
- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

→ recursive call for `List("Y")`
→ ~~recursive call for `List()`~~

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

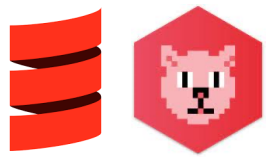
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

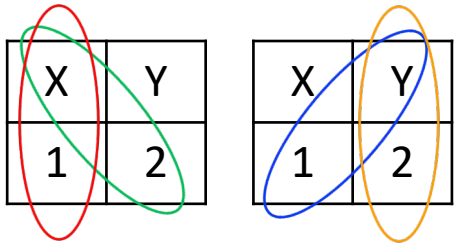
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~

`List("Y1")`

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs // "Y1"::List()
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

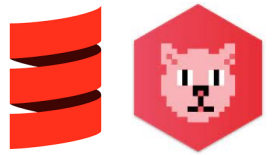
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

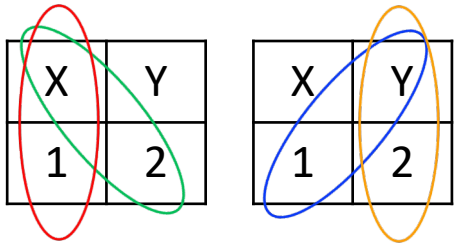
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~

```
List("Y1")
```

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // ??
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

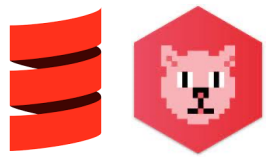
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

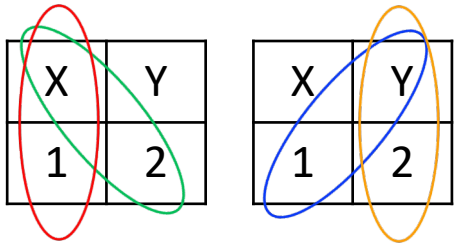
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~
→ recursive call for `List()`

```
List("Y1")
```

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure // List(List())
  case a::as =>
    for
      b <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

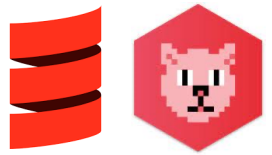
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

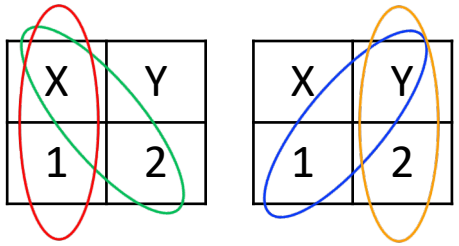
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~
~~→ recursive call for `List()`~~

```
List("Y1")
```

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

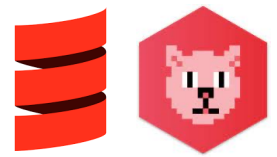
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

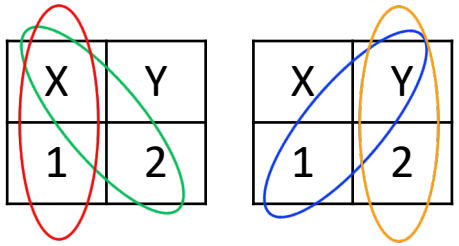
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~
~~→ recursive call for `List()`~~

```
List("Y1")
```

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

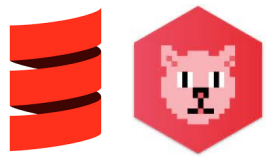
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

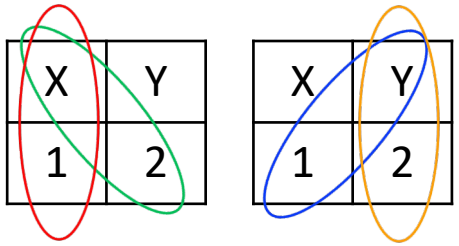
→ recursive call for `List("Y")`
~~→ recursive call for `List()`~~
~~→ recursive call for `List()`~~

`List("Y1") List("Y2")`

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "Y"::List()
    for
      b <- f(a) // List("Y1","Y2")
      bs <- mapM(as)(f) // List(List())
    yield b::bs // "Y2"::List()
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

X	Y
1	2

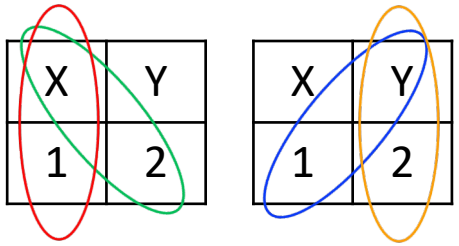
- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

~~→recursive call for List("Y")~~
~~→recursive call for List()~~
~~→recursive call for List()~~

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```

```
M = List
```

```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

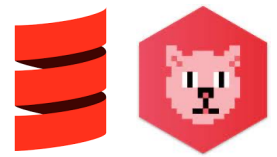
X	Y
1	2

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

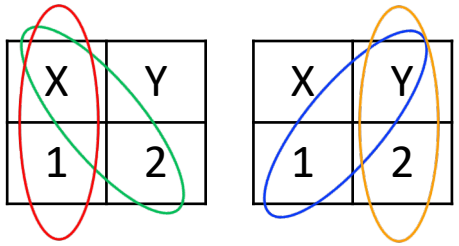
~~→recursive call for List("Y")~~
~~→recursive call for List()~~
~~→recursive call for List()~~

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```

```
M = List
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```

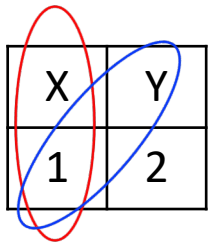



```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

~~→recursive call for List("Y")~~
~~→recursive call for List()~~
~~→recursive call for List()~~

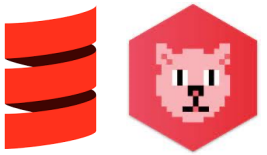
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



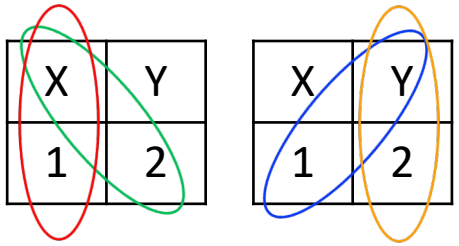

```
List(List("X1","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs // List("X1", "Y1")
```



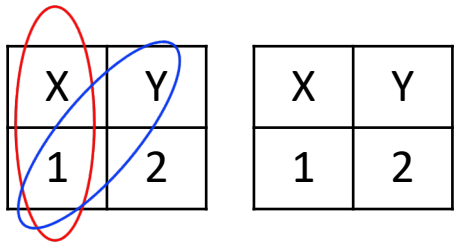
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

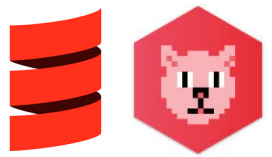
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



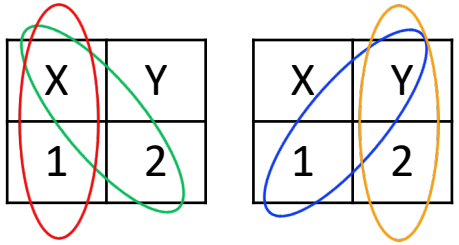

```
List(List("X1","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```



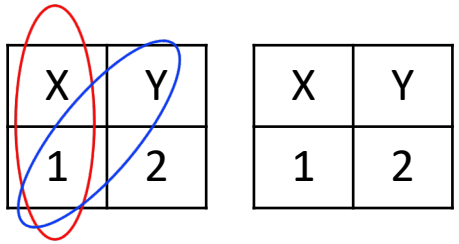
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

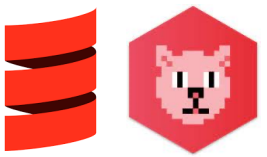
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



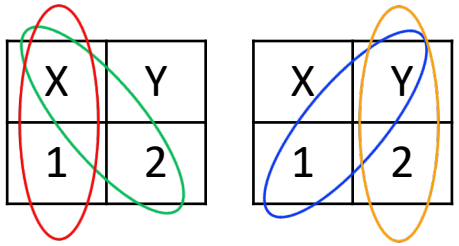

```
List(List("X1","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```



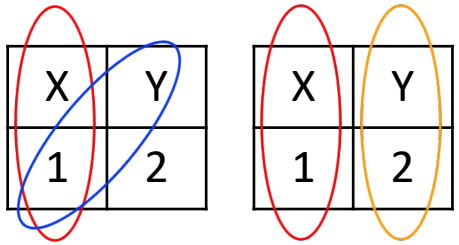
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

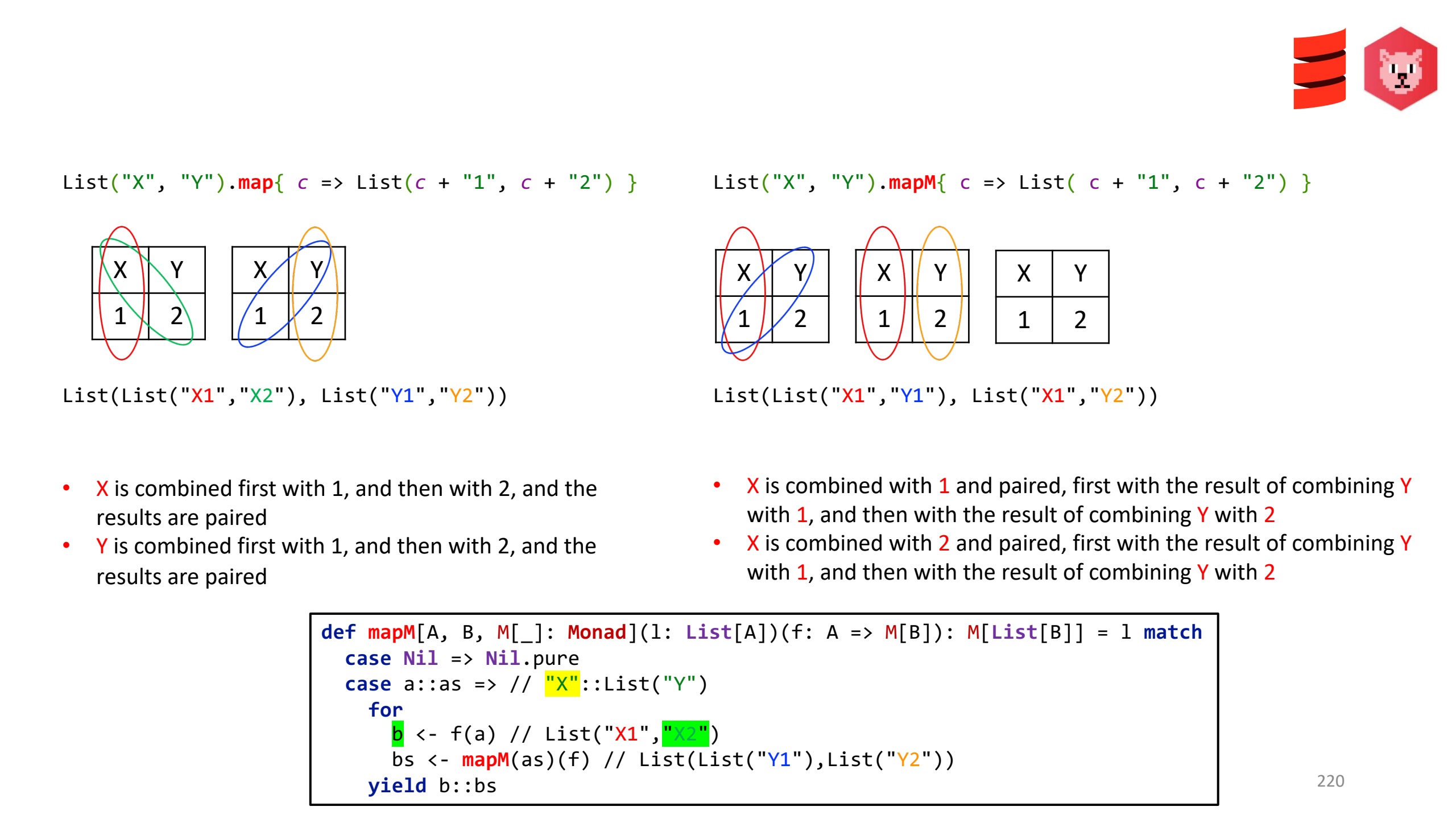
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```




```
List(List("X1","Y1"), List("X1","Y2"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs // List("X1","Y2")
```



```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```



```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

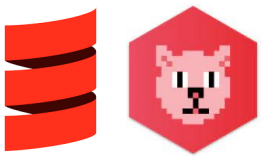
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



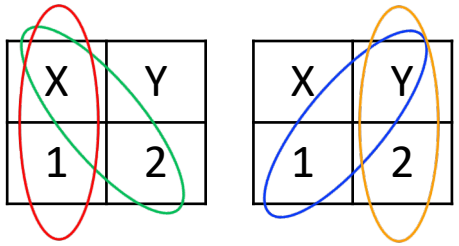
```
List(List("X1","Y1"), List("X1","Y2"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```



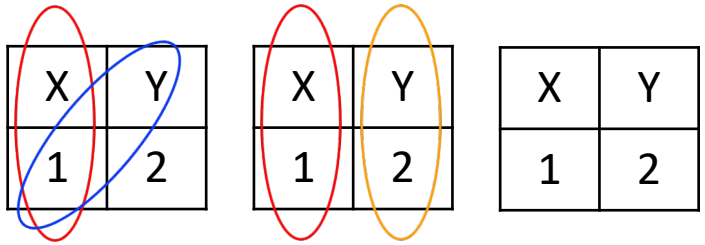
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

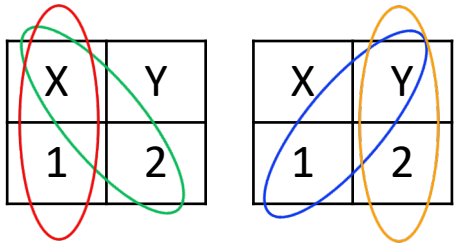



```
List(List("X1","Y1"), List("X1","Y2"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```

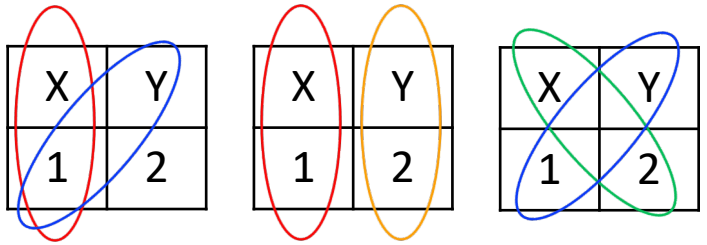
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

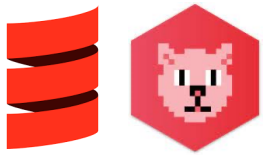
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



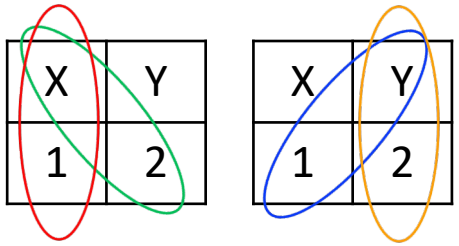

```
List(List("X1","Y1"), List("X1","Y2"),
     List("X2","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs // List("X2","Y1")
```



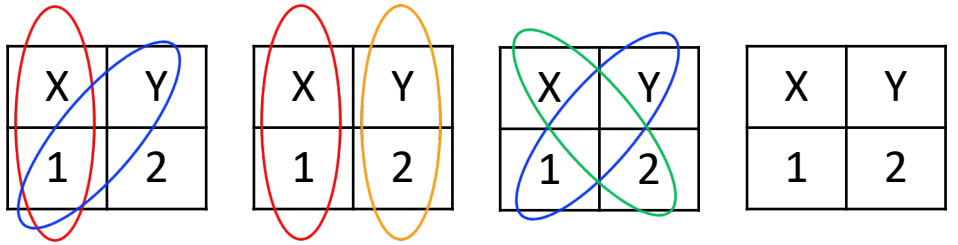
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

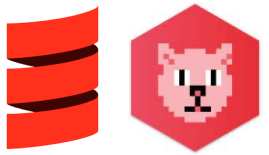
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



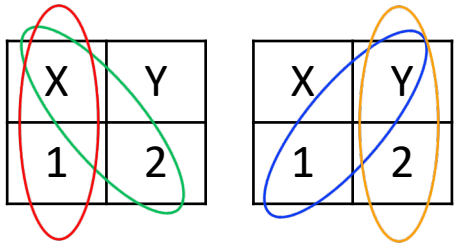

```
List(List("X1","Y1"), List("X1","Y2"),
     List("X2","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```



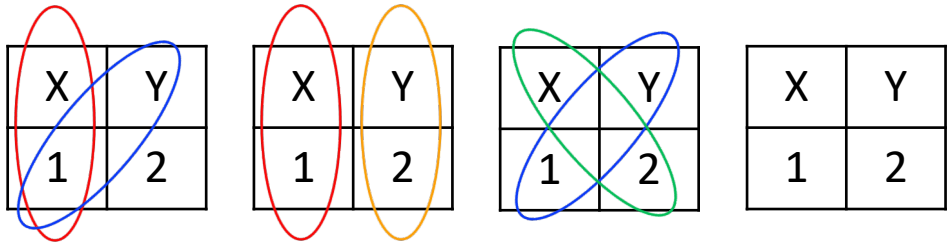
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```

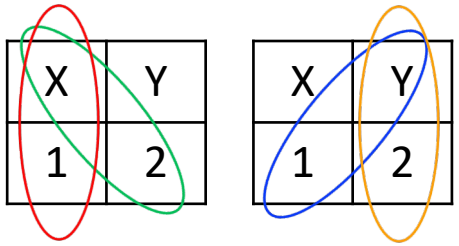



```
List(List("X1","Y1"), List("X1","Y2"),
     List("X2","Y1"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs
```

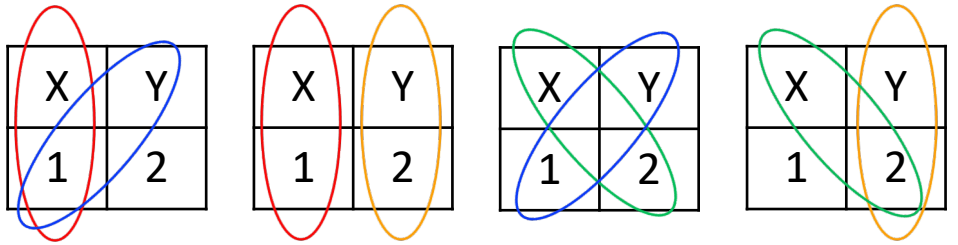
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

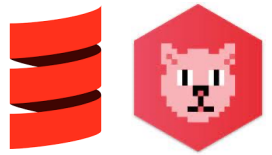
```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```



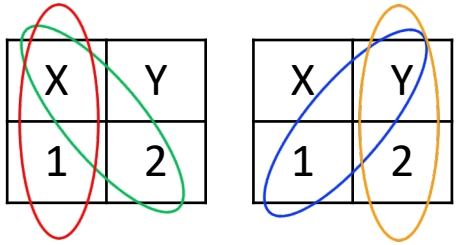

```
List(List("X1","Y1"), List("X1","Y2"),
     List("X2","Y1"), List("X2","Y2"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs // List("X2","Y2")
```



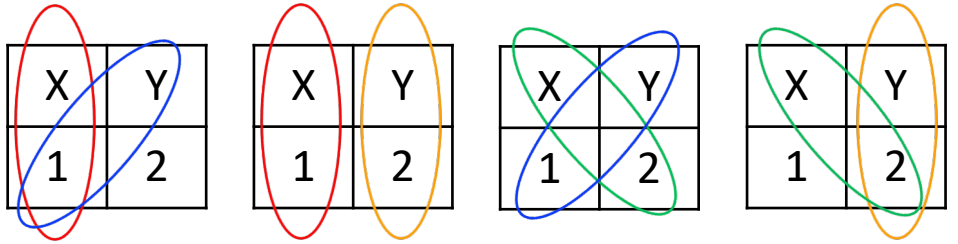
```
List("X", "Y").map{ c => List(c + "1", c + "2") }
```




```
List(List("X1","X2"), List("Y1","Y2"))
```

- X is combined first with 1, and then with 2, and the results are paired
- Y is combined first with 1, and then with 2, and the results are paired

```
List("X", "Y").mapM{ c => List( c + "1", c + "2") }
```




```
List(List("X1","Y1"), List("X1","Y2"),
     List("X2","Y1"), List("X2","Y2"))
```

- X is combined with 1 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2
- X is combined with 2 and paired, first with the result of combining Y with 1, and then with the result of combining Y with 2

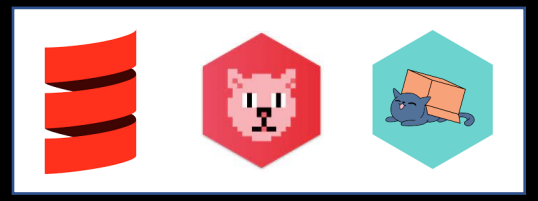
```
def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as => // "X"::List("Y")
    for
      b <- f(a) // List("X1","X2")
      bs <- mapM(as)(f) // List(List("Y1"),List("Y2"))
    yield b::bs // List("X2","Y2")
```

Examples of **mapM** usage

Example	Monadic Context	How the result of **mapM** is affected
1	**Option**	There may or may not be a result.
2	**List**	There may be zero, one, or more results.
3		

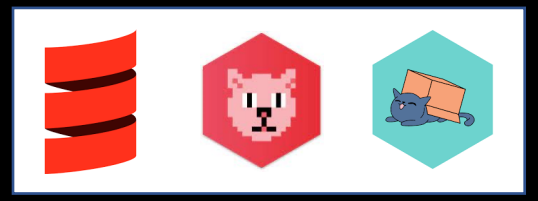
Examples of **mapM** usage

Example	Monadic Context	How the result of **mapM** is affected
1	**Option**	There may or may not be a result.
2	**List**	There may be zero, one, or more results.
3	**IO**	The result suspends any side effects.

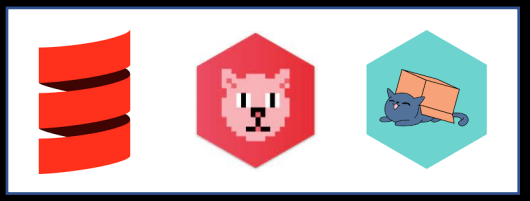


```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))
```

side effects suspended in pure value



```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")
```



```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }
```

side effects
suspended in
pure value

```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }

val numbers: List[Int] = getNumbers.unsafeRunSync()
```

side effects occur

```
Enter a number: 3
Enter another: 2
```

```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }

val numbers: List[Int] = getNumbers.unsafeRunSync()

println(s"The two numbers add up to ${numbers.sum}")
```

```
Enter a number: 3
Enter another: 2
The two numbers add up to 5
```

```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }

val numbers: List[Int] = getNumbers.unsafeRunSync()

println(s"The two numbers add up to ${numbers.sum}")
```

if `getNumber` cannot parse an input, it returns 0

```
Enter a number: 3
Enter another: a
The two numbers add up to 3
```

```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }

val numbers: List[Int] = getNumbers.unsafeRunSync()

println(s"The two numbers add up to ${numbers.sum}")
```

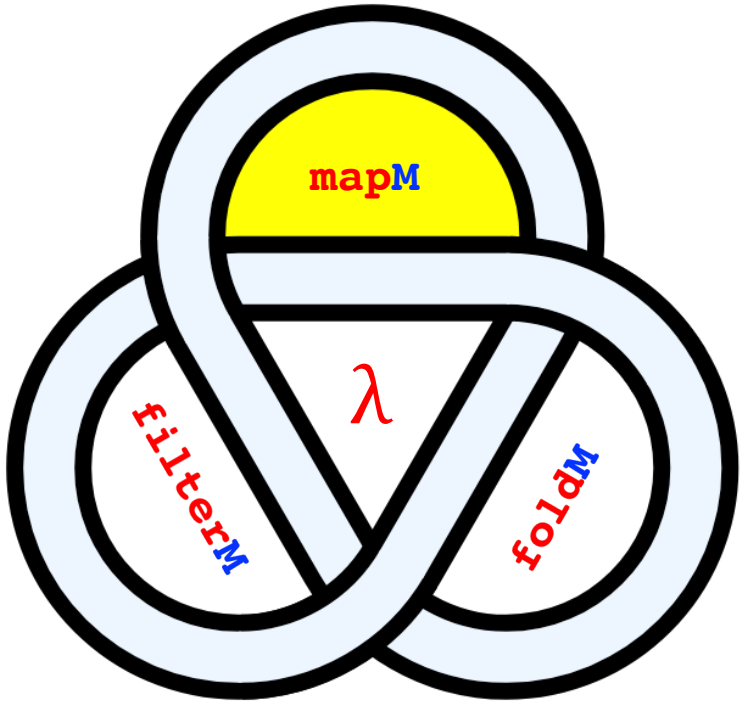
If `IO.readLine` hangs waiting for input, so does the whole program

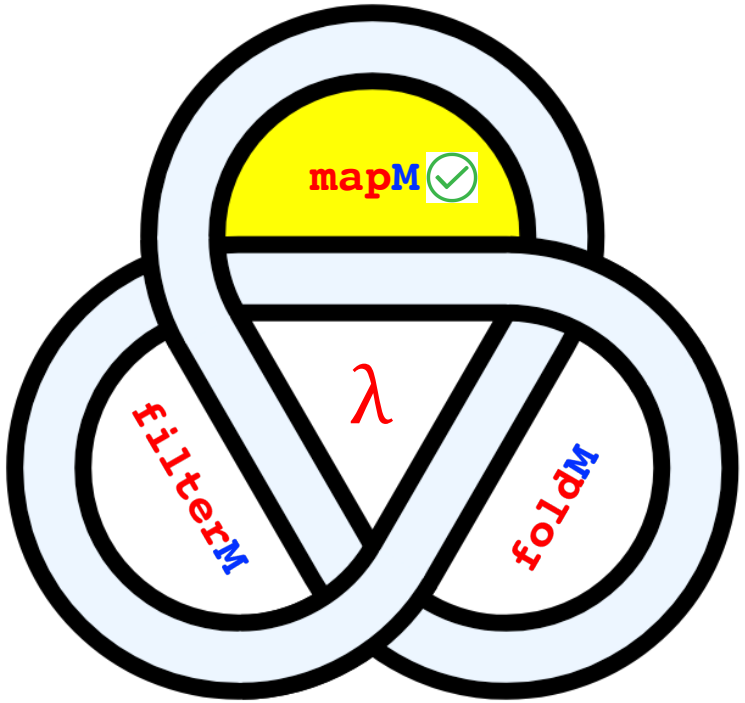
```
Enter a number: ⌛...
```

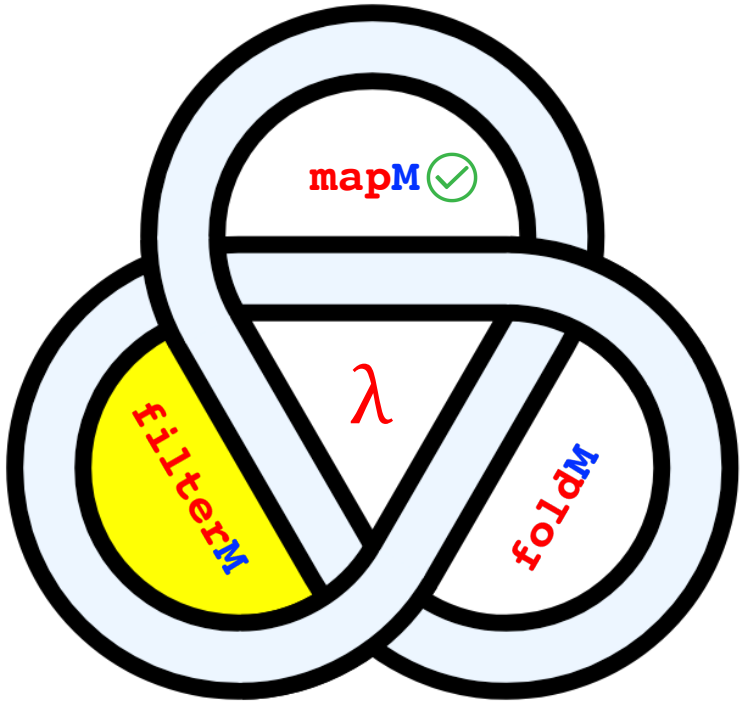
```scala
import cats.effect.IO
import cats.effect.unsafe.implicits.global

import scala.io.StdIn.readLine

def getNumber(msg: String): IO[Int] =
  (IO.print(msg) *> IO.readLine).map(_.toIntOption.getOrElse(0))

val messages = List("Enter a number: ", "Enter another: ")

val getNumbers: IO[List[Int]] =
  messages.mapM{ getNumber }

val numbers: List[Int] = getNumbers.unsafeRunSync()

println(s"The two numbers add up to ${numbers.sum}")
```

If `IO.readLine` throws an exception, so does the whole program

```
Enter a number: ⚡⚡⚡
```


mapM
λ
filterM
foldM


mapM
λ
filterM
foldM

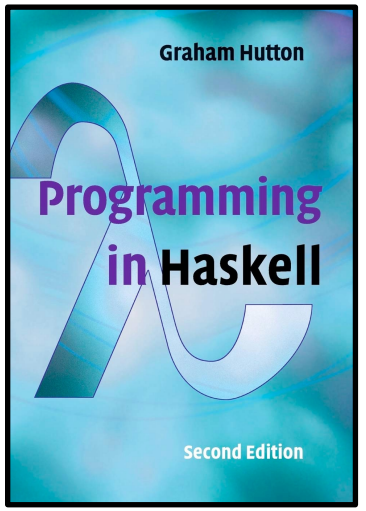

mapM
λ
filterM
foldM

Graham Hutton

@haskellhutt

A **monadic** version of the **filter** function on lists is defined by generalizing its type and definition in a similar manner to **mapM**:

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```

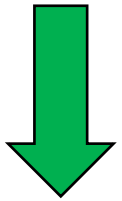



```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

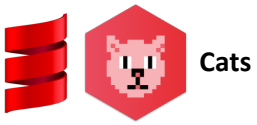
```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```

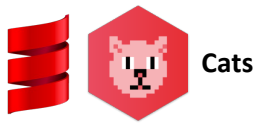
```haskell
filter :: (a -> Bool) -> [a] -> [a]
```

```haskell
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```



```scala
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(f: A => M[Boolean]): M[List[B]] = l
match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```



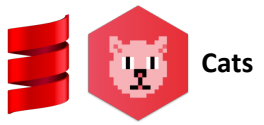


mapM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b  <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

filterM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA   <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```

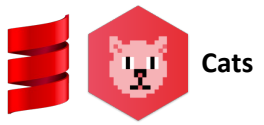


mapM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b  <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

filterM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA   <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```



mapM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b  <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

filterM

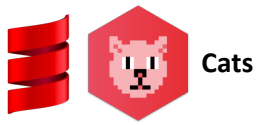
```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA   <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```

mapM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b  <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

filterM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA   <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```



mapM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def mapM[A, B, M[_]: Monad](l: List[A])(f: A => M[B]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      b  <- f(a)
      bs <- mapM(as)(f)
    yield b::bs
```

filterM

```
import cats.Monad
import cats.syntax.functor.*      // map
import cats.syntax.flatMap.*      // flatMap
import cats.syntax.applicative.*  // pure

def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      retainingA   <- p(a)
      retainedAs <- filterM(as)(p)
    yield if retainingA then a::retainedAs else retainedAs
```

`filterM`

?

https://typelevel.org/cats/api/cats/TraverseFilter.html

cats

TraverseFilter **Companion object TraverseFilter**

```
trait  TraverseFilter[F[_]] extends FunctorFilter[F]
```

TraverseFilter, also known as Witherable, represents list-like structures that can essentially have a traverse and a filter applied as a single combined operation (traverseFilter).

```
def filterA[G[_], A](fa: F[A])(f: (A) ⇒ G[Boolean])(implicit G: Applicative[G]): G[F[A]]
```

Filter values inside a G context.

This is a generalized version of Haskell's filterM. This StackOverflow question about filterM may be helpful in understanding how it behaves.

```haskell
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```scala
def filterM [A, M[_]: Monad](as: List[A])(f: A => M[Boolean]): M[List[A]]
```

```scala
import cats.Monad
import cats.syntax.traverseFilter.*

extension[A, M[_]: Monad](as: List[A])
  def filterM(f: A => M[Boolean]): M[List[A]] =
    as.filterA(f)
```

Examples of **filterM** usage

Example	Monadic Context
1	
2	

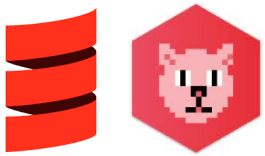
Examples of **filterM** usage

Example	Monadic Context
1	**Option**
2	

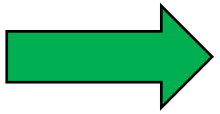
```haskell
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing

> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a"
Nothing
```

conv function returns `Int` in an `Option` monadic context

```scala
def convert(c: Char): Option[Int] =
  Option.when(c.isDigit)(c.asDigit)

assert(
  "1234".toList.mapM(convert)
  ==
  Some(List(1, 2, 3, 4))
)

assert(
  "1234a".toList.mapM(convert)
  ==
  None
)
```



```haskell
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing

> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a"
Nothing
```

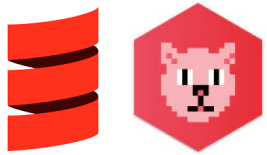


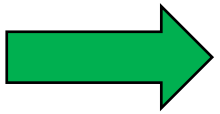
```haskell
maybeIsEven :: Char -> Maybe Bool
maybeIsEven c | isDigit c = Just (even (digitToInt c))
              | otherwise = Nothing

> filterM maybeIsEven "1234"
Just [2,4]

> mapM maybeIsEven "123a"
Nothing
```

conv function returns **Int** in an **Option** monadic context

maybeIsEven function returns a **Boolean** in an **Option** monadic context

```scala
def conv(c: Char): Option[Int] =
  Option.when(c.isDigit)(c.asDigit)

assert(
  "12a4".toList.mapM(conv)
  ==
  None
)

assert(
  "1234".toList.mapM(conv)
  ==
  Some(List(1, 2, 3, 4))
)
```





```scala
def maybeIsEven(c: Char): Option[Boolean] =
  Option.when(c.isDigit)(c.asDigit % 2 == 0)

assert(
  "12a4".toList.filterM(maybeIsEven)
  ==
  None
)

assert(
  "1234".toList.filterM(maybeIsEven)
  ==
  Some(List('2','4'))
)
```

Examples of **filterM** usage

Example	Monadic Context
1	**Option**
2	

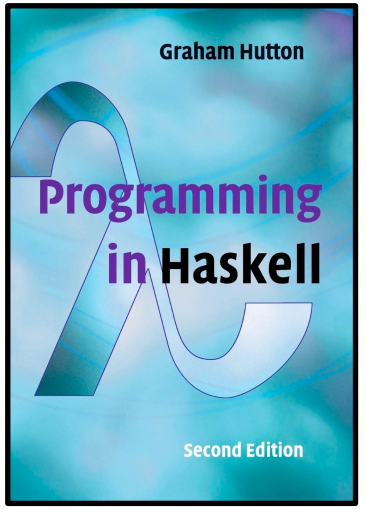
Examples of **filterM** usage

Example	Monadic Context
1	**Option**
2	**List**

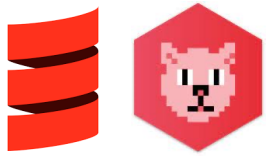
Graham Hutton
@haskellhutt

For example, in the case of the **list monad**, **using filterM provides a particularly concise means of computing the powerset of a list**, which is given by **all possible ways of including or excluding each element of the list**:

```
> filterM (\x -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```




```
assert(
  List(1, 2, 3).filterM(_ => List(true, false))
  ==
  List(List(1, 2, 3),
       List(1, 2),
       List(1, 3),
       List(1),
       List(2, 3),
       List(2),
       List(3),
       List())
)
```



the anonymous function returns a `Boolean` in a `List` monadic context

```scala
List().filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a)
      filteredAs <- filterM(as)(p)
    yield if keepingA then a::filteredAs else filteredAs
```

```
[]
```

```
List().filterM(_ => List(true, false))
```

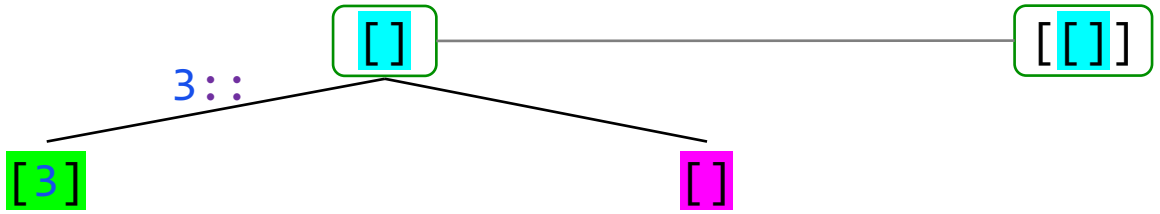
```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a)
      filteredAs <- filterM(as)(p)
    yield if keepingA then a::filteredAs else filteredAs
```

```scala
List(3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List())
    yield if keepingA then a::filteredAs else filteredAs
```

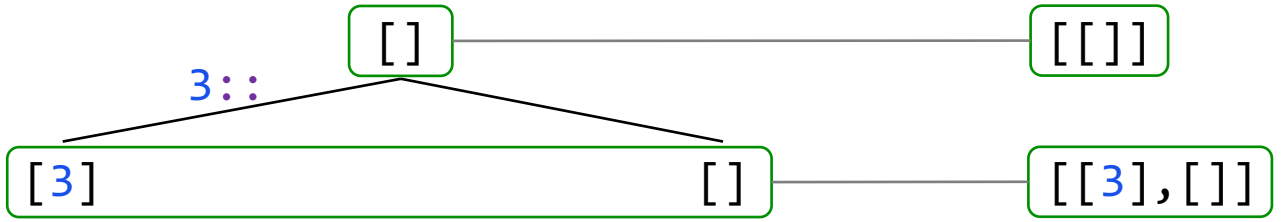
```scala
List(3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List())
    yield if keepingA then a::filteredAs else filteredAs
```



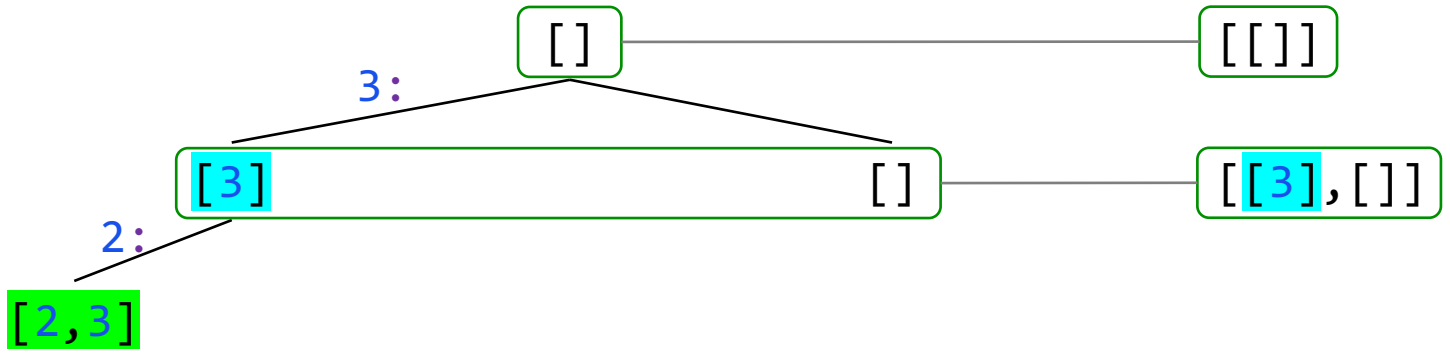
```scala
List(3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List())
    yield if keepingA then a::filteredAs else filteredAs
```



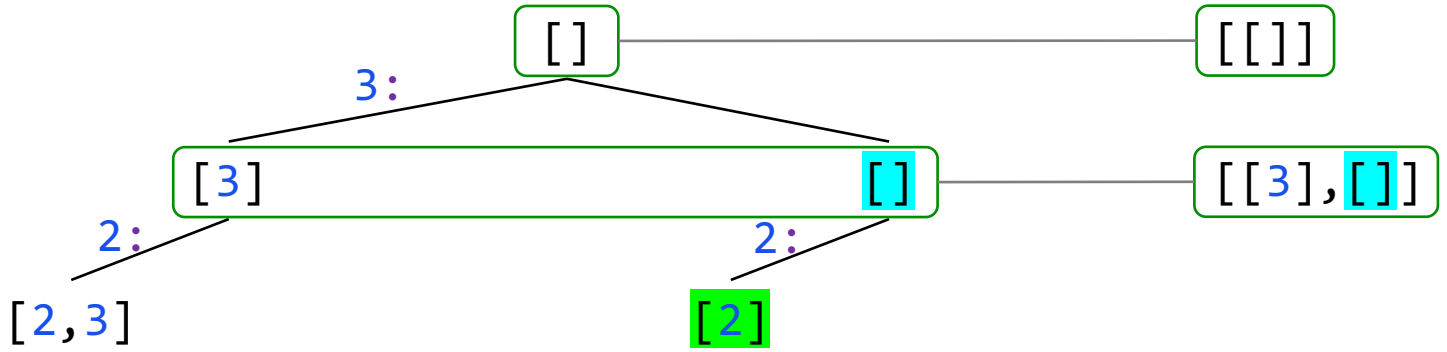
```
List(2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



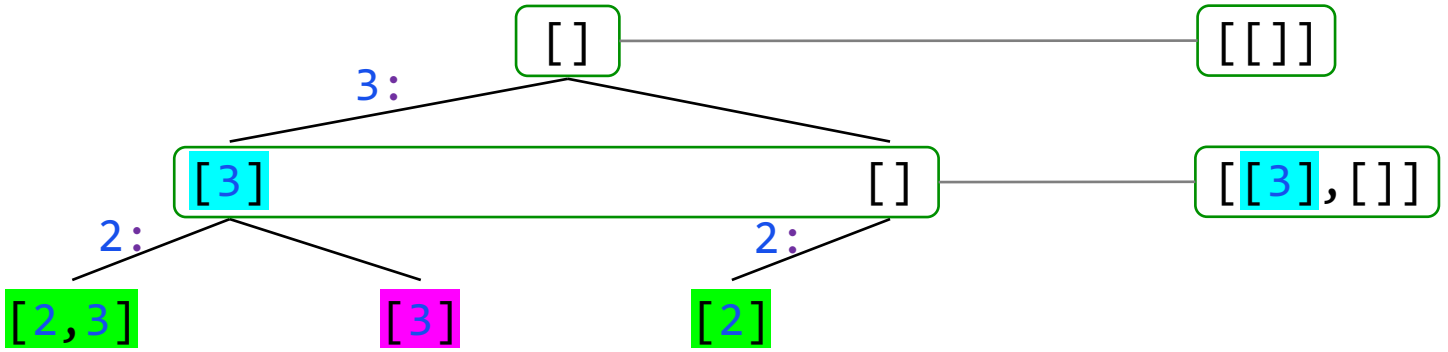
```
List(2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



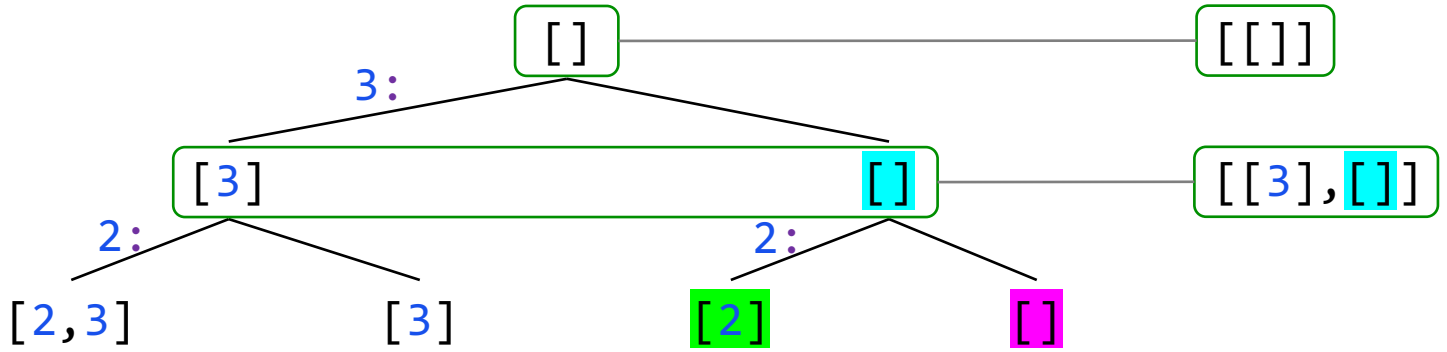
```
List(2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



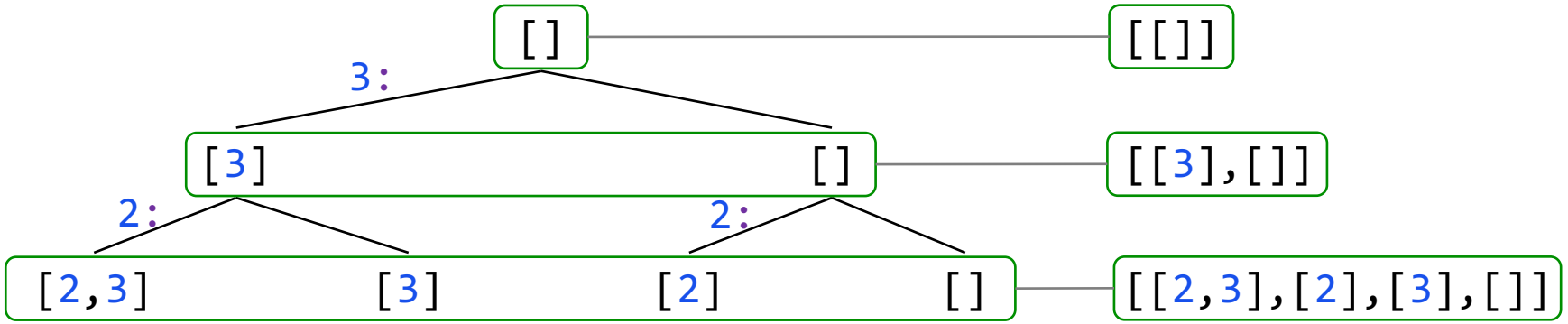
```
List(2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



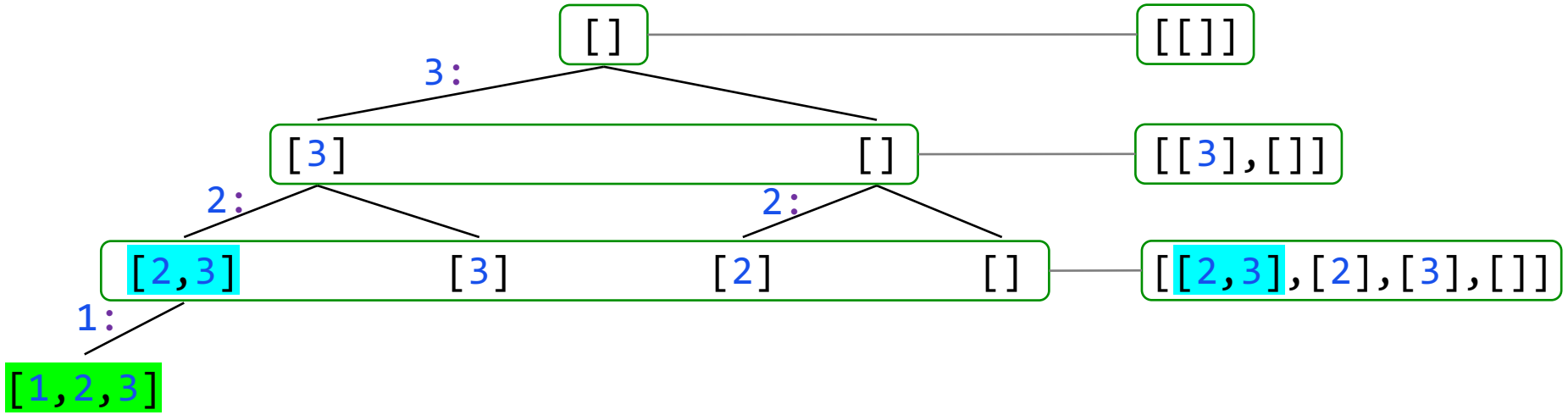
```
List(2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



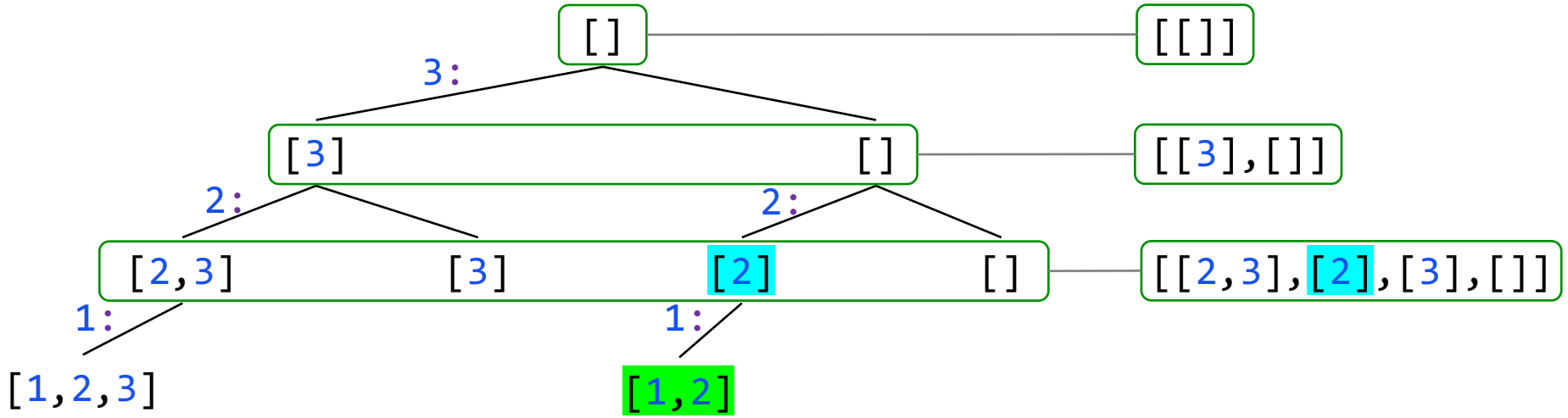
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



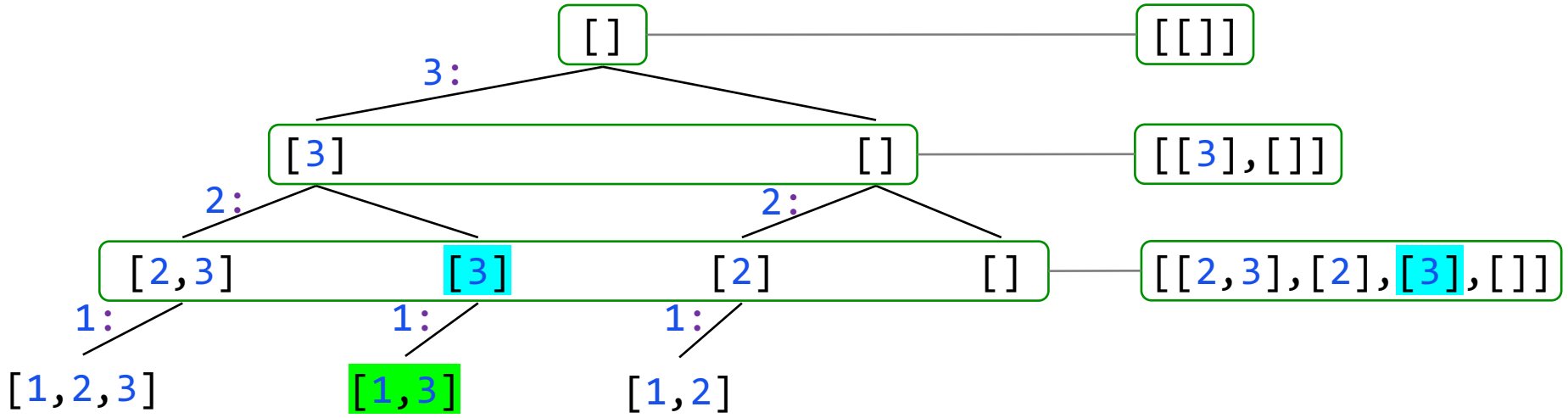
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



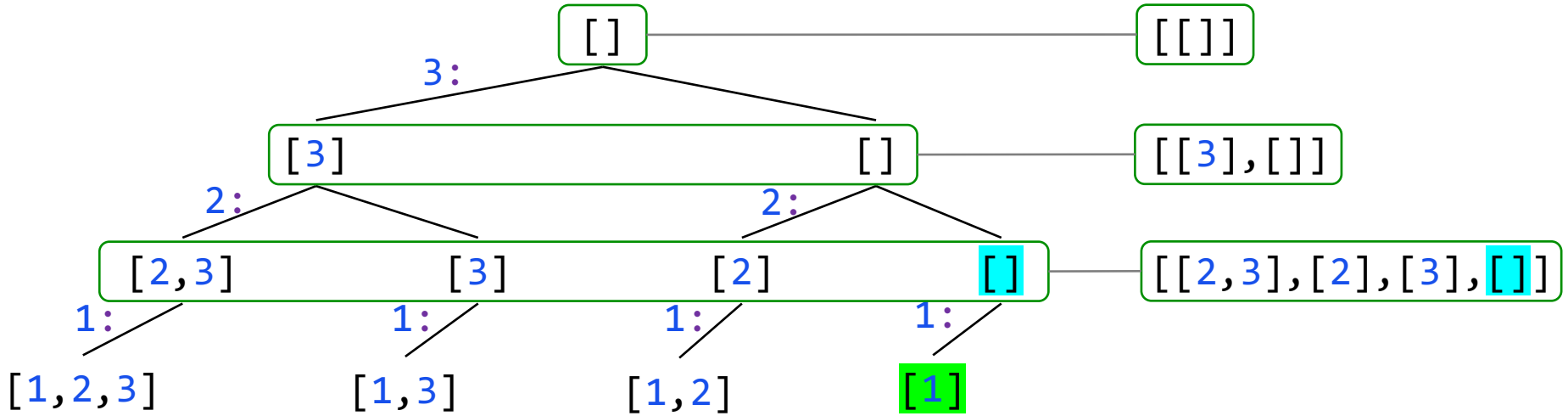
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



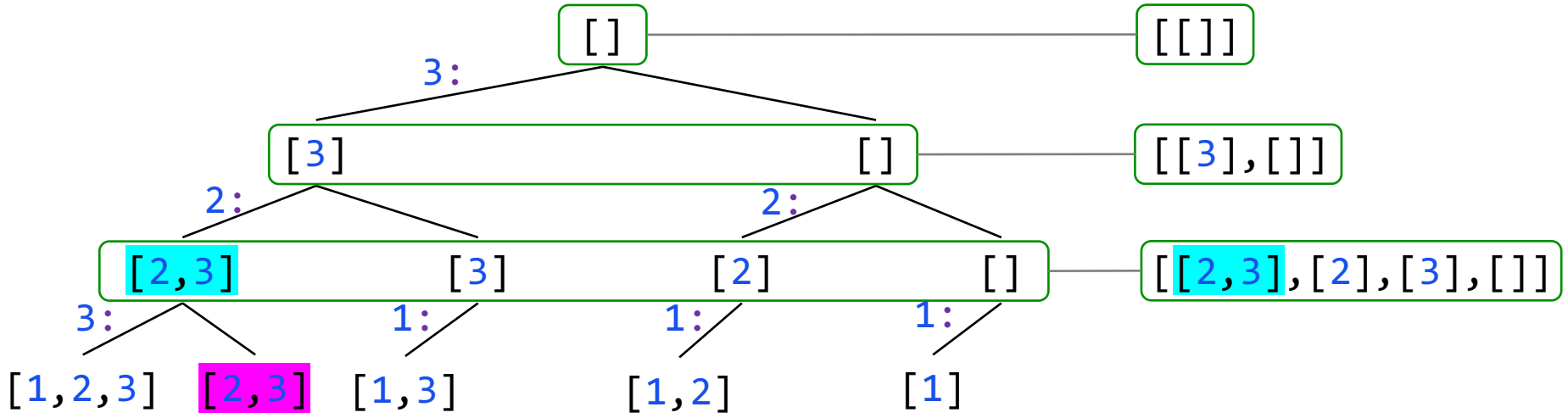
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



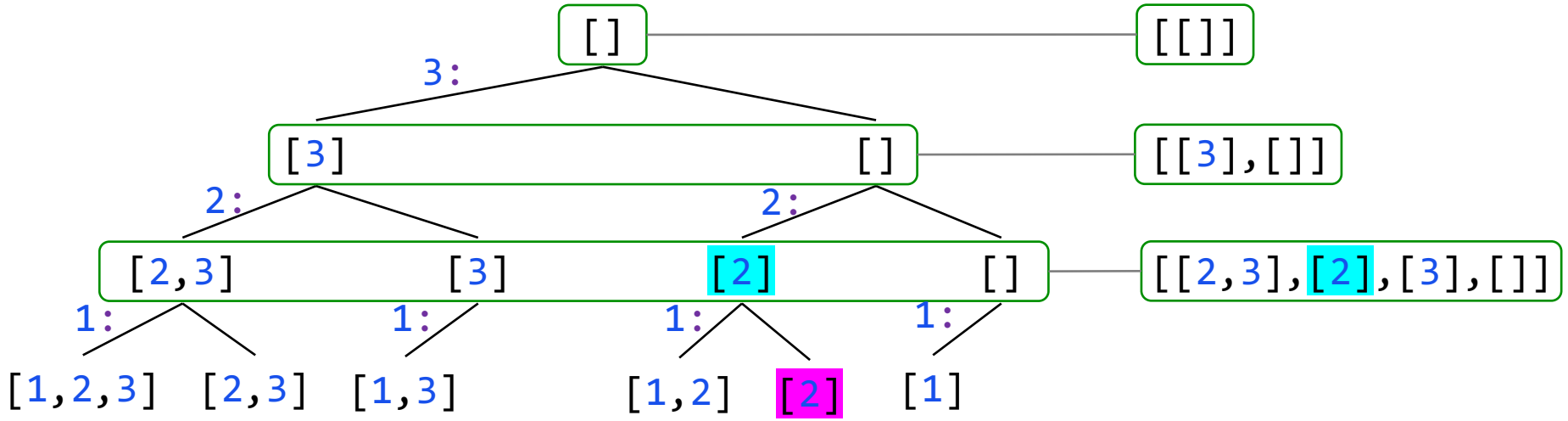
```
List(1, 2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



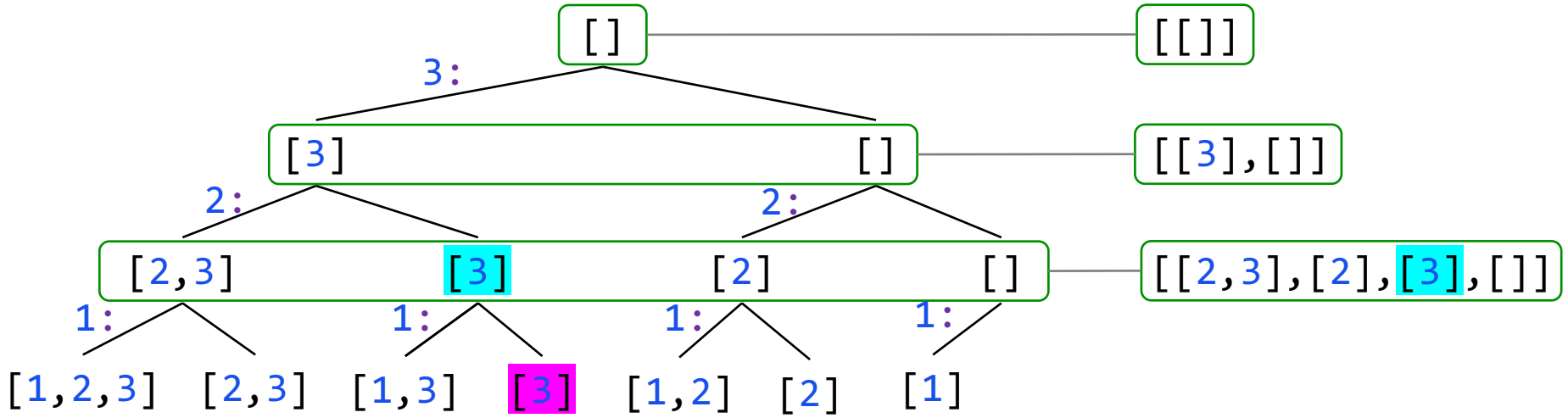
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



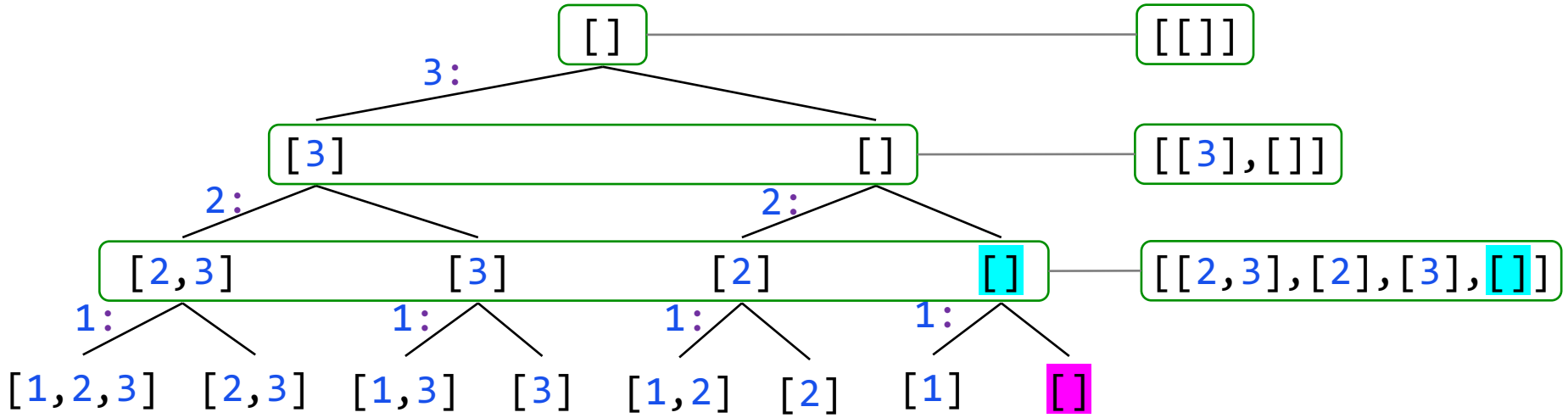
```
List(1, 2, 3).filterM(_ => List(true, false))
```

```
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```



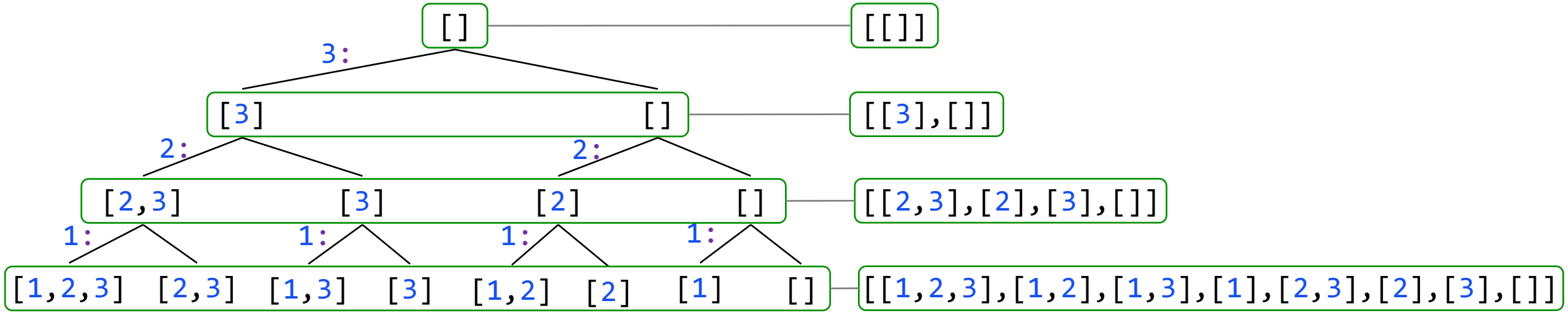
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

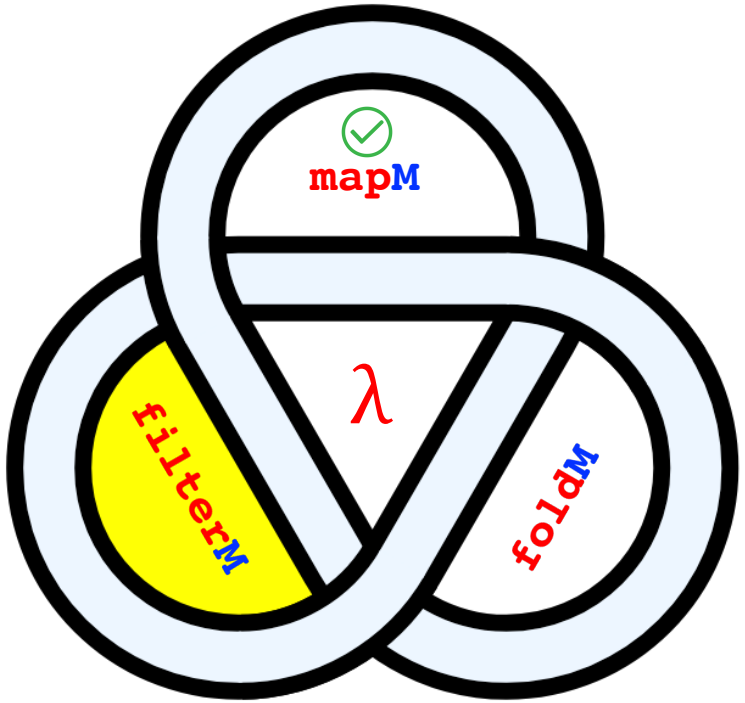
```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a) // List(true, false)
      filteredAs <- filterM(as)(p) // List(List(2,3),List(2),List(3),List())
    yield if keepingA then a::filteredAs else filteredAs
```

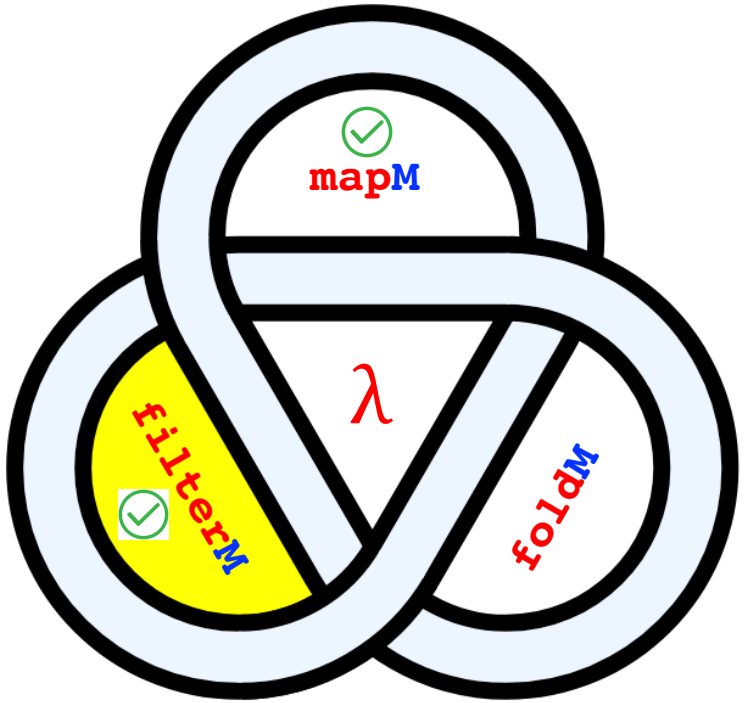


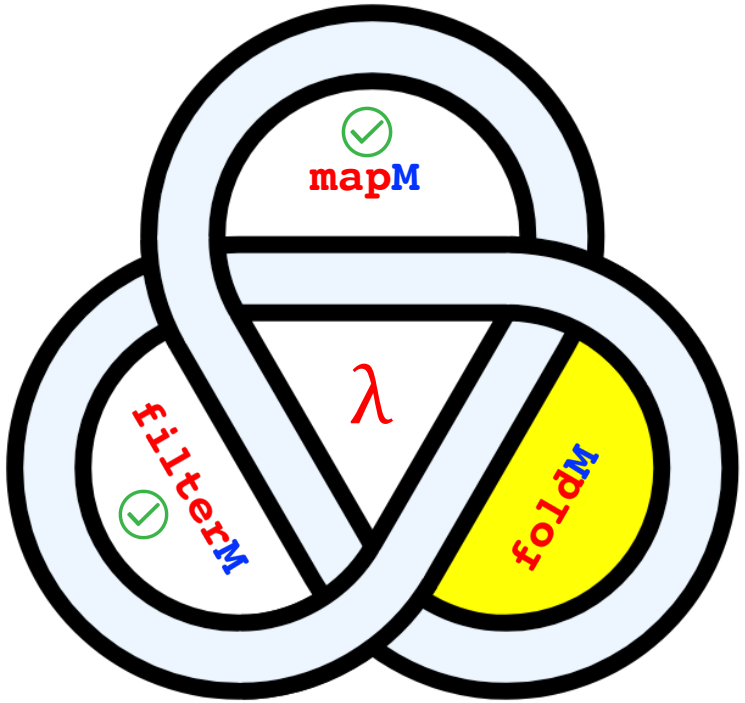
```scala
List(1, 2, 3).filterM(_ => List(true, false))
```

```scala
def filterM[A, M[_]: Monad](l: List[A])(p: A => M[Boolean]): M[List[B]] = l match
  case Nil => Nil.pure
  case a::as =>
    for
      keepingA   <- p(a)
      filteredAs <- filterM(as)(p)
    yield if keepingA then a::filteredAs else filteredAs
```




mapM
λ
filterM
foldM


mapM
λ
filterM
foldM


mapM
λ
filterM
foldM

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

Applies a **binary operator** to a **start value** and all elements of this collection, going **left** to **right**.

Miran Lipovača

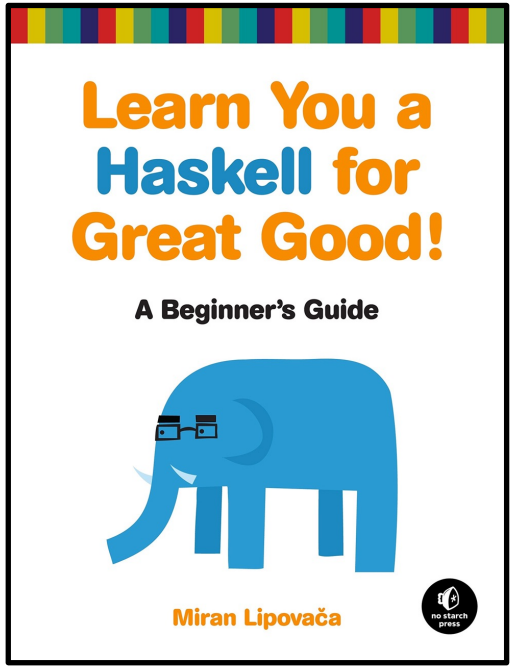
Let's sum a list of numbers with a **fold**

```
> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

…

Now what if we wanted to **sum a list of numbers** but with the **added condition** that **if any number is greater than 9 in the list, the whole thing fails?**

…



Miran Lipovača

foldM

The monadic counterpart to foldl is foldM.

…

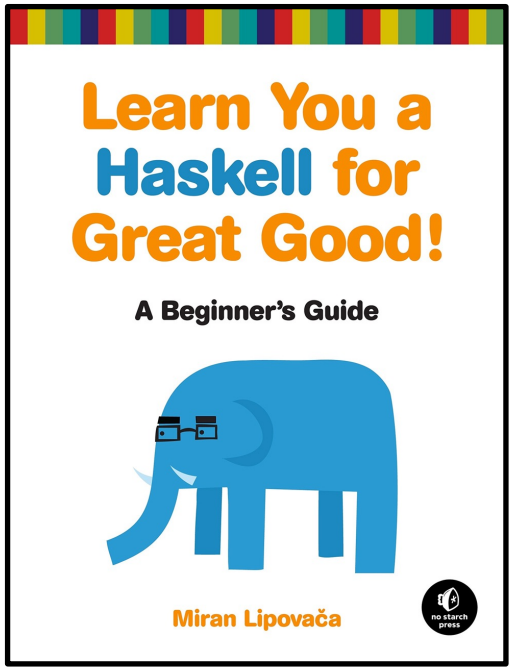
The type of foldl is this:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Whereas foldM has the following type:

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

The value that the binary function returns is monadic and so the result of the whole fold is monadic as well



Miran Lipovača

foldM

The monadic counterpart to foldl is foldM.

…

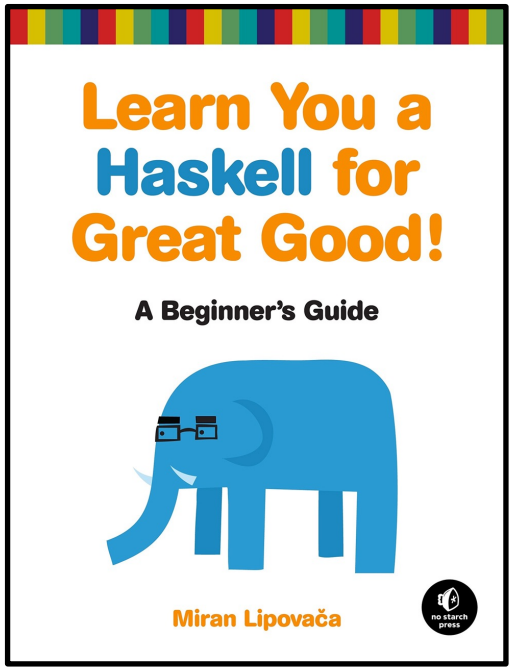
The type of foldl is this:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Whereas foldM has the following type:

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

The value that the binary function returns is monadic and so the result of the whole fold is monadic as well

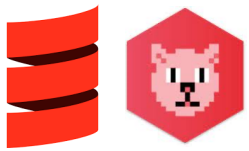


https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Monad.html

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b        # Source
```

> The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the `folded function' are not commutative.

https://typelevel.org/cats/api/cats/Foldable$.html



cats

Foldable

Companion object Foldable

```
trait  Foldable[F[_]] extends UnorderedFoldable[F] with FoldableNFunctions[F]
```

Data structures that can be folded to a summary value.

```
def foldM[G[_], A, B](fa: F[A], z: B)(f: (B, A) ⇒ G[B])(implicit G: Monad[G]): G[B]
```

Perform a stack-safe monadic left fold from the source context F into the target monad G.

Examples of **foldM** usage:

Example	Monadic Context	How the result of **foldM** is affected
1		
2		

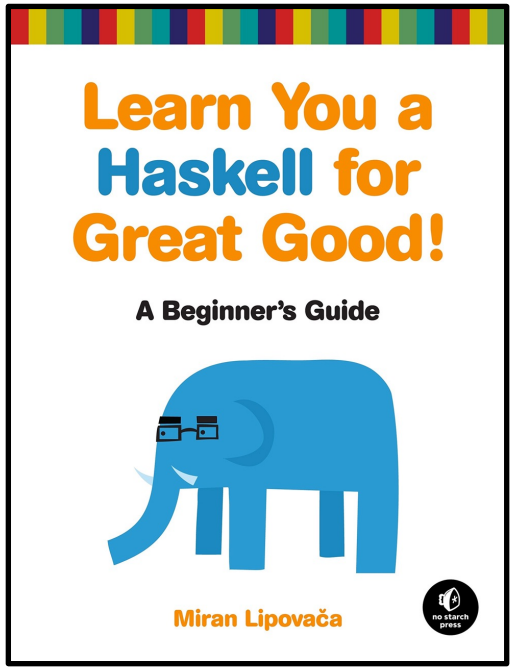
Examples of **foldM** usage:

Example	Monadic Context	How the result of **foldM** is affected
1	**Option**	There may or may not be a result.
2		

Miran Lipovača

```haskell
binSmalls  :: Int -> Int -> Maybe Int
binSmalls  acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)

ghci> foldM binSmalls 0 [2,8,3,1]
Just 14

ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

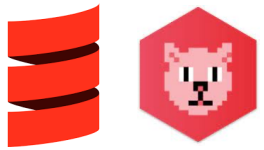
Excellent! Because one number in the list was greater than **9**, the whole thing resulted in a **Nothing**.




```scala
def foldM[G[_], B](z: B)(f: (B, A) => G[B])(implicit G: Monad[G]): G[B]
```

```scala
def binSmalls(acc: Int, x: Int): Option[Int] = x match
  case x if x > 9 => None
  case otherwise  => Some(acc + x)
```

```scala
import cats.syntax.foldable._

assert( List(2,11,3,1).foldM(0)(binSmalls) == None )
assert(  List(2,8,3,1).foldM(0)(binSmalls) == Some(14) )
```



That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

Things get a bit harder to understand when the **binary function** returns a **list of values**.

That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

Things get a bit harder to understand when the **binary function** returns a **list of values**.

The way we are going to solve the **N-Queens puzzle** using **foldM** is by passing the latter a **binary function** returning a **list of values**.

That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

Things get a bit harder to understand when the **binary function** returns a **list of values**.

The way we are going to solve the **N-Queens puzzle** using **foldM** is by passing the latter a **binary function** returning a **list of values**.

So in upcoming slides we are going to look at a number of examples that do just that.

That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

Things get a bit harder to understand when the **binary function** returns a **list of values**.

The way we are going to solve the **N-Queens puzzle** using **foldM** is by passing the latter a **binary function** returning a **list of values**.

So in upcoming slides we are going to look at a number of examples that do just that.

This is to strengthen our understanding of the **foldM** function.

That example of using **foldM** with a **binary function** that returns an **optional value** is useful.

Things get a bit harder to understand when the **binary function** returns a **list of values**.

The way we are going to solve the **N-Queens puzzle** using **foldM** is by passing the latter a **binary function** returning a **list of values**.

So in upcoming slides we are going to look at a number of examples that do just that.

This is to strengthen our understanding of the **foldM** function.

Before we do that though, let's take another look at the definition of **foldM**.

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b   # Source
```

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the `folded function' are not commutative.

```
foldM f a1 [x1, x2, ..., xm]

==

do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1 -- access the value that is wrapped in a monadic context
  a3 <- f a2 x2 -- access the value that is wrapped in a monadic context
  ...
  f am xm - return the value leaving it wrapped in a monadic context
```

If right-to-left evaluation is required, the input list should be reversed.

Note: `foldM` is the same as `foldlM`

https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Monad.html

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

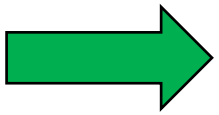
we can look at the above as follows

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  return f am xm
```

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

we can look at the above as follows

```
foldM f a1 [x1, x2, ..., xm]
==
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  return f am xm
```



which translates to

```
List(x1, x2, ..., xm).foldM(a1)(f)
==
(for
   a2 <- f(a1,x1)
   a3 <- f(a2,x2)
   ...
 yield f(am,xm)).flatten
```

Examples of **foldM** usage:

Example	Monadic Context	How the result of **foldM** is affected
1	**Option**	There may or may not be a result.
2		

Examples of **foldM** usage:

Example	Monadic Context	How the result of **foldM** is affected
1	**Option**	There may or may not be a result.
2	**List**	There may be zero, one, or more results. So **foldM** manages zero, one, or more accumulators.

In the following examples, the following **property** holds…

If **foldM** is applied to
- a list of length **n**
- an initial accumulator **z**
- a function returning a list of length **m**

Then **foldM** returns a list of length **m** ^ **n**.

```
assert(
  List(...).foldM(...)((acc, val) => List(...))
  ==    n=?          z                    m=?
  List(...)
)       m^n=?
```

In the following examples, the following **property** holds…

If **foldM** is applied to
- a list of length **n**
- an initial accumulator **z**
- a function returning a list of length **m**

Then **foldM** returns a list of length **m** ^ **n**.

```
assert(
  List(...).foldM(...)((acc, val) => List(...))
  ==    n=?          z                    m=?
  List(...)
)       m^n=?
```

```
assert(
  List().foldM(...)((acc, val) => List(...))
  ==    n=0          z                    m=?
  List(z)
)       m^n=1
```

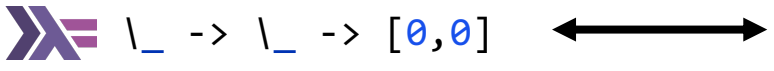
when **n**=0, the returned list is a singleton containing **z**

In this first example, let's choose a **binary function** that ignores both its parameters.
This is to stress the fact that the **property** holds regardless of the particular values contained in the lists.

```
Let

  f = (_, _) => List(0, 0)
  List(x1,x2,x3) = List(1,2,3)
  a1 = 9

In

  List(x1, x2, ..., xm).foldM(a1)(f)

i.e.

  (for
     a2 <- f(a1,x1)
     a3 <- f(a2,x2)
     ...
   yield f(am,xm)).flatten


Result:

  List(0,0,0,0,0,0,0,0)
```

```
assert(
  List(1, 2, 3).foldM(9)((_, _) => List(0, 0))
  == n=3              z                m=2
  List(0, 0, 0, 0, 0, 0, 0, 0)
)          m^n = 2^3 = 8
```

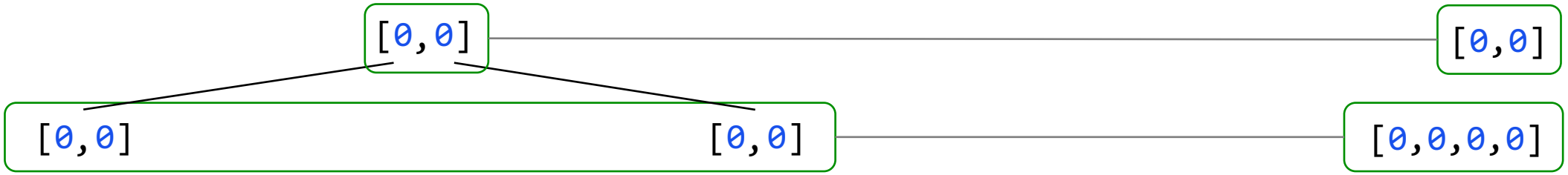
```
assert(
  List().foldM(9)((_, _) => List(0, 0))
  == n=0       z                m=2
  List(9)
)     m^n = 2^0 = 1
```

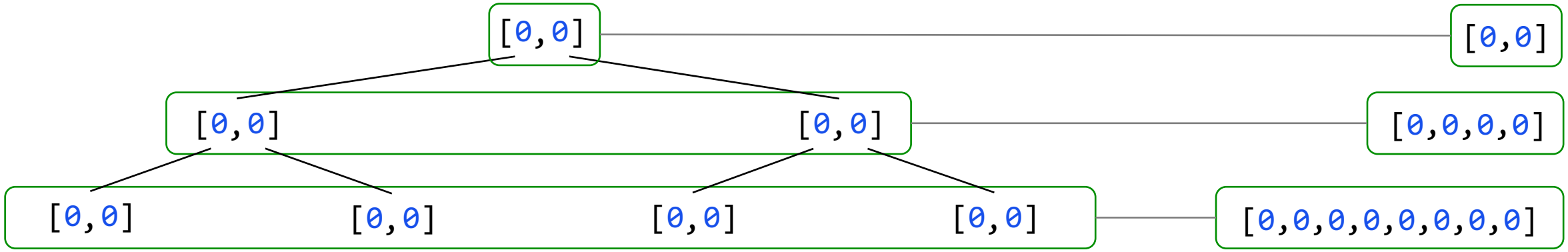
Invocation	m	n	Result	Result length (m^n)
`foldM (\_ -> \_ -> [0,0]) 9 []`	2	0	[9]	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	2	1	[0,0]	2^1=2

Invocation	m	n	Result	Result length (m^n)
`foldM (\_ -> \_ -> [0,0]) 9 []`	2	0	`[9]`	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	2	1	`[0,0]`	2^1=2

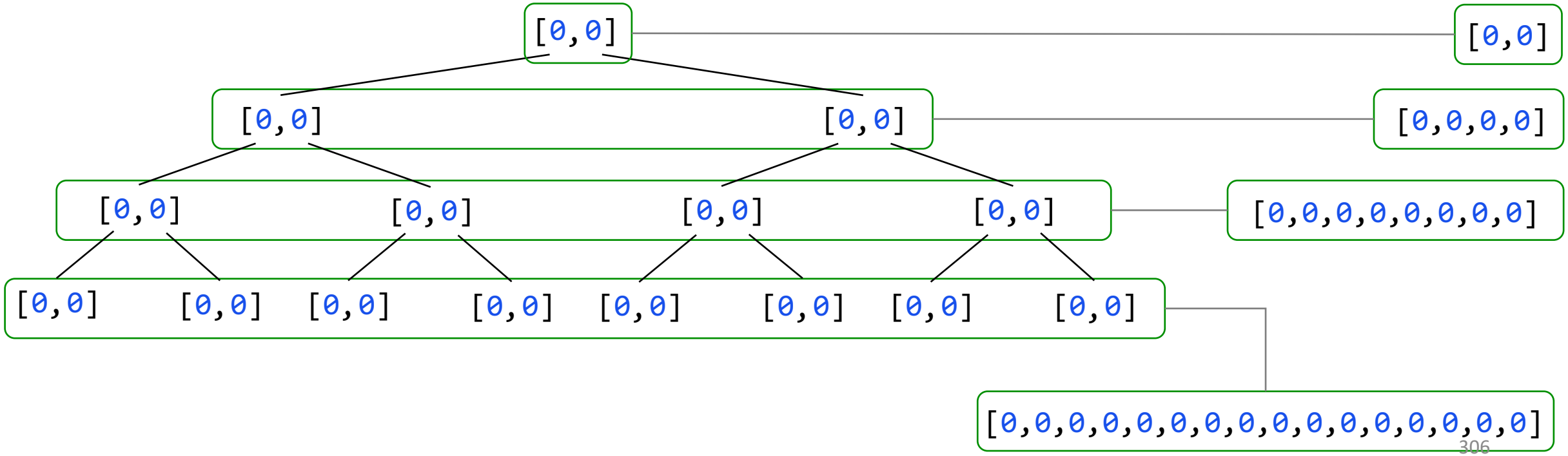
Invocation	m	n	Result	Result length (m^n)
`foldM (\_ -> \_ -> [0,0]) 9 []`	2	0	`[9]`	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	2	1	`[0,0]`	2^1=2
`foldM (\_ -> \_ -> [0,0]) 9 [1,2]`	2	2	`[0,0,0,0]`	2^2=4



Invocation	m	n	Result	Result length (m^n)
`foldM (\_ -> \_ -> [0,0]) 9 []`	2	0	`[9]`	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	2	1	`[0,0]`	2^1=2
`foldM (\_ -> \_ -> [0,0]) 9 [1,2]`	2	2	`[0,0,0,0]`	2^2=4
`foldM (\_ -> \_ -> [0,0]) 9 [1,2,3]`	2	3	`[0,0,0,0,0,0,0,0]`	2^3=8



Invocation	m	n	Result	Result length (m^n)
`foldM (\_ -> \_ -> [0,0]) 9 []`	2	0	`[9]`	2^0=1
`foldM (\_ -> \_ -> [0,0]) 9 [1]`	2	1	`[0,0]`	2^1=2
`foldM (\_ -> \_ -> [0,0]) 9 [1,2]`	2	2	`[0,0,0,0]`	2^2=4
`foldM (\_ -> \_ -> [0,0]) 9 [1,2,3]`	2	3	`[0,0,0,0,0,0,0,0]`	2^3=8
`foldM (\_ -> \_ -> [0,0]) 9 [1,2,3,4]`	2	4	`[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]`	2^4=16



In this second example, we change the **binary function** that we pass to **foldM** so that…

In this second example, we change the **binary function** that we pass to **foldM** so that…

…rather than ignoring its two parameters (the **accumulator acc** and the current list element **x**) and always returning the same two-element list `List(0,0)` …

```
(_, _) => List(0, 0)
```


In this second example, we change the **binary function** that we pass to **foldM** so that…

…rather than ignoring its two parameters (the **accumulator acc** and the current list element **x**) and always returning the same two-element list `List(0,0)` …

```
(_, _) => List(0, 0)
```

…the function now returns a two-element list containing
1. the result of adding the current element to the **accumulator** and
2. the result of subtracting the current element from the **accumulator**

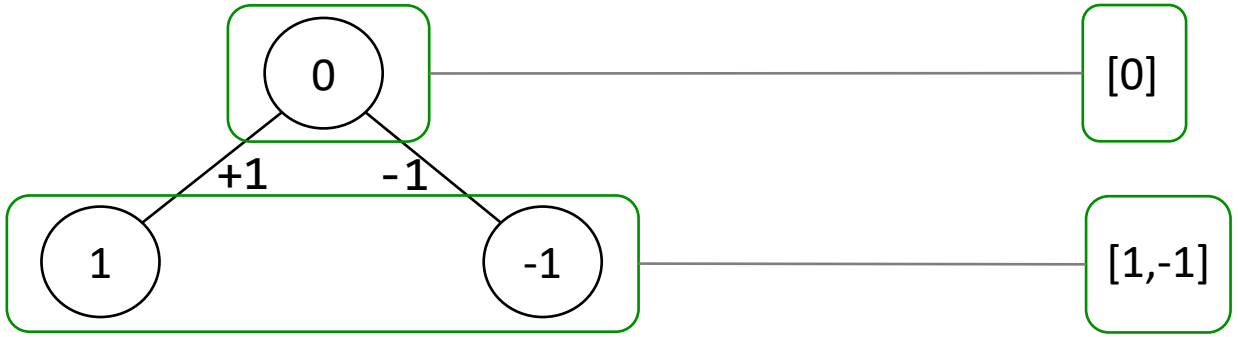
```
(acc,x) => List(acc+x, acc-x)
```


```
Let

  f = (acc,x) => List(acc+x, acc-x)
  List(x1,x2,x3) = List(1,2,3)
  a1 = 0

In

  List(x1, x2, ..., xm).foldM(a1)(f)

i.e.

  (for
     a2 <- f(a1,x1)
     a3 <- f(a2,x2)
     ...
   yield f(am,xm)).flatten


Result:

 List(6,0,2,-4,4,-2,0,-6))
```

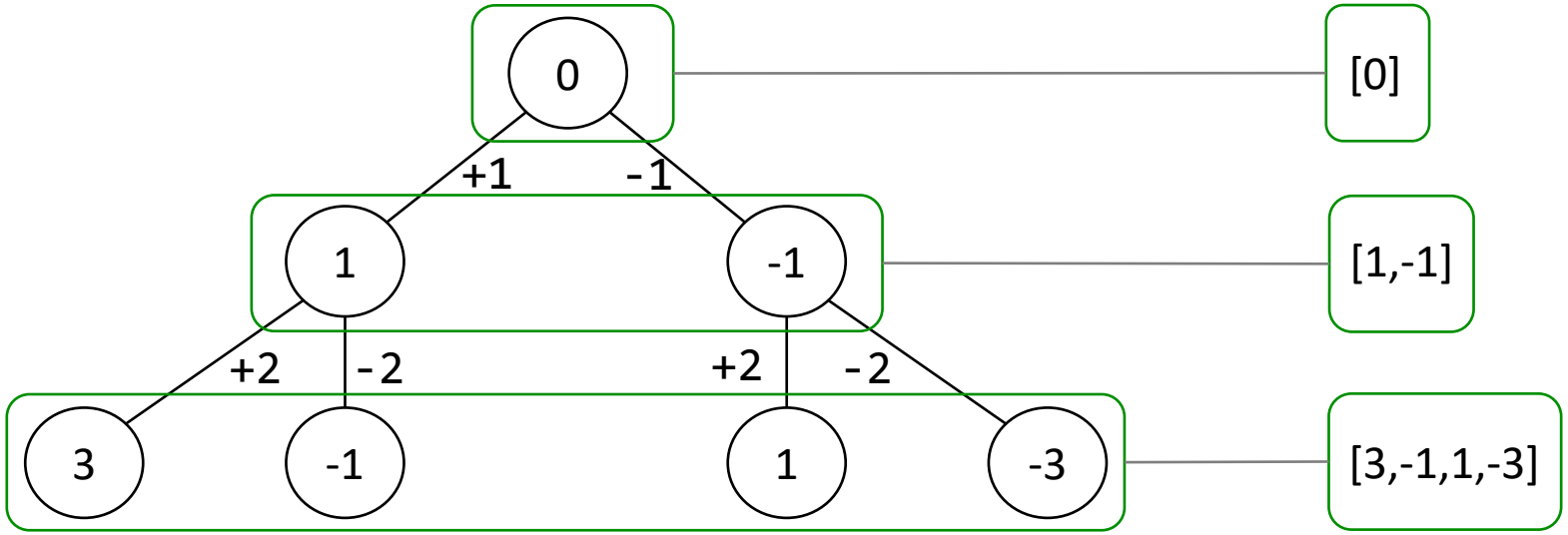
```
assert(
  List(1,2,3).foldM(0)((acc,x) => List(acc+x, acc-x))
  ==
  List(6,0,2,-4,4,-2,0,-6)
)
```


Invocation	Result	Result length (m^n)
`foldM (\acc x -> [acc + x, acc - x]) 0 []`	`[0]`	`2^0=1`

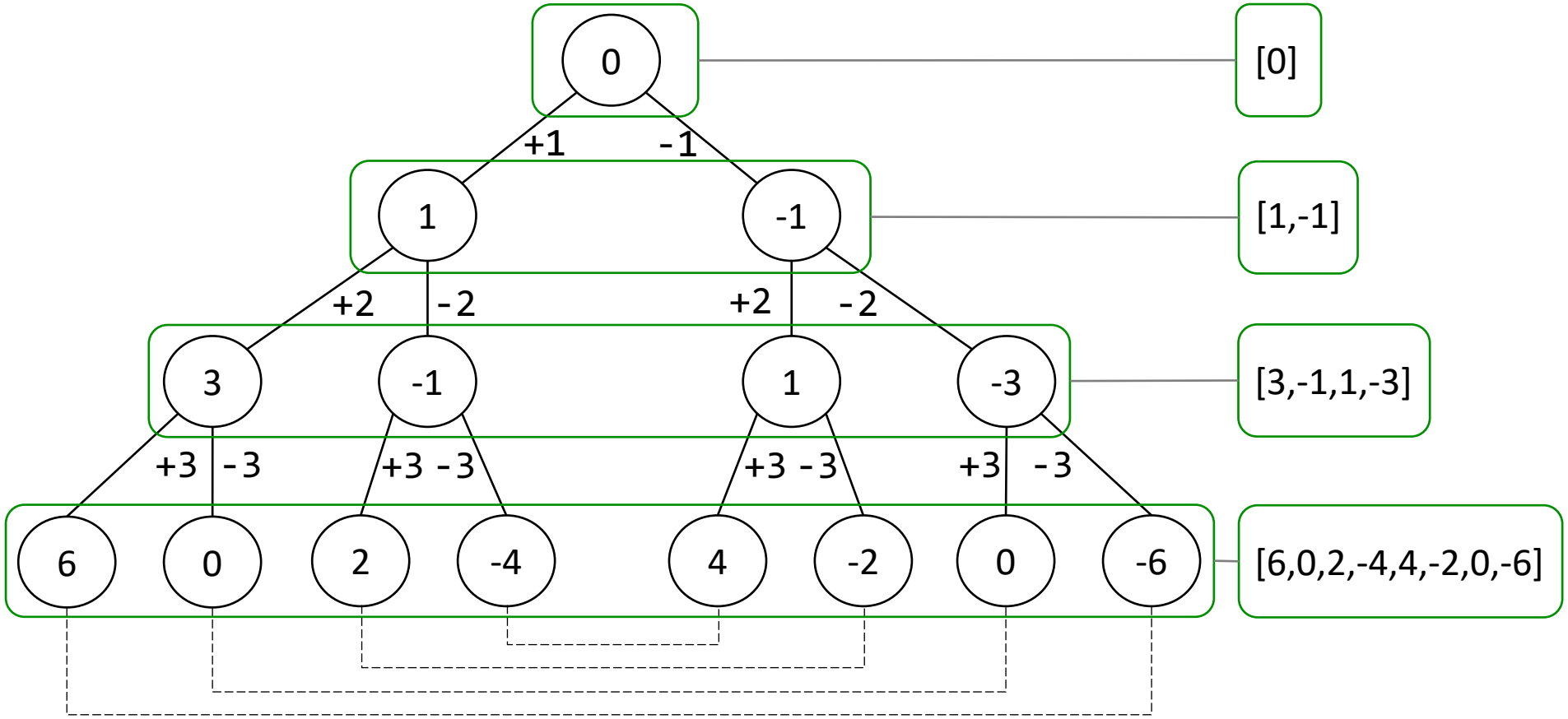
Invocation	Result	Result length (m^n)
`foldM (\acc x -> [acc + x, acc - x]) 0 []`	`[0]`	2^0=1
`foldM (\acc x -> [acc + x, acc - x]) 0 [1]`	`[1,-1]`	2^1=2



Invocation	Result	Result length (m^n)
`foldM (\acc x -> [acc + x, acc - x]) 0 []`	`[0]`	2^0=1
`foldM (\acc x -> [acc + x, acc - x]) 0 [1]`	`[1,-1]`	2^1=2
`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2]`	`[3,-1,1,-3]`	2^2=4



Invocation	Result	Result length (m^n)
`foldM (\acc x -> [acc + x, acc - x]) 0 []`	`[0]`	2^0=1
`foldM (\acc x -> [acc + x, acc - x]) 0 [1]`	`[1,-1]`	2^1=2
`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2]`	`[3,-1,1,-3]`	2^2=4
`foldM (\acc x -> [acc + x, acc - x]) 0 [1,2,3]`	`[6,0,2,-4,4,-2,0,-6]`	2^3=8



In this third example, we change the **binary function** that we pass to **foldM** so that…

In this third example, we change the **binary function** that we pass to **foldM** so that…

…rather than returning a two-element list containing
1) the result of adding the current element **x** to the **accumulator**
2) the result of subtracting **x** from the **accumulator**

```
(acc,x) => List(acc+x, acc-x)
```


In this third example, we change the **binary function** that we pass to **foldM** so that…

…rather than returning a two-element list containing
1) the result of adding the current element **x** to the **accumulator**
2) the result of subtracting **x** from the **accumulator**

```
(acc,x) => List(acc+x, acc-x)
```

…it returns a two element list containing
1) the result of adding **x** to the front of the **accumulator list**
2) the **accumulator list**

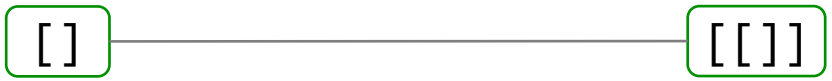
```
(acc,x) => List(x::acc, acc)
```

As a result, **foldM** computes the **powerset** of its list parameter.

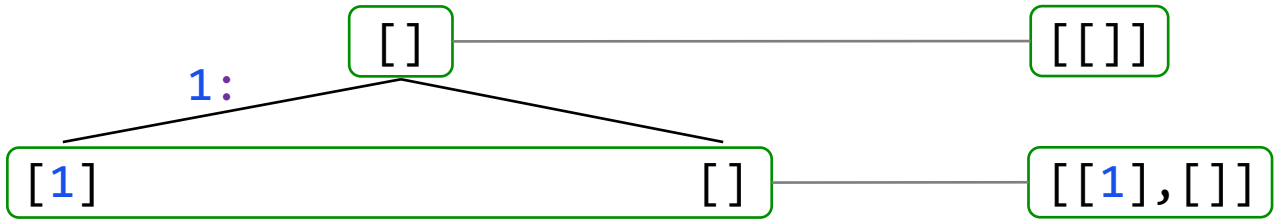
```
Let

  f = (acc,x) => List(x::acc, acc)
  List(x1,x2,x3) = List(1,2,3)
  a1 = List.empty

In

  List(x1, x2, ..., xm).foldM(a1)(f)

i.e.

  (for
     a2 <- f(a1,x1)
     a3 <- f(a2,x2)
     ...
   yield f(am,xm)).flatten


Result:

  List(
    List(3,2,1), List(2,1),
    List(3,1), List(1), List(3,2),
    List(2), List(3), List()))
```

```
assert(
  List(1,2,3).foldM(List.empty)((acc,x) => List(x::acc, acc))
  ==
  List(
    List(3,2,1),
    List(2,1),
    List(3,1),
    List(1),
    List(3,2),
    List(2),
    List(3),
    List())
  )
)
```

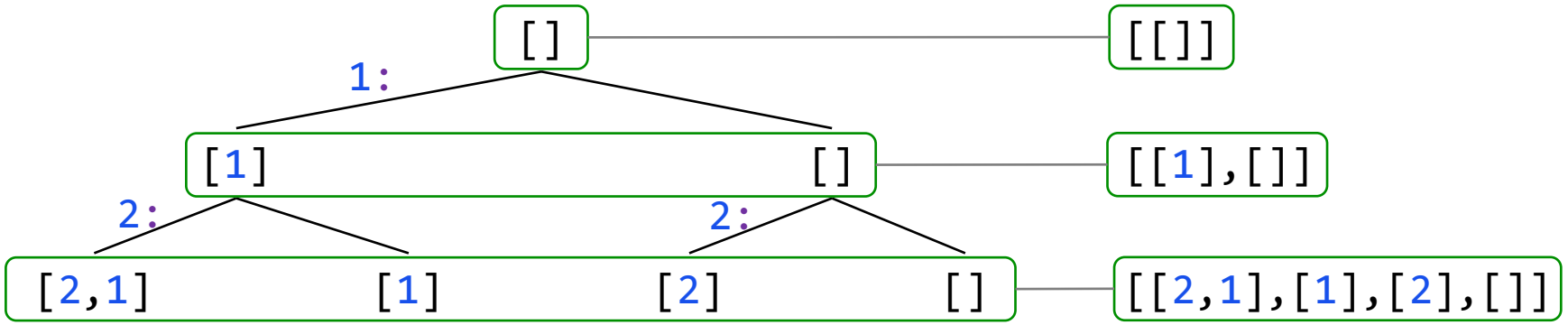

Invocation	Result	Result length (m^n)
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1



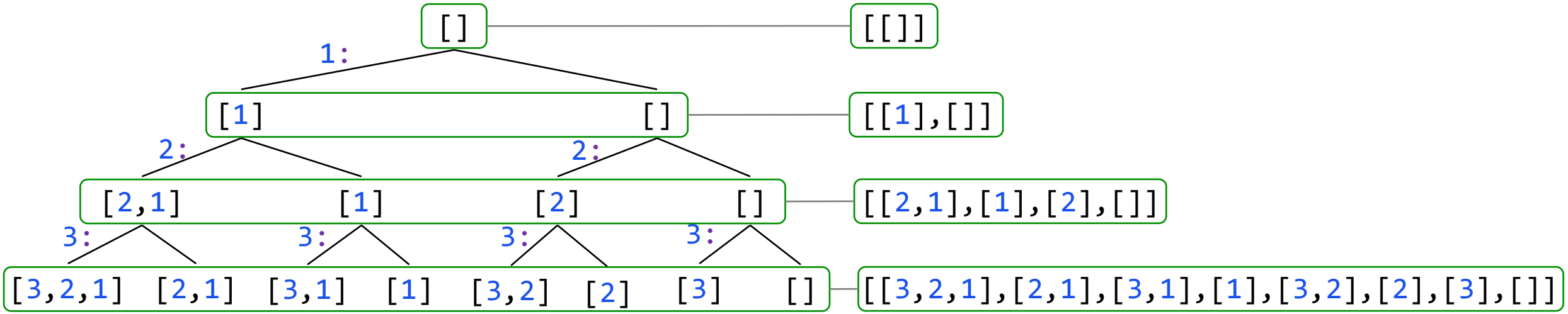
Invocation	Result	Result length (m^n)
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1
`foldM (\acc x -> [x:acc,acc]) [] [1]`	`[[1],[]]`	2^1=2



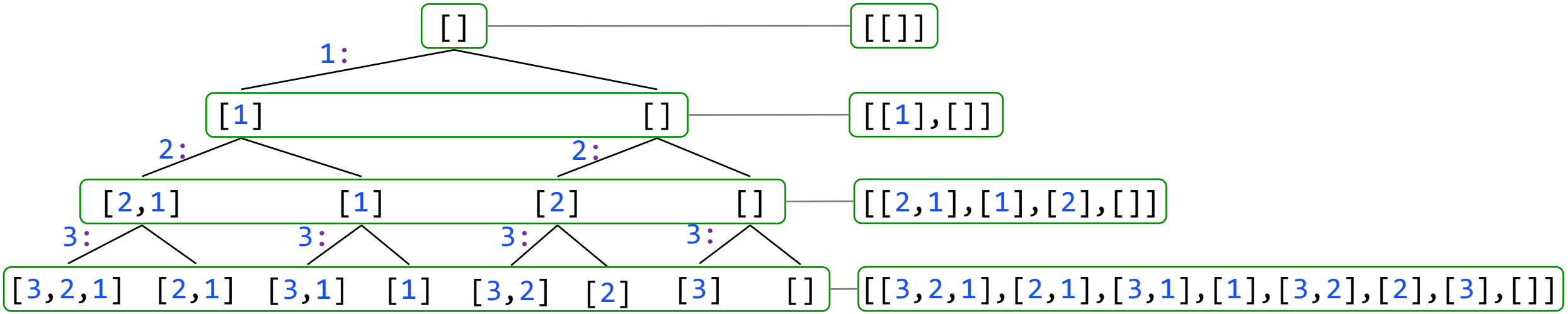
Invocation	Result	Result length (m^n)
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1
`foldM (\acc x -> [x:acc,acc]) [] [1]`	`[[1],[]]`	2^1=2
`foldM (\acc x -> [x:acc,acc]) [] [1,2]`	`[[2,1],[1],[2],[]]`	2^2=4



Invocation	Result	Result length (m^n)
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1
`foldM (\acc x -> [x:acc,acc]) [] [1]`	`[[1],[]]`	2^1=2
`foldM (\acc x -> [x:acc,acc]) [] [1,2]`	`[[2,1],[1],[2],[]]`	2^2=4
`foldM (\acc x -> [x:acc,acc]) [] [1,2,3]`	`[[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]`	2^3=8



Invocation	Result	Result length (m^n)
`foldM (\acc x -> [x:acc,acc]) [] []`	`[[]]`	2^0=1
`foldM (\acc x -> [x:acc,acc]) [] [1]`	`[[1],[]]`	2^1=2
`foldM (\acc x -> [x:acc,acc]) [] [1,2]`	`[[2,1],[1],[2],[]]`	2^2=4
`foldM (\acc x -> [x:acc,acc]) [] [1,2,3]`	`[[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]`	2^3=8



Yes, this is the same process that we saw in this previous example: `List(1, 2, 3).filterM(_ => List(true, false))`

So the following also computes the powerset of the input list: `List(1, 2, 3).foldM(List.empty)((acc,x) => List(x::acc, acc))`

The three **foldM** examples that we have just gone through

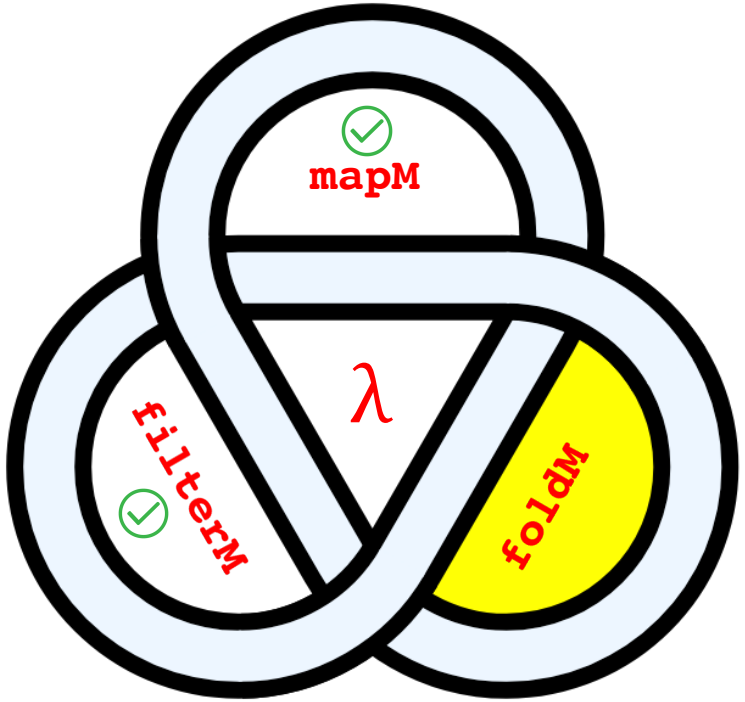
```
import Control.Monad
```

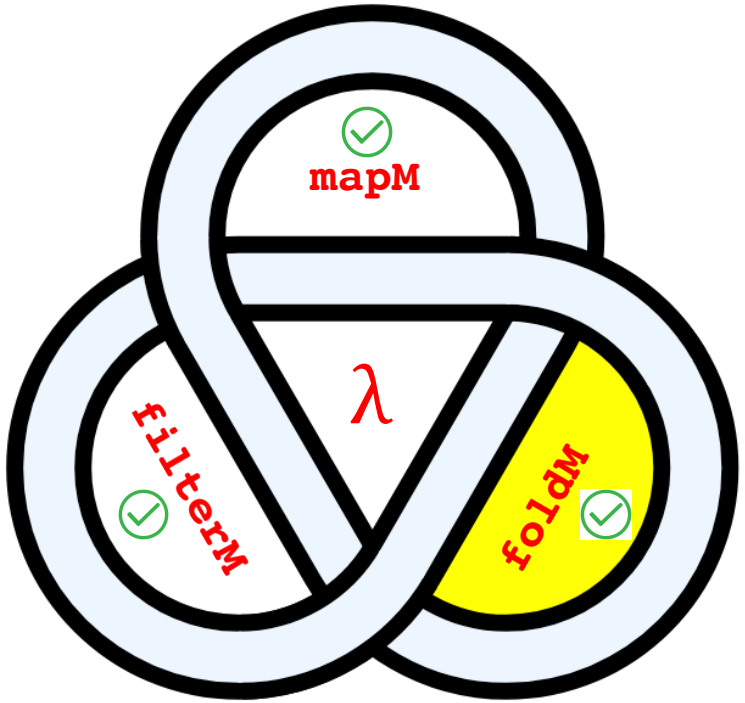
```
assertEqual
  "foldM test 1"
  (foldM (\_ _ -> [0,0]) 9 [1,2,3,4])
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

assertEqual
  "foldM test 2"
  (foldM (\acc x -> [acc+x,acc-x]) 0 [1,2,3])
  [6,0,2,-4,4,-2,0,-6]

assertEqual
  "foldM test 3"
  (foldM (\acc x -> [x:acc,acc])[] [1,2,3])
  [[3,2,1],[2,1],[3,1],[1],[3,2],[2],[3],[]]
```

```
import cats.syntax.foldable._
```


```
assert(
  List(1,2,3,4).foldM(9)((_,_) => List(0, 0))
  ==
  List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0))

assert(
  List(1,2,3).foldM(0)((acc,x) => List(acc+x, acc-x))
  ==
  List(6,0,2,-4,4,-2,0,-6))

assert(
  List(1,2,3).foldM(List.empty)((acc,x) => List(x::acc,acc))
  ==
  List(List(3,2,1),List(2,1),List(3,1),List(1),List(3,2),List(2),List(3),List()))
```

In the rest of this talk we'll refer to the function passed to **foldM** as an **updater function**


mapM
λ
filterM
foldM


mapM
λ
filterM
foldM

Now that we have gained some familiarity with the **foldM** function...

…let's begin to see how it can be used to solve the **N-Queens combinatorial problem**.

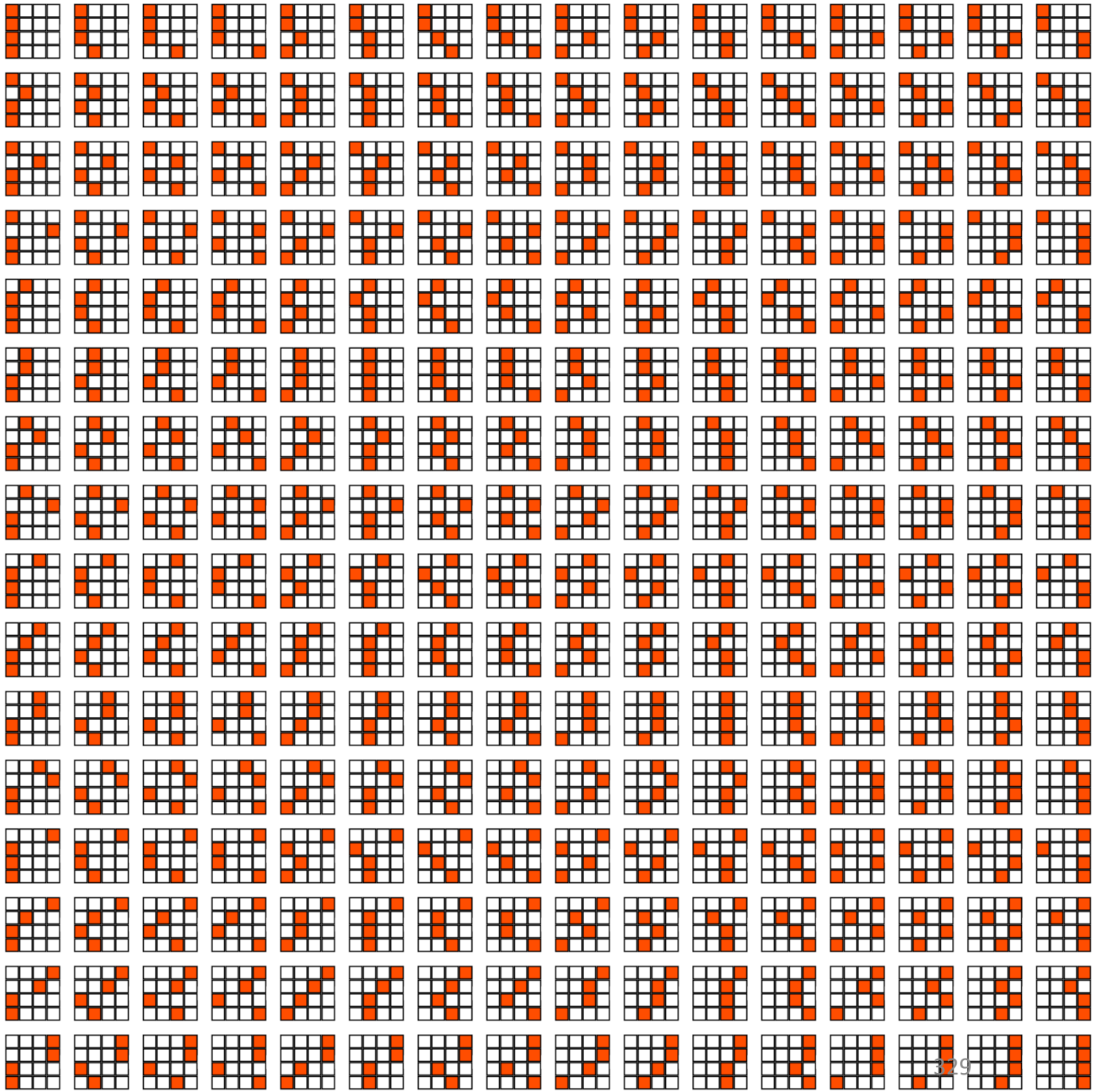
Let’s refer to the solution that uses **foldM** as the **folding queens solution**.

The **folding queens solution** needs to generate the **permutations** of a list of integers.

N = 4
# of permutations = $4^4 = 256$

```
def permutations(n:Int = 4): List[List[Int]] =
  if n == 0 then List(List())
  else
    for
      queens <- permutations(n-1)
      queen <- 1 to 4
    yield queen :: queens
```

Permutations with
repetition allowed



Permutations with repetition not allowed

The number of **permutations** of a list of length n is $n!$, the **factorial** of n.

e.g. the number of **permutations** of a list of three elements is

$$3! = 3 * 2 * 1 = 6$$

List

(1) (2) (3)

Permutations

(1) (2) (3)

(1) (3) (2)

(2) (1) (3)

(2) (3) (1)

(3) (1) (2)

(3) (2) (1)

Permutations with repetition not allowed

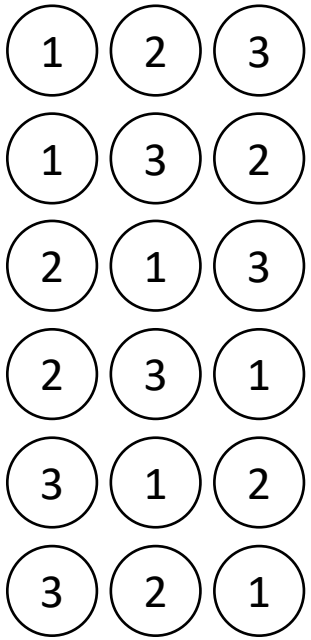
The number of **permutations** of a list of length n is $n!$, the **factorial** of n.

e.g. the number of **permutations** of a list of three elements is

$$3! = 3 * 2 * 1 = 6$$

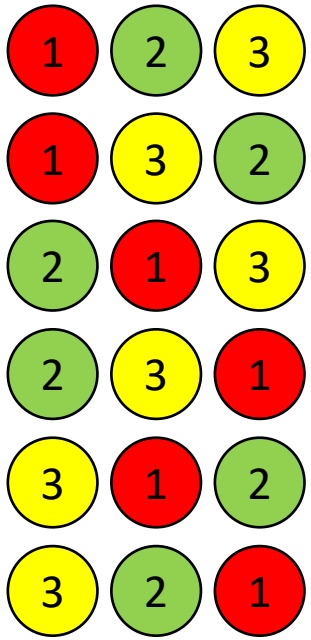
List

Permutations



List

Permutations

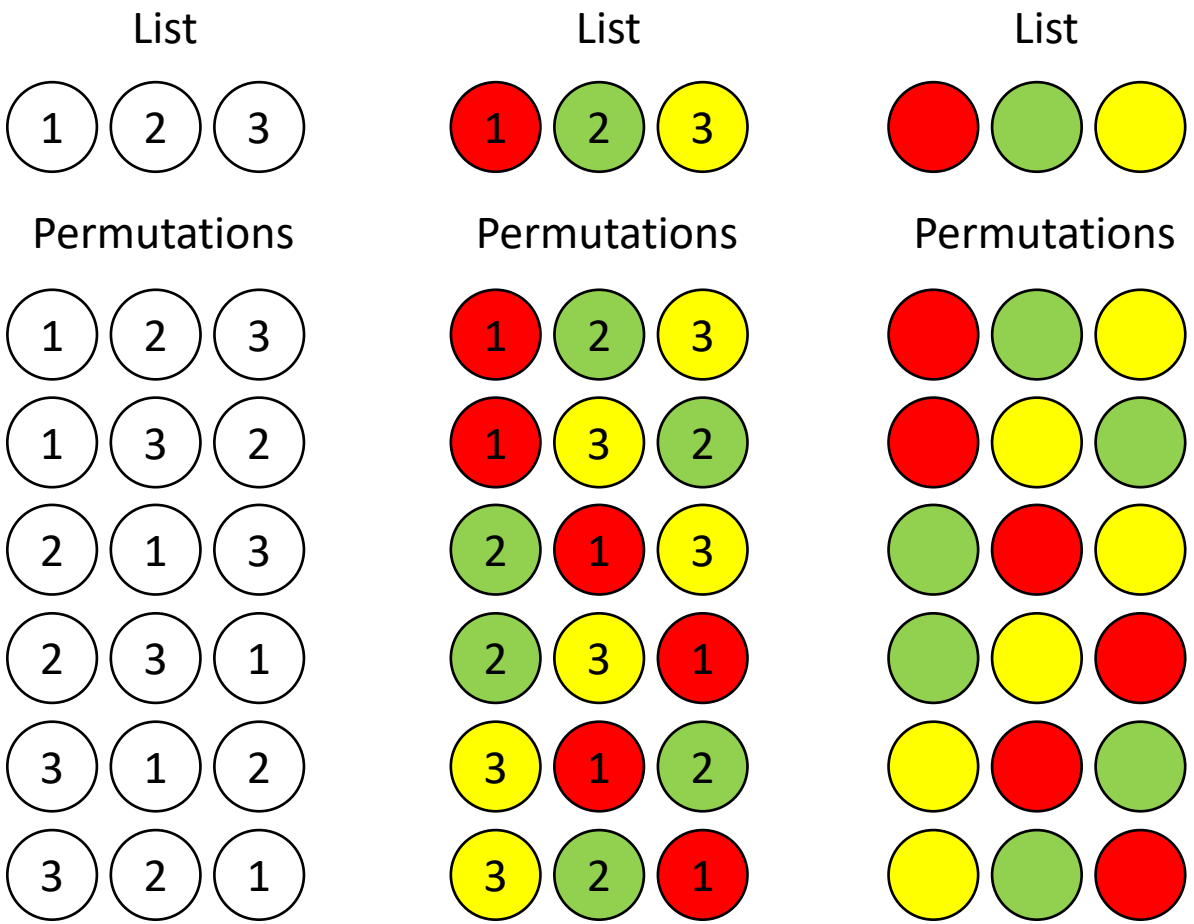


Permutations with repetition not allowed

The number of **permutations** of a list of length n is $n!$, the **factorial** of n.

e.g. the number of **permutations** of a list of three elements is

$$3! = 3 * 2 * 1 = 6$$



Let’s look at an **updater function** that can be used to generate the **permutations** of a list.

```
update :: Eq a => ([a], [a]) -> p -> [([a], [a])]
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

Let's look at an **updater function** that can be used to generate the **permutations** of a list.

```
update :: Eq a => ([a], [a]) -> p -> [([a], [a])]
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

Invocation

```
foldM update ([],[1,2,3]) [1,2,3]
```

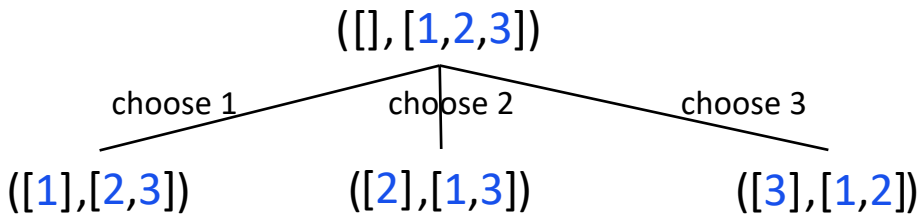
Result

```
[([3,2,1],[]),
 ([2,3,1],[]),
 ([3,1,2],[]),
 ([1,3,2],[]),
 ([2,1,3],[]),
 ([1,2,3],[])]
```

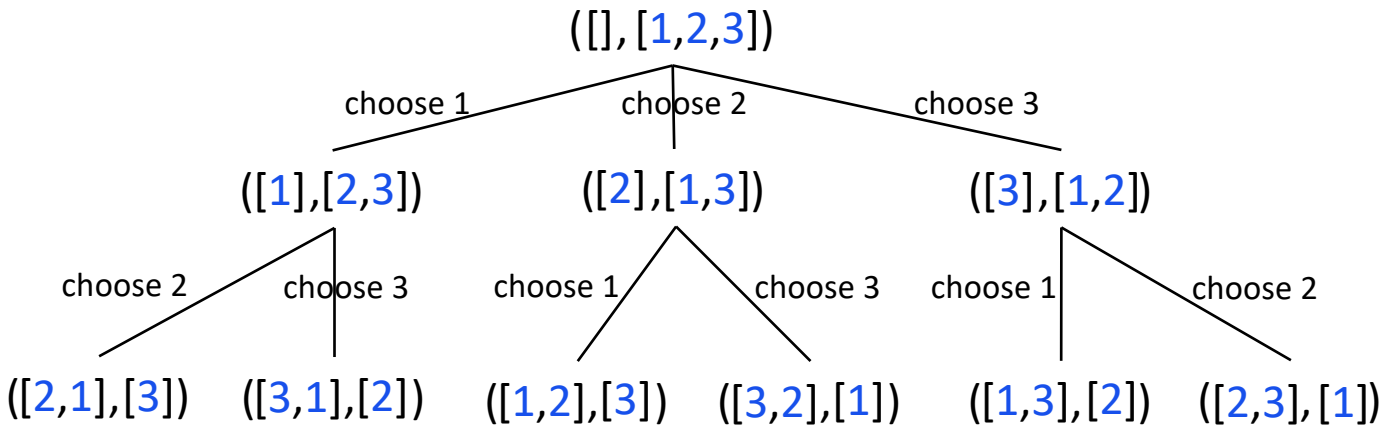
Invocation	Result
`foldM update ([],[1,2,3]) []`	`[([],[1,2,3])]`

([], [1,2,3])

Invocation	Result
`foldM update ([],[1,2,3]) []`	`[([],[1,2,3])]`
`foldM update ([],[1,2,3]) [1]`	`[([1],[2,3]), ([2],[1,3]), ([3],[1,2])]`

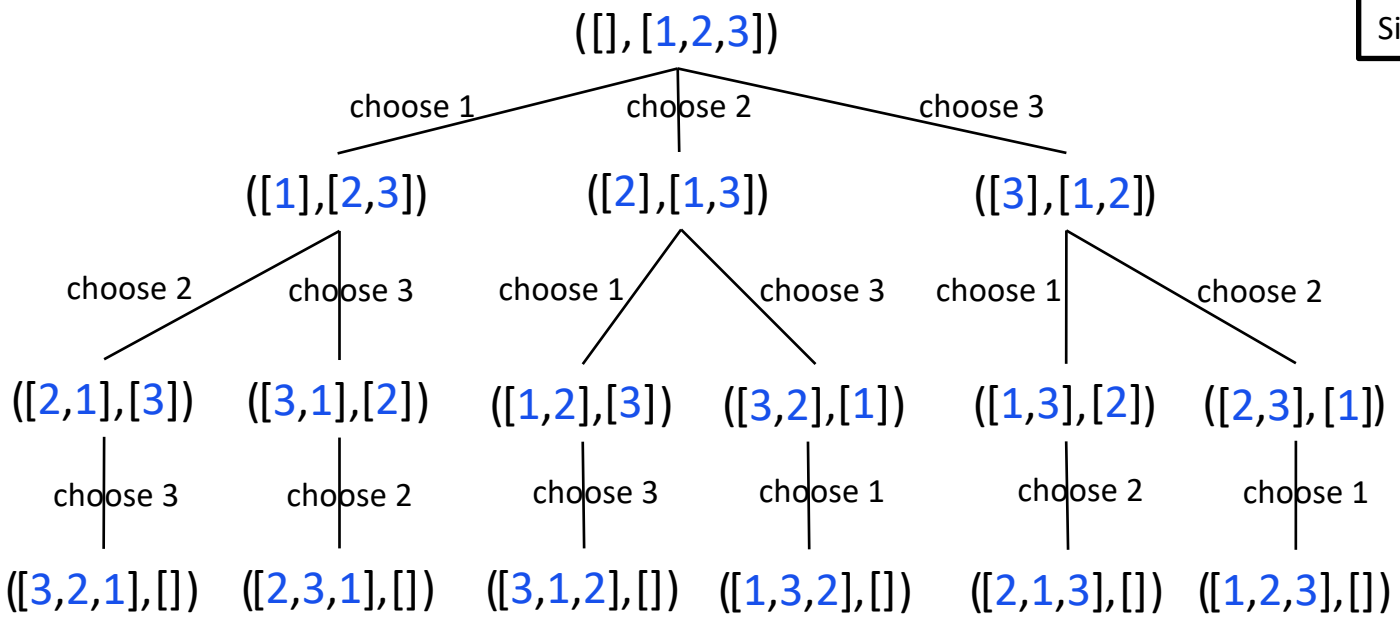


Invocation	Result
`foldM update ([],[1,2,3]) []`	`[([],[1,2,3])]`
`foldM update ([],[1,2,3]) [1]`	`[([1],[2,3]), ([2],[1,3]), ([3],[1,2])]`
`foldM update ([],[1,2,3]) [1,2]`	`[([2,1],[3]), ([3,1],[2]), ([1,2],[3]), ([3,2],[1]), ([1,3],[2]), ([2,3],[1])]`



Invocation	Result
`foldM update ([],[1,2,3]) []`	`[([],[1,2,3])]`
`foldM update ([],[1,2,3]) [1]`	`[([1],[2,3]), ([2],[1,3]), ([3],[1,2])]`
`foldM update ([],[1,2,3]) [1,2]`	`[([2,1],[3]), ([3,1],[2]), ([1,2],[3]), ([3,2],[1]), ([1,3],[2]), ([2,3],[1])]`
`foldM update ([],[1,2,3]) [1,2,3]`	`[([3,2,1],[]), ([2,3,1],[]), ([3,1,2],[]), ([1,3,2],[]), ([2,1,3],[]), ([1,2,3],[])]`

Size of input list being folded: *N*
Size of output list containing permutations: *N!*



The result of **foldM** is not exactly a list of **permutations**, but rather, a list of a pair of a **permutation** and the empty list.

The result of **foldM** is not exactly a list of **permutations**, but rather, a list of a pair of a **permutation** and the empty list.

Let's defined a couple of **helper functions** to make it more convenient to get hold of the desired list of **permutations**.

```
permute :: Eq a => [a] -> [([a], [a])]
permute xs = foldM update ([],xs) xs

permutations :: (Eq a) => [a] -> [[a]]
permutations xs = map fst (permute xs)
```

```
> permutations []
[[]]

> permutations [1]
[[1]]

> permutations [1,2]
[[2,1],[1,2]]

> permutations [1,2,3]
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]

> permutations [1,2,3,4]
[[4,3,2,1],[3,4,2,1],[4,2,3,1],[2,4,3,1],[3,2,4,1]
,[2,3,4,1],[4,3,1,2],[3,4,1,2],[4,1,3,2],[1,4,3,2]
,[3,1,4,2],[1,3,4,2],[4,2,1,3],[2,4,1,3],[4,1,2,3]
,[1,4,2,3],[2,1,4,3],[1,2,4,3],[3,2,1,4],[2,3,1,4]
,[3,1,2,4],[1,3,2,4],[2,1,3,4],[1,2,3,4]]
```

```scala
def update[A](acc:(List[A], List[A]), a:A): List[(List[A], List[A])] = acc match
  case (permutation, choices) =>
    for
      choice <- choices
    yield (choice :: permutation, choices.diff(List(choice)))
```

```scala
assert(
  List(1,2,3).foldM((List.empty[Int],List(1,2,3)))(update)
  ==
  List(
    (List(3,2,1),List()),
    (List(2,3,1),List()),
    (List(3,1,2),List()),
    (List(1,3,2),List()),
    (List(2,1,3),List()),
    (List(1,2,3),List()))
)
```

```scala
def permute[A](as: List[A]): List[(List[A], List[A])] =
  as.foldM((List.empty[A], as))(update)

def permutations[A](as: List[A]): List[List[A]] =
  permute(as).map(_.head)
```

```scala
assert(
  permutations(List(1,2,3))
  ==
  List(
    List(3, 2, 1),
    List(2, 3, 1),
    List(3, 1, 2),
    List(1, 3, 2),
    List(2, 1, 3),
    List(1, 2, 3)
  )
)
```

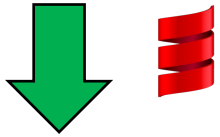
We want to use **foldM** to solve the **N-Queens combinatorial problem**.

So let's rename the variables of the **update** function to reflect the fact that the **permutations** that we want to generate are the possible lists of positions (columns) of n queens on an $n \times n$ board.

```
update (permutation,choices) _ = [(choice:permutation, delete choice choices) | choice <- choices]
```

```
oneMoreQueen (queens,emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns]
```

```
def update[A](acc:(List[A], List[A]), a:A): List[(List[A], List[A])] = acc match
  case (permutation, choices) =>
    for
      choice <- choices
    yield (choice :: permutation, choices.diff(List(choice)))
```



```
def oneMoreQueen[A](acc:(List[A], List[A]), a:A): List[(List[A], List[A])] = acc match
  case (queens, emptyColumns) =>
    for
      queen <- emptyColumns
    yield (queen :: queens, emptyColumns.diff(List(queen)))
```

Not all **permutations** are valid though: we need to filter out unsafe **permutations**.

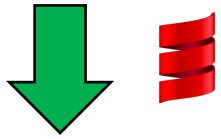
Just like in the **recursive solution**, we are going to determine if a **permutation** is safe is by using a **safe** function.

Let's add to **oneMoreQueen** a filter that invokes the **safe** function.

```
oneMoreQueen (queens, emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns]
```

```
oneMoreQueen (queens, emptyColumns) _ = [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns,isSafe queen]
```

```
def oneMoreQueen[A](acc:(List[A], List[A]), a:A): List[(List[A], List[A])] = acc match
  case (queens, emptyColumns) =>
    for
      queen <- emptyColumns
    yield (queen :: queens, emptyColumns.diff(List(queen)))
```

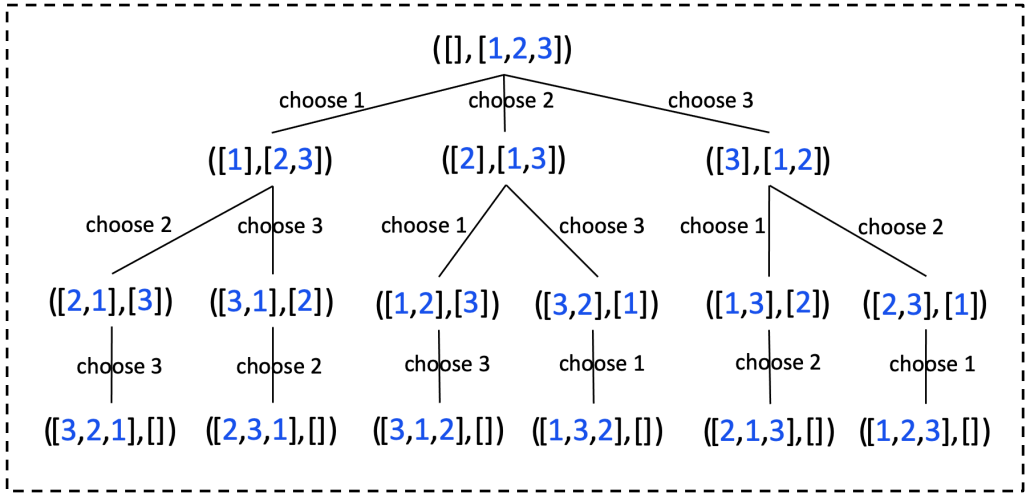


```
def oneMoreQueen[A](acc:(List[A], List[A]), a:A): List[(List[A], List[A])] = acc match
  case (queens, emptyColumns) =>
    for
      queen <- emptyColumns
      if isSafe(queen)
    yield (queen :: queens, emptyColumns.diff(List(queen)))
```

Earlier we tried out our **update** function as follows (to compute permutations)

```
foldM update ([],[1,2,3]) [1,2,3]
```

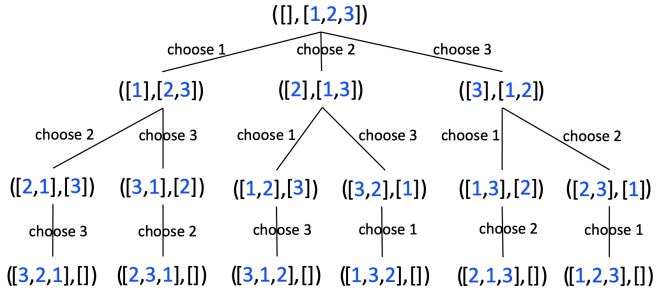
```
List(1,2,3).foldM((List.empty[Int],List(1,2,3)))(update)
```



Earlier we tried out our **update** function as follows (to compute permutations)

```
foldM update ([],[1,2,3]) [1,2,3]
```

```
List(1,2,3).foldM((List.empty[Int],List(1,2,3)))(update)
```



The **queens** function that we need to implement is this:

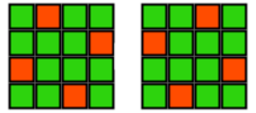
```
queens :: Int -> [[Int]]
queens n = ???
```

```
def queens(n: Int): List[List[Int]] = ???
```

```
queens(4)
```

⇩

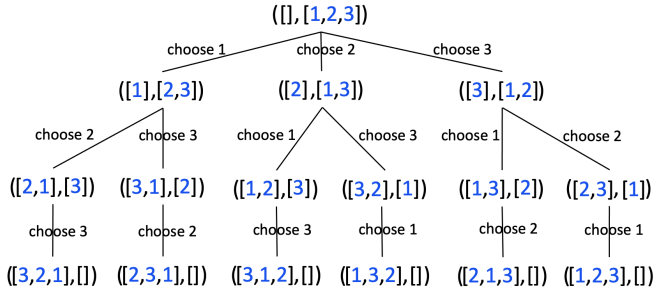
```
List(List(3,1,4,2),List(2,4,1,3)))
```



Earlier we tried out our **update** function as follows (to compute permutations)

```
foldM update ([],[1,2,3]) [1,2,3]
```

```
List(1,2,3).foldM((List.empty[Int],List(1,2,3)))(update)
```



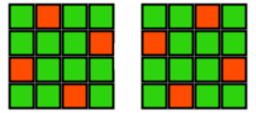
The **queens** function that we need to implement is this:

```
queens :: Int -> [[Int]]
queens n = ???
```

```
def queens(n: Int): List[List[Int]] = ???
```

```
queens(4)
```

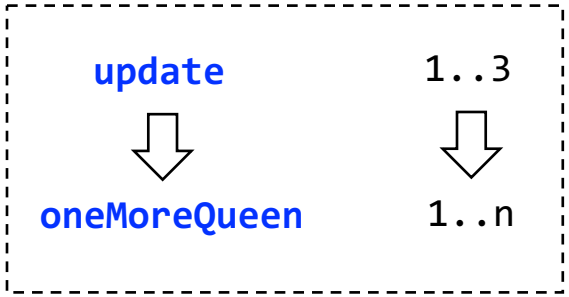
```
List(List(3,1,4,2),List(2,4,1,3)))
```



Given that we have renamed **update** to **oneMoreQueen**, here is how we need to call **foldM**:

```
foldM oneMoreQueen ([],[1..n]) [1..n]
```

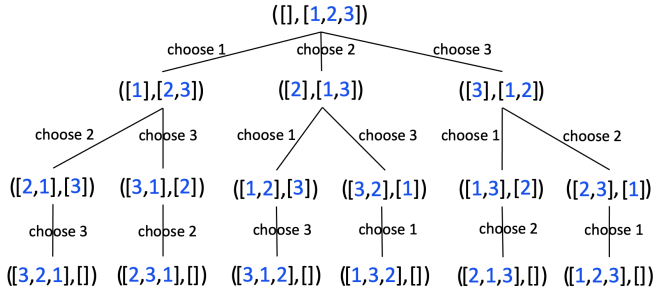
```
List(1 to n *).foldM(List.empty[Int], List(1 to n *))(oneMoreQueen)
```



Earlier we tried out our **update** function as follows (to compute permutations)

```
foldM update ([],[1,2,3]) [1,2,3]
```

```
List(1,2,3).foldM((List.empty[Int],List(1,2,3)))(update)
```



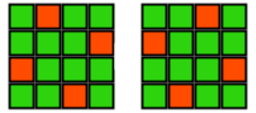
The **queens** function that we need to implement is this:

```
queens :: Int -> [[Int]]
queens n = ???
```

```
def queens(n: Int): List[List[Int]] = ???
```

```
queens(4)
```

```
List(List(3,1,4,2),List(2,4,1,3)))
```



Given that we have renamed **update** to **oneMoreQueen**, here is how we need to call **foldM**:

```
foldM oneMoreQueen ([],[1..n]) [1..n]
```

```
List(1 to n *).foldM(Nil, List(1 to n *))(oneMoreQueen)
```

update  **oneMoreQueen**

1..3  **1..n**

We saw earlier that in order to extract the list of **permutations** / **queens** from the result of **update** / **oneMoreQueen**, we need to map over the result list a function that takes the first element of each pair in the list. So here is how we implement queens:

```
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n])
```

```
def queens(n: Int): List[List[Int]] =
  List(1 to n *).foldM(Nil, List(1 to n *))(oneMoreQueen).map(_.head)
```

```
List[(List[Int], List[Int])]
```

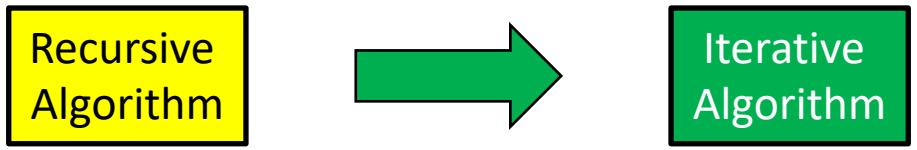
```
List[List[Int]]
```

```haskell
import Control.Monad
```

```haskell
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n]) where
  oneMoreQueen (queens, emptyColumns) _ =
    [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns, isSafe queen] where
    isSafe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] queens]
```

```scala
import cats.syntax.foldable._
```


```scala
def queens(n: Int): List[List[Int]] =
  def oneMoreQueen(acc:(List[Int],List[Int]),x:Int): List[(List[Int],List[Int])] =
    acc match { case (queens, emptyColumns) =>
      def isSafe(x:Int): Boolean = { for (c,n) <- queens zip (1 to n) yield x != c + n && x != c - n } forall identity
      for
        queen <- emptyColumns
        if isSafe(queen)
      yield (queen::queens, emptyColumns diff List(queen)) }
  val oneToN = List(1 to n *)
  oneToN.foldM(Nil, oneToN)(oneMoreQueen).map(_.head)
```


Recursive
Algorithm
Iterative
Algorithm

Scala

Haskell

Recursive Algorithm

```scala
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if isSafe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```

```haskell
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k = [queen:queens |
                     queens <- placeQueens(k-1),
                     queen <- [1..n],
                     isSafe queen queens]
```

```scala
def isSafe(queen: Int, queens: List[Int]): Boolean =
  val (row, column) = (queens.length, queen)
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)
  zipWithRows(queens) forall safe
```

```haskell
isSafe queen queens = all safe (zipWithRows queens)
  where
    safe (r,c) = c /= col && not (onDiagonal col row c r)
    row = length queens
    col = queen
```

```scala
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =
  math.abs(row - otherRow) ==  math.abs(column - otherColumn)
```

```haskell
onDiagonal row column otherRow otherColumn =
  abs (row - otherRow) == abs (column - otherColumn)
```

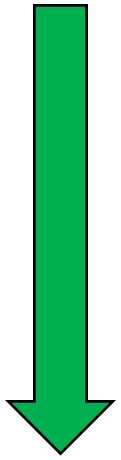
```scala
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =
  val rowCount = queens.length
  val rowNumbers = rowCount - 1 to 0 by -1
  rowNumbers zip queens
```

```haskell
zipWithRows queens = zip rowNumbers queens
  where
    rowCount = length queens
    rowNumbers = [rowCount-1,rowCount-2..0]
```

Haskell

Recursive Algorithm

```
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k = [queen:queens |
                     queens <- placeQueens(k-1),
                     queen <- [1..n],
                     isSafe queen queens]
```



Iterative Algorithm

```
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n]) where
  oneMoreQueen (queens,emptyColumns) _ =
    [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns, isSafe queen] where
    isSafe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] queens]
```

https://rosettacode.org/wiki/N-queens_problem#Haskell

Rosetta Code Version

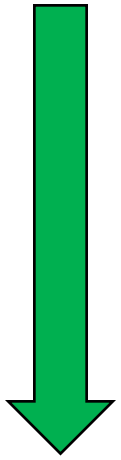
```haskell
-- given n, "queens n" solves the n-queens problem, returning a list of all the
-- safe arrangements. each solution is a list of the columns where the queens are
-- located for each row
queens :: Int -> [[Int]]
queens n = map fst $ foldM oneMoreQueen ([],[1..n]) [1..n]  where

  -- foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
  -- foldM folds (from left to right) in the list monad, which is convenient for
  -- "nondeterminstically" finding "all possible solutions" of something. the
  -- initial value [] corresponds to the only safe arrangement of queens in 0 rows

  -- given a safe arrangement y of queens in the first i rows, and a list of
  -- possible choices, "oneMoreQueen y _" returns a list of all the safe
  -- arrangements of queens in the first (i+1) rows along with remaining choices
  oneMoreQueen (y,d) _ = [(x:y, delete x d) | x <- d, safe x]  where

    -- "safe x" tests whether a queen at column x is safe from previous queens
    safe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] y]
```

Our Version

```haskell
queens :: Int -> [[Int]]
queens n = map fst (foldM oneMoreQueen ([],[1..n]) [1..n]) where

  oneMoreQueen (queens, emptyColumns) _ =
    [(queen:queens, delete queen emptyColumns) | queen <- emptyColumns, isSafe queen] where

    isSafe x = and [x /= c + n && x /= c - n | (n,c) <- zip [1..] queens]
```

Recursive Algorithm

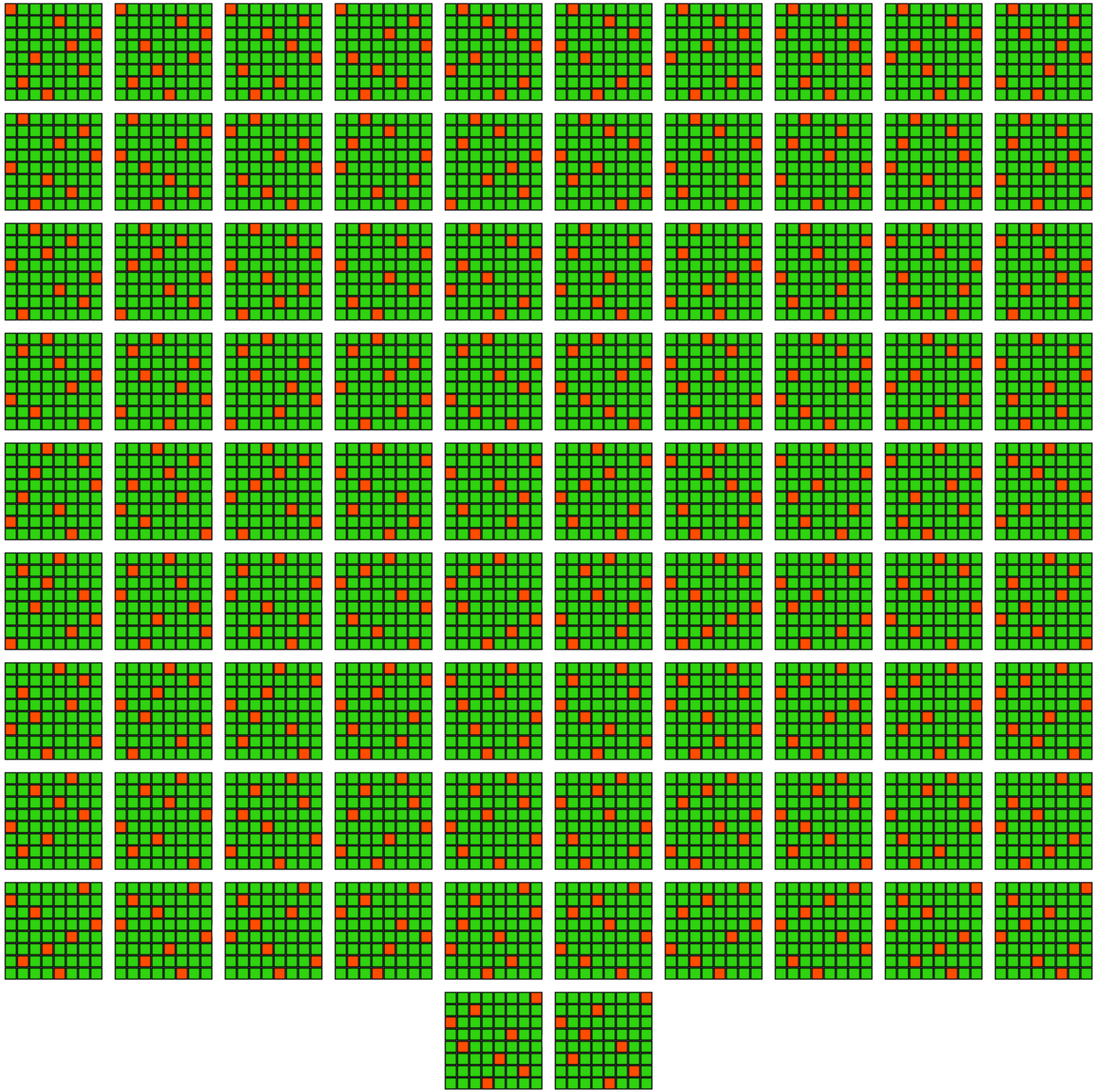
```scala
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if safe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```



```scala
import cats.syntax.foldable._
```

Iterative Algorithm

```scala
def queens(n: Int): List[List[Int]] =
  def oneMoreQueen(acc: (List[Int],List[Int]), x: Int): List[(List[Int],List[Int])] = acc match
    case (queens, emptyColumns) =>
      def isSafe(queen:Int): Boolean = …
      for
        queen <- emptyColumns
        if isSafe(queen)
      yield (queen::queens, emptyColumns diff List(queen))
  List(1 to n *).foldM(Nil, List(1 to n *))(oneMoreQueen).map(_.head)
```

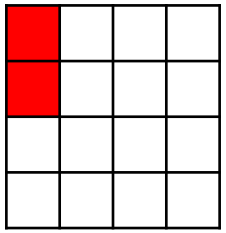


N = 8
92 solutions

```
> queens 8
[[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],[3,5,2,8,6,4,7,1],[3,6,4,2,8,5,7,1],[5,7,1,3,8,6,4,2]
,[4,6,8,3,1,7,5,2],[3,6,8,1,4,7,5,2],[5,3,8,4,7,1,6,2],[5,7,4,1,3,8,6,2],[4,1,5,8,6,3,7,2]
,[3,6,4,1,8,5,7,2],[4,7,5,3,1,6,8,2],[6,4,2,8,5,7,1,3],[6,4,7,1,8,2,5,3],[1,7,4,6,8,2,5,3]
,[6,8,2,4,1,7,5,3],[6,2,7,1,4,8,5,3],[4,7,1,8,5,2,6,3],[5,8,4,1,7,2,6,3],[4,8,1,5,7,2,6,3]
,[2,7,5,8,1,4,6,3],[1,7,5,8,2,4,6,3],[2,5,7,4,1,8,6,3],[4,2,7,5,1,8,6,3],[5,7,1,4,2,8,6,3]
,[6,4,1,5,8,2,7,3],[5,1,4,6,8,2,7,3],[5,2,6,1,7,4,8,3],[6,3,7,2,8,5,1,4],[2,7,3,6,8,5,1,4]
,[7,3,1,6,8,5,2,4],[5,1,8,6,3,7,2,4],[1,5,8,6,3,7,2,4],[3,6,8,1,5,7,2,4],[6,3,1,7,5,8,2,4]
,[7,5,3,1,6,8,2,4],[7,3,8,2,5,1,6,4],[5,3,1,7,2,8,6,4],[2,5,7,1,3,8,6,4],[3,6,2,5,8,1,7,4]
,[6,1,5,2,8,3,7,4],[8,3,1,6,2,5,7,4],[2,8,6,1,3,5,7,4],[5,7,2,6,3,1,8,4],[3,6,2,7,5,1,8,4]
,[6,2,7,1,3,5,8,4],[3,7,2,8,6,4,1,5],[6,3,7,2,4,8,1,5],[4,2,7,3,6,8,1,5],[7,1,3,8,6,4,2,5]
,[1,6,8,3,7,4,2,5],[3,8,4,7,1,6,2,5],[6,3,7,4,1,8,2,5],[7,4,2,8,6,1,3,5],[4,6,8,2,7,1,3,5]
,[2,6,1,7,4,8,3,5],[2,4,6,8,3,1,7,5],[3,6,8,2,4,1,7,5],[6,3,1,8,4,2,7,5],[8,4,1,3,6,2,7,5]
,[4,8,1,3,6,2,7,5],[2,6,8,3,1,4,7,5],[7,2,6,3,1,4,8,5],[3,6,2,7,1,4,8,5],[4,7,3,8,2,5,1,6]
,[4,8,5,3,1,7,2,6],[3,5,8,4,1,7,2,6],[4,2,8,5,7,1,3,6],[5,7,2,4,8,1,3,6],[7,4,2,5,8,1,3,6]
,[8,2,4,1,7,5,3,6],[7,2,4,1,8,5,3,6],[5,1,8,4,2,7,3,6],[4,1,5,8,2,7,3,6],[5,2,8,1,4,7,3,6]
,[3,7,2,8,5,1,4,6],[3,1,7,5,8,2,4,6],[8,2,5,3,1,7,4,6],[3,5,2,8,1,7,4,6],[3,5,7,1,4,2,8,6]
,[5,2,4,6,8,3,1,7],[6,3,5,8,1,4,2,7],[5,8,4,1,3,6,2,7],[4,2,5,8,6,1,3,7],[4,6,1,5,2,8,3,7]
,[6,3,1,8,5,2,4,7],[5,3,1,6,8,2,4,7],[4,2,8,6,1,3,5,7],[6,3,5,7,1,4,2,8],[6,4,7,1,3,5,2,8]
,[4,7,5,2,6,1,3,8],[5,7,2,6,3,1,4,8]]

> length (queens 8)
92
```

```
assert(
  queens(8)
  ==
  List(
    List(4,2,7,3,6,8,5,1),List(5,2,4,7,3,8,6,1),List(3,5,2,8,6,4,7,1),List(3,6,4,2,8,5,7,1),List(5,7,1,3,8,6,4,2),
    List(4,6,8,3,1,7,5,2),List(3,6,8,1,4,7,5,2),List(5,3,8,4,7,1,6,2),List(5,7,4,1,3,8,6,2),List(4,1,5,8,6,3,7,2),
    List(3,6,4,1,8,5,7,2),List(4,7,5,3,1,6,8,2),List(6,4,2,8,5,7,1,3),List(6,4,7,1,8,2,5,3),List(1,7,4,6,8,2,5,3),
    List(6,8,2,4,1,7,5,3),List(6,2,7,1,4,8,5,3),List(4,7,1,8,5,2,6,3),List(5,8,4,1,7,2,6,3),List(4,8,1,5,7,2,6,3),
    List(2,7,5,8,1,4,6,3),List(1,7,5,8,2,4,6,3),List(2,5,7,4,1,8,6,3),List(4,2,7,5,1,8,6,3),List(5,7,1,4,2,8,6,3),
    List(6,4,1,5,8,2,7,3),List(5,1,4,6,8,2,7,3),List(5,2,6,1,7,4,8,3),List(6,3,7,2,8,5,1,4),List(2,7,3,6,8,5,1,4),
    List(7,3,1,6,8,5,2,4),List(5,1,8,6,3,7,2,4),List(1,5,8,6,3,7,2,4),List(3,6,8,1,5,7,2,4),List(6,3,1,7,5,8,2,4),
    List(7,5,3,1,6,8,2,4),List(7,3,8,2,5,1,6,4),List(5,3,1,7,2,8,6,4),List(2,5,7,1,3,8,6,4),List(3,6,2,5,8,1,7,4),
    List(6,1,5,2,8,3,7,4),List(8,3,1,6,2,5,7,4),List(2,8,6,1,3,5,7,4),List(5,7,2,6,3,1,8,4),List(3,6,2,7,5,1,8,4),
    List(6,2,7,1,3,5,8,4),List(3,7,2,8,6,4,1,5),List(6,3,7,2,4,8,1,5),List(4,2,7,3,6,8,1,5),List(7,1,3,8,6,4,2,5),
    List(1,6,8,3,7,4,2,5),List(3,8,4,7,1,6,2,5),List(6,3,7,4,1,8,2,5),List(7,4,2,8,6,1,3,5),List(4,6,8,2,7,1,3,5),
    List(2,6,1,7,4,8,3,5),List(2,4,6,8,3,1,7,5),List(3,6,8,2,4,1,7,5),List(6,3,1,8,4,2,7,5),List(8,4,1,3,6,2,7,5),
    List(4,8,1,3,6,2,7,5),List(2,6,8,3,1,4,7,5),List(7,2,6,3,1,4,8,5),List(3,6,2,7,1,4,8,5),List(4,7,3,8,2,5,1,6),
    List(4,8,5,3,1,7,2,6),List(3,5,8,4,1,7,2,6),List(4,2,8,5,7,1,3,6),List(5,7,2,4,8,1,3,6),List(7,4,2,5,8,1,3,6),
    List(8,2,4,1,7,5,3,6),List(7,2,4,1,8,5,3,6),List(5,1,8,4,2,7,3,6),List(4,1,5,8,2,7,3,6),List(5,2,8,1,4,7,3,6),
    List(3,7,2,8,5,1,4,6),List(3,1,7,5,8,2,4,6),List(8,2,5,3,1,7,4,6),List(3,5,2,8,1,7,4,6),List(3,5,7,1,4,2,8,6),
    List(5,2,4,6,8,3,1,7),List(6,3,5,8,1,4,2,7),List(5,8,4,1,3,6,2,7),List(4,2,5,8,6,1,3,7),List(4,6,1,5,2,8,3,7),
    List(6,3,1,8,5,2,4,7),List(5,3,1,6,8,2,4,7),List(4,2,8,6,1,3,5,7),List(6,3,5,7,1,4,2,8),List(6,4,7,1,3,5,2,8),
    List(4,7,5,2,6,1,3,8),List(5,7,2,6,3,1,4,8))
)

assert(queens(8).size == 92)
```

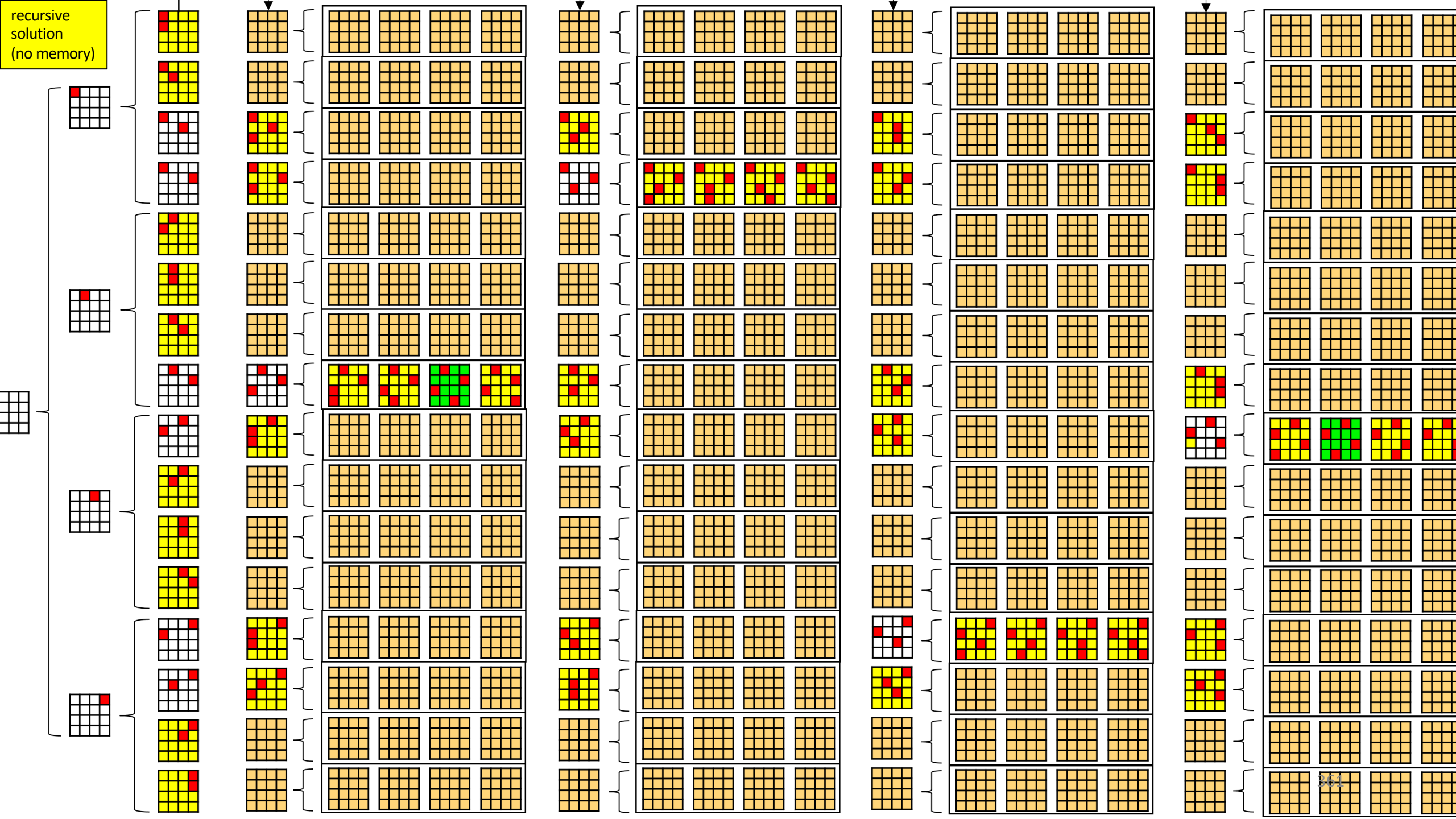
The **recursive algorithm** has no ‘memory’ of the columns in which queens have already been placed, so it will generate and test a **permutation** for safety, even if it contains repetitions, like the following one.

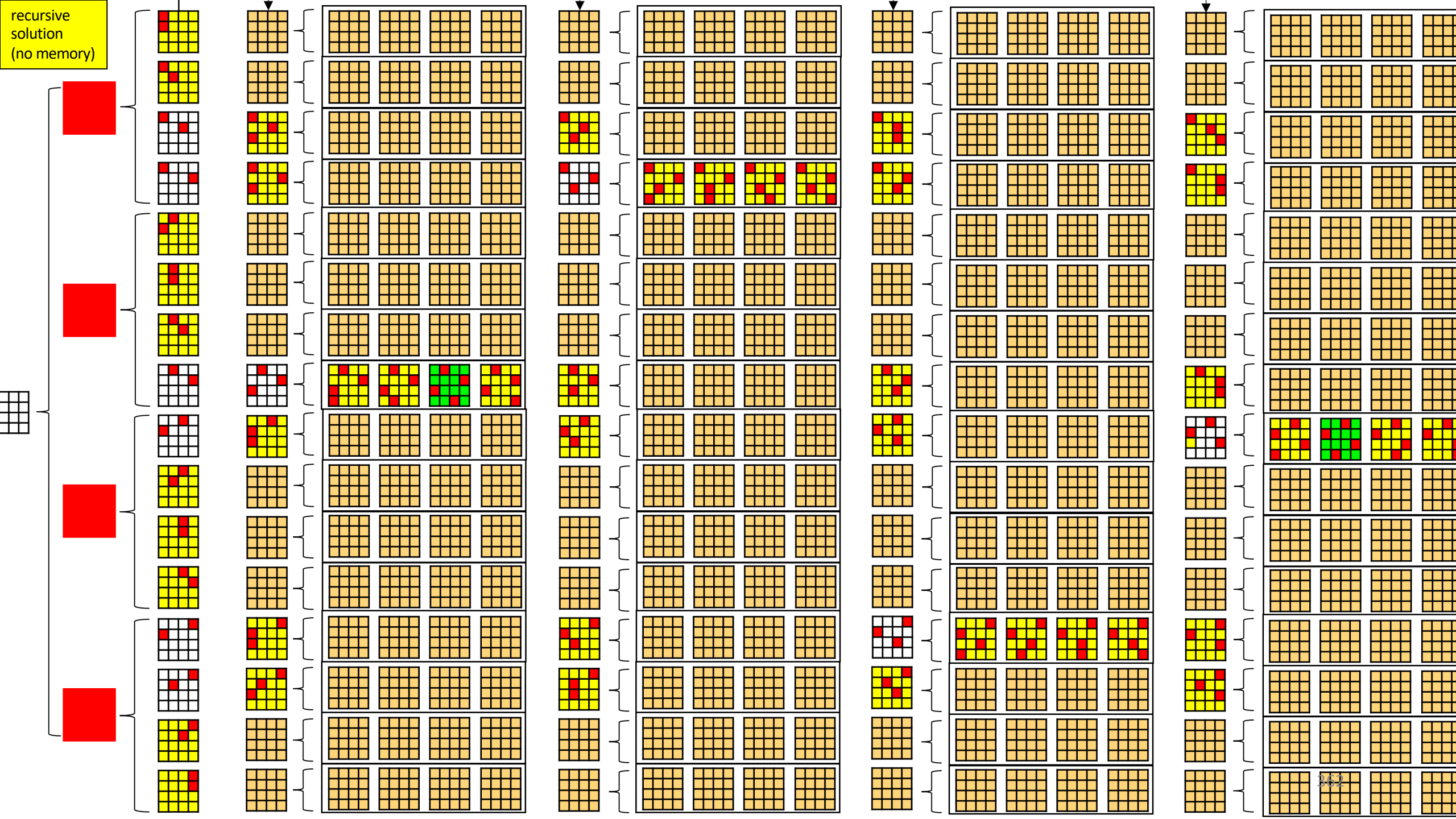


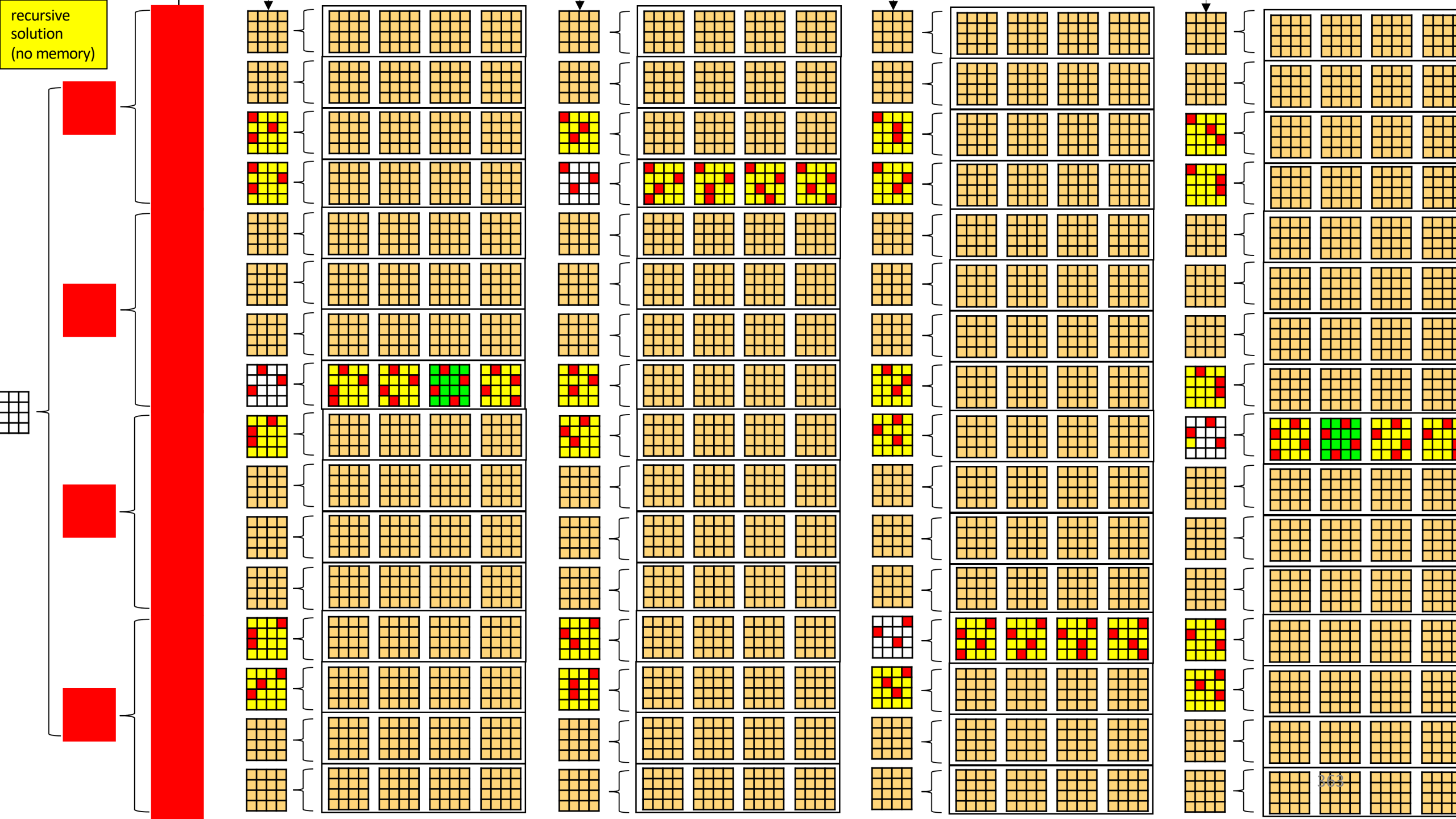
The **iterative algorithm** does have ‘memory’, in that **permutations** with **repetition**, like the one above, are not even considered as candidate solutions, i.e. it ‘remembers’ where it has already placed previous queens.

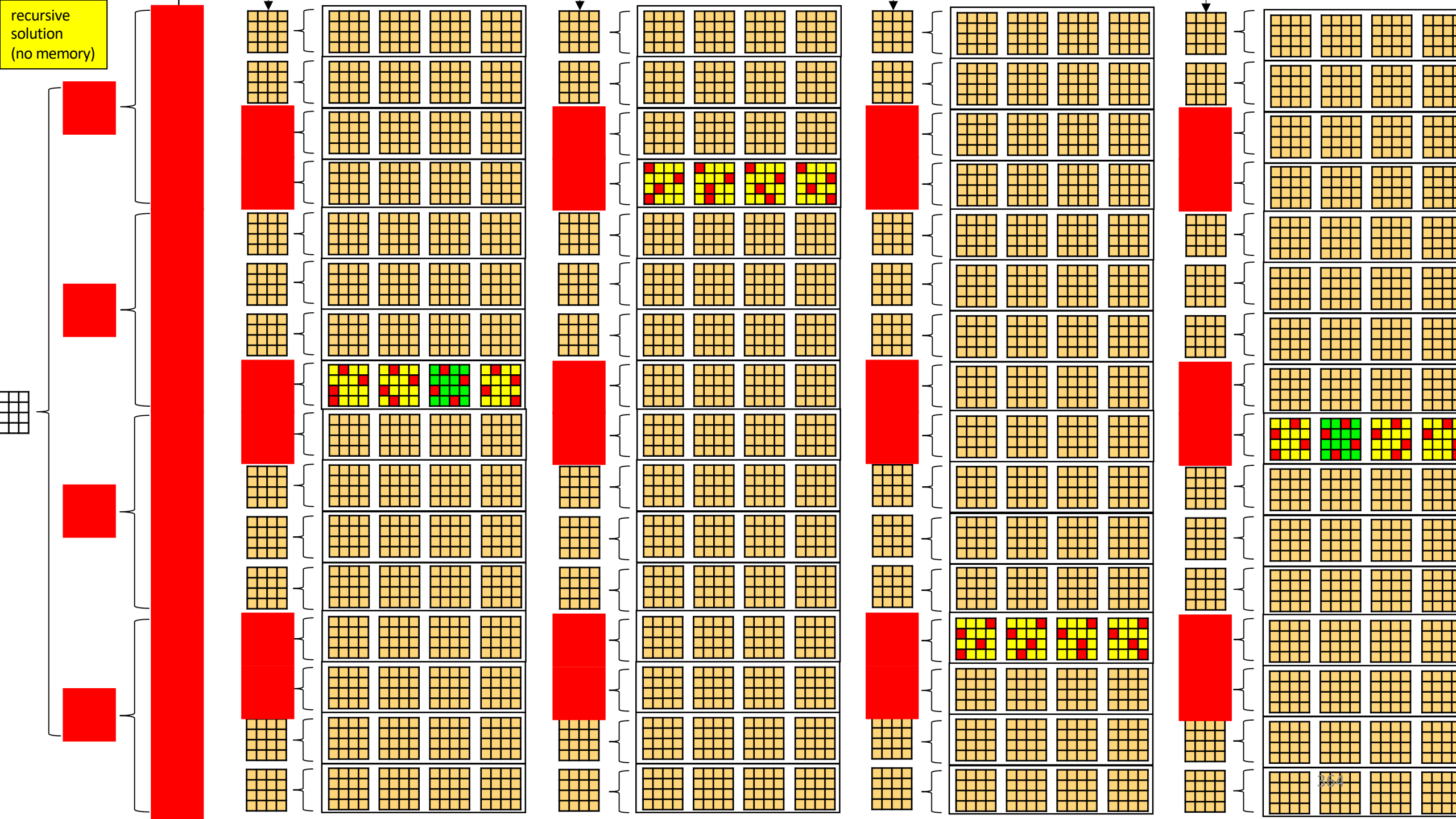
Without the **isSafe** function, the recursive algorithm would generate $\mathbf{4^4}$ = **256** candidate solution boards.

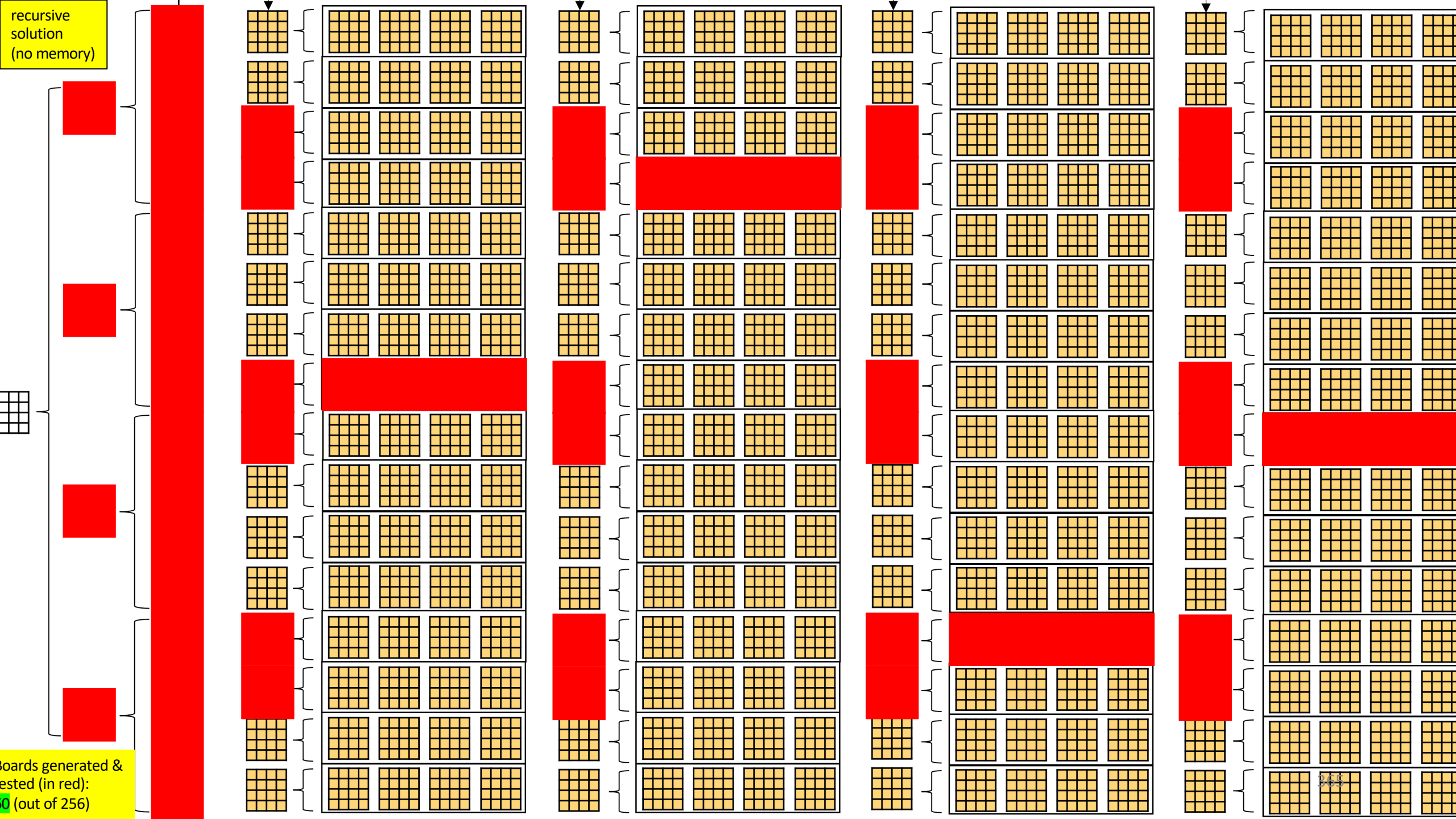
With the **isSafe** function, the recursive algorithm generates **60** boards.


recursive
solution
(no memory)

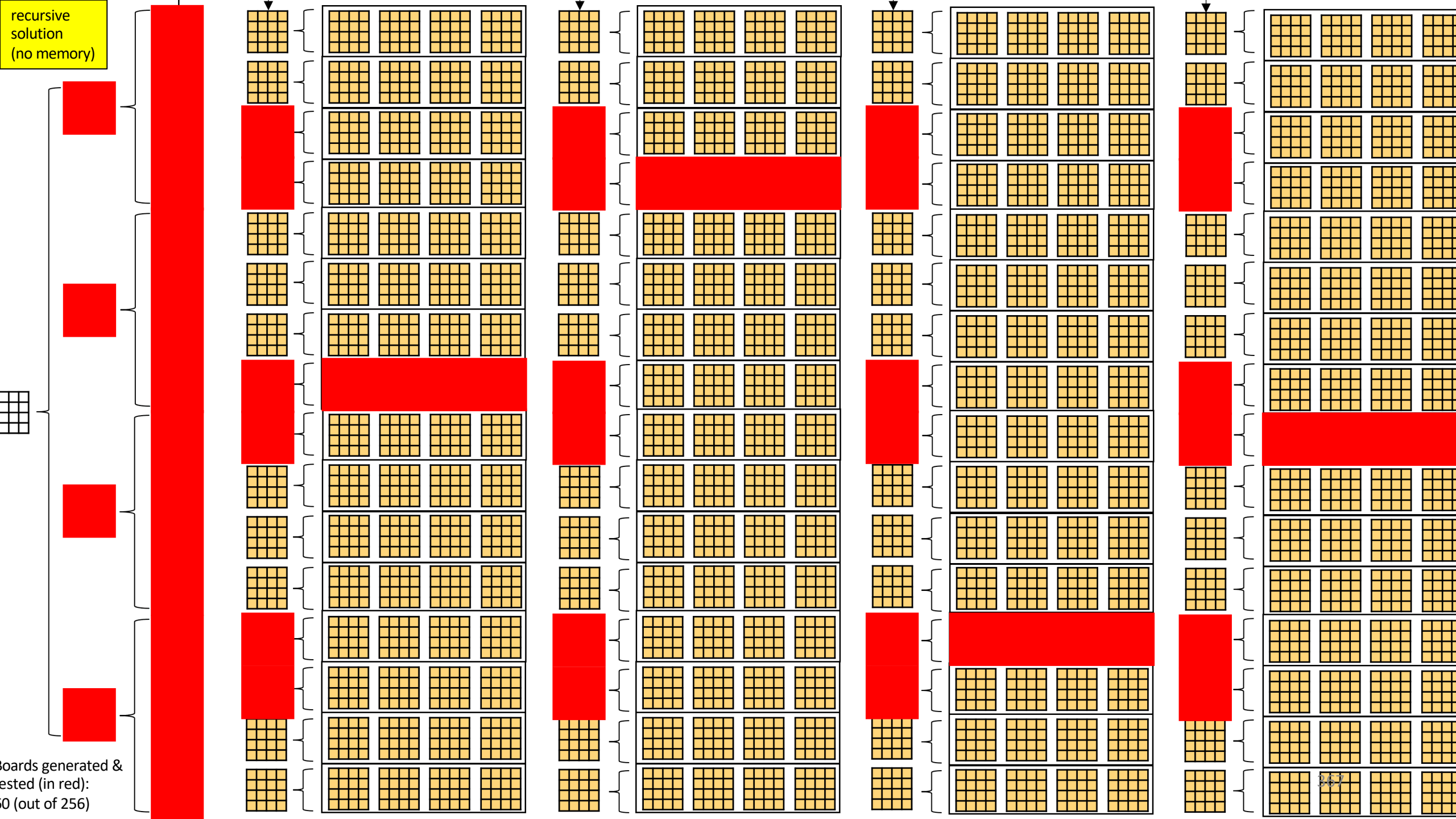

recursive
solution
(no memory)

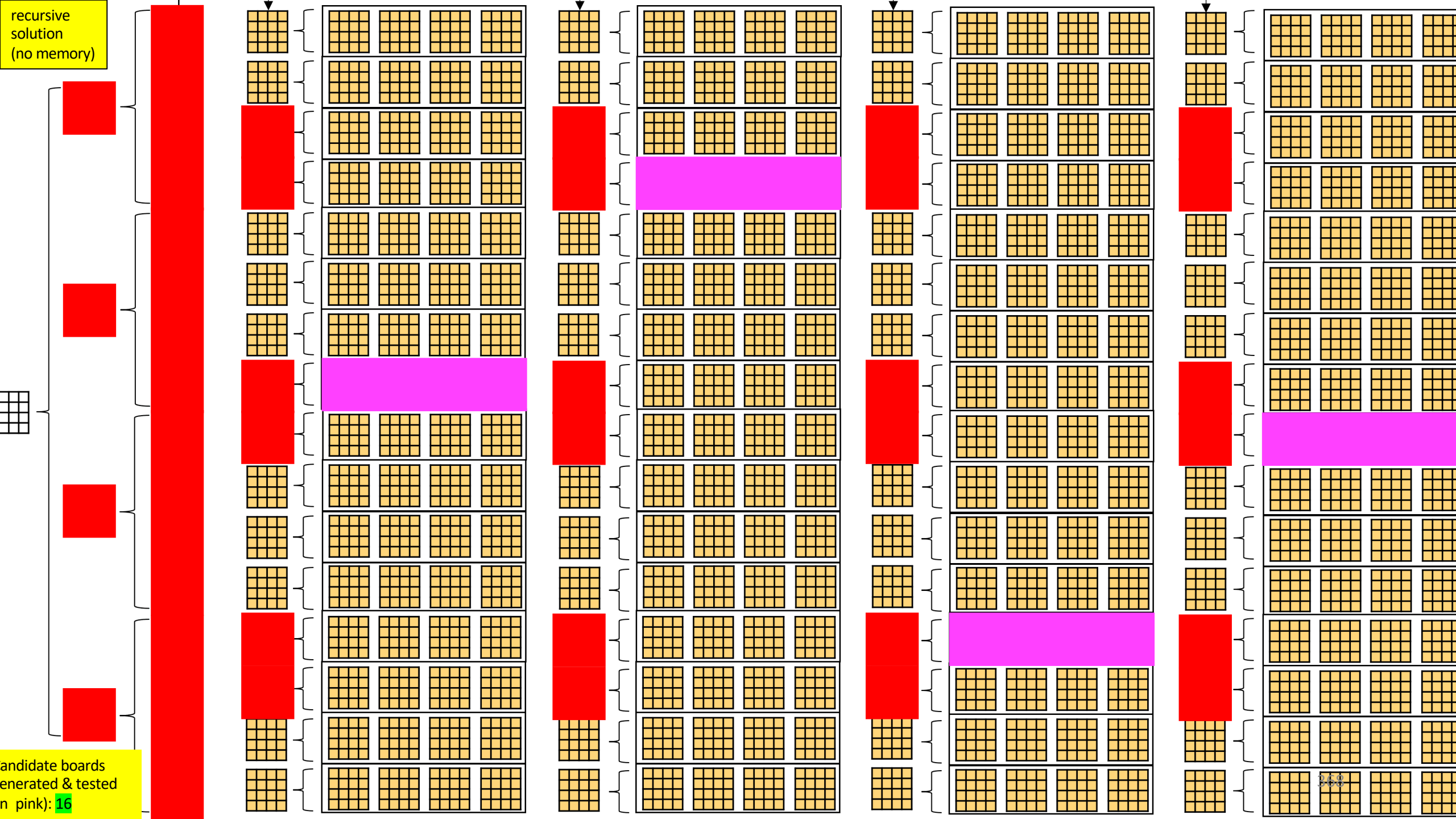

recursive
solution
(no memory)


recursive
solution
(no memory)


recursive
solution
(no memory)
Boards generated &
ested (in red):
0 (out of 256)
365

Of the **60** boards generated, **16** are candidate solution boards.

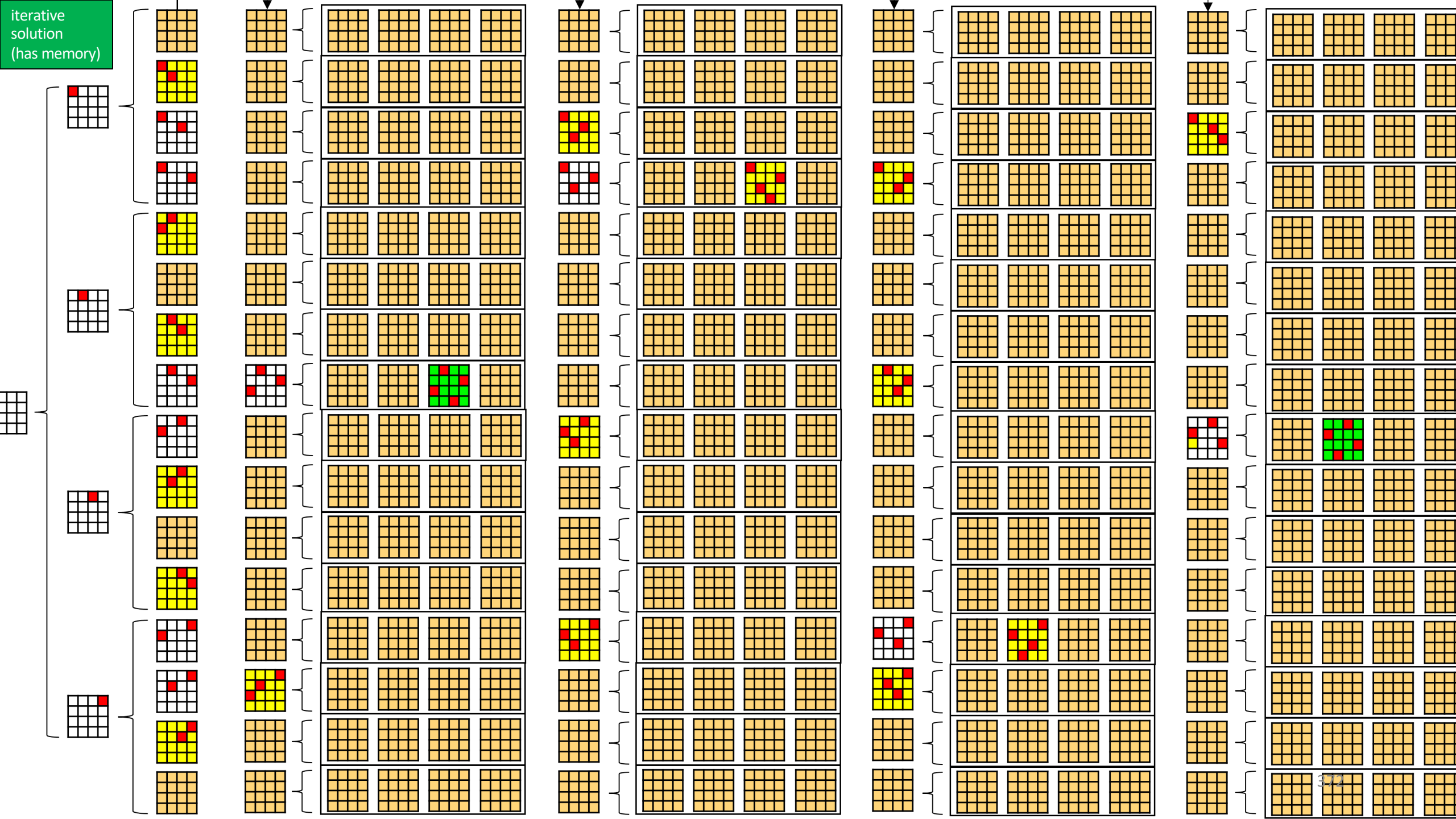

recursive
solution
(no memory)
Boards generated &
ested (in red):
60 (out of 256)
367

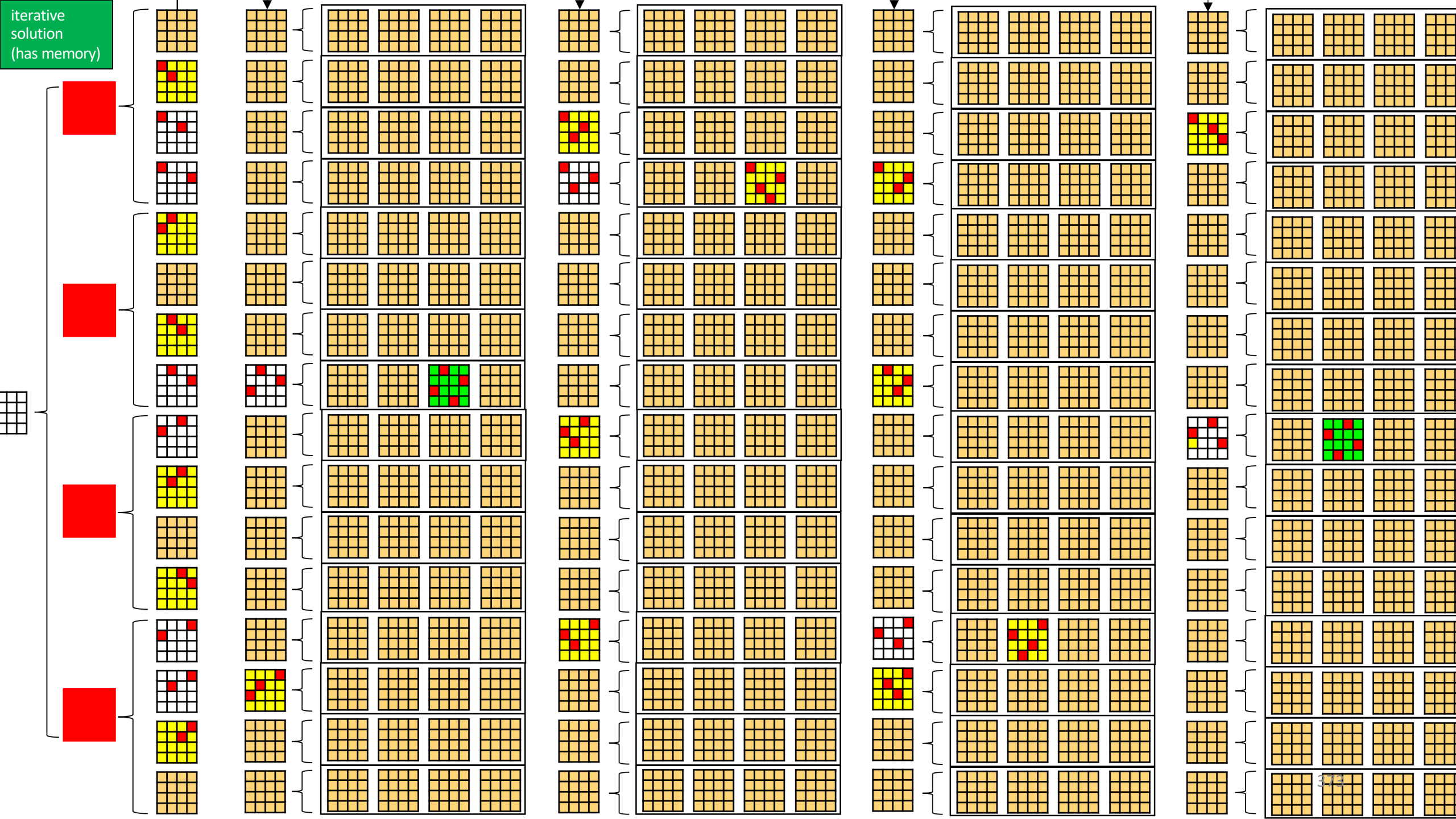

recursive
solution
(no memory)
andidate boards
enerated & tested
n pink): 16

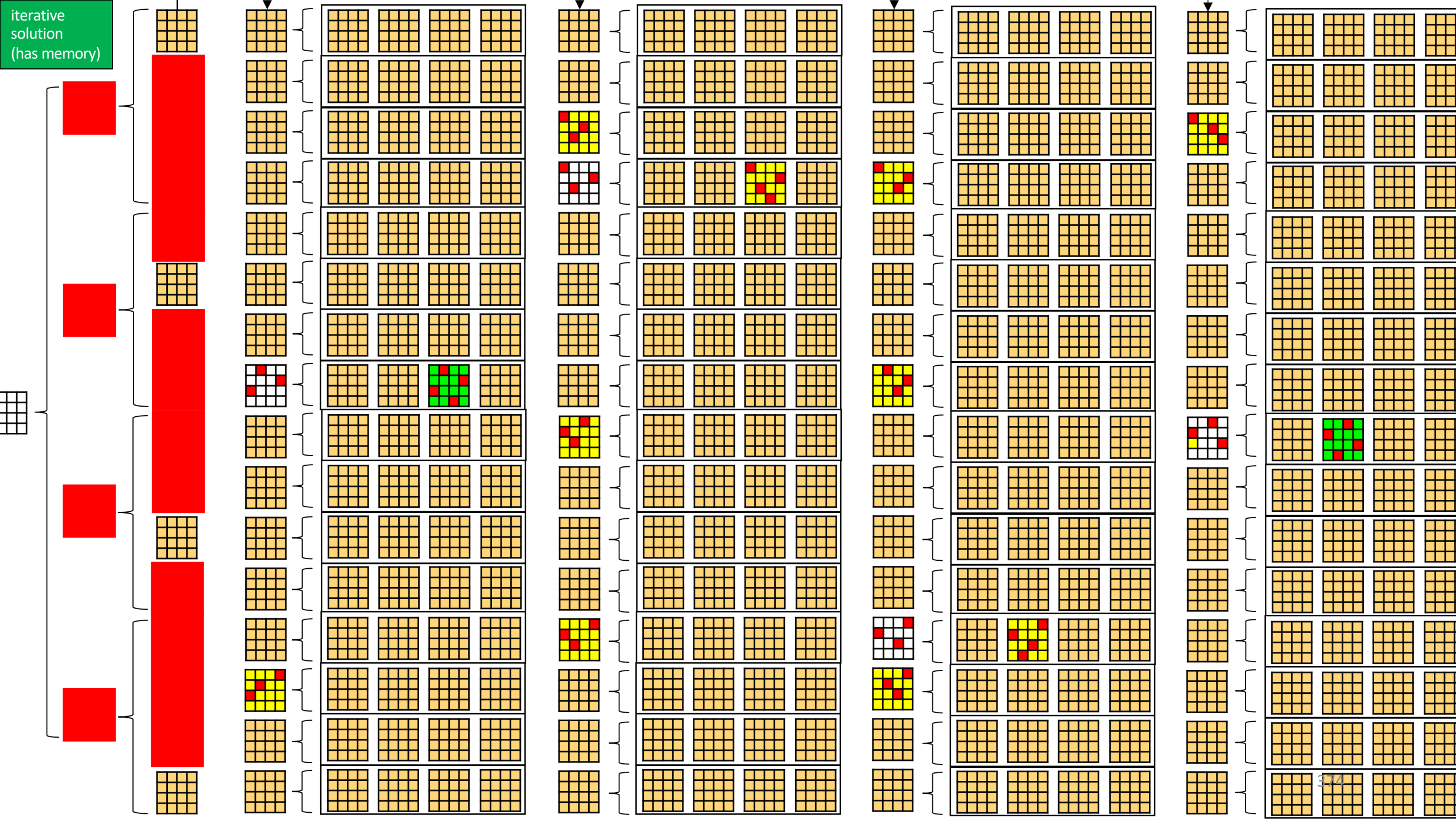
Lets compare that with the **iterative** solution

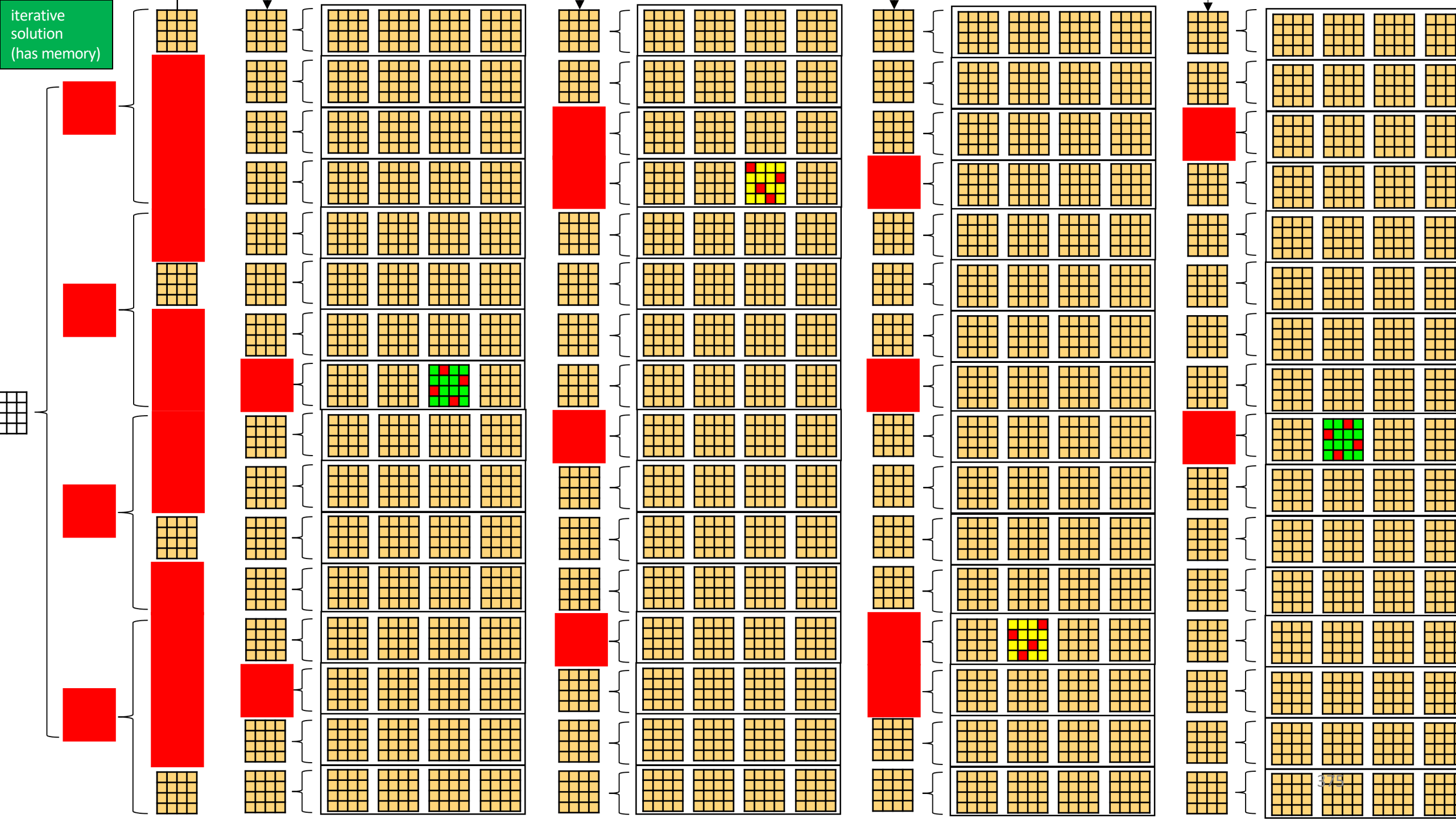
Without the **isSafe** function, the iterative algorithm would generate **4!** = **24** candidate solution boards.

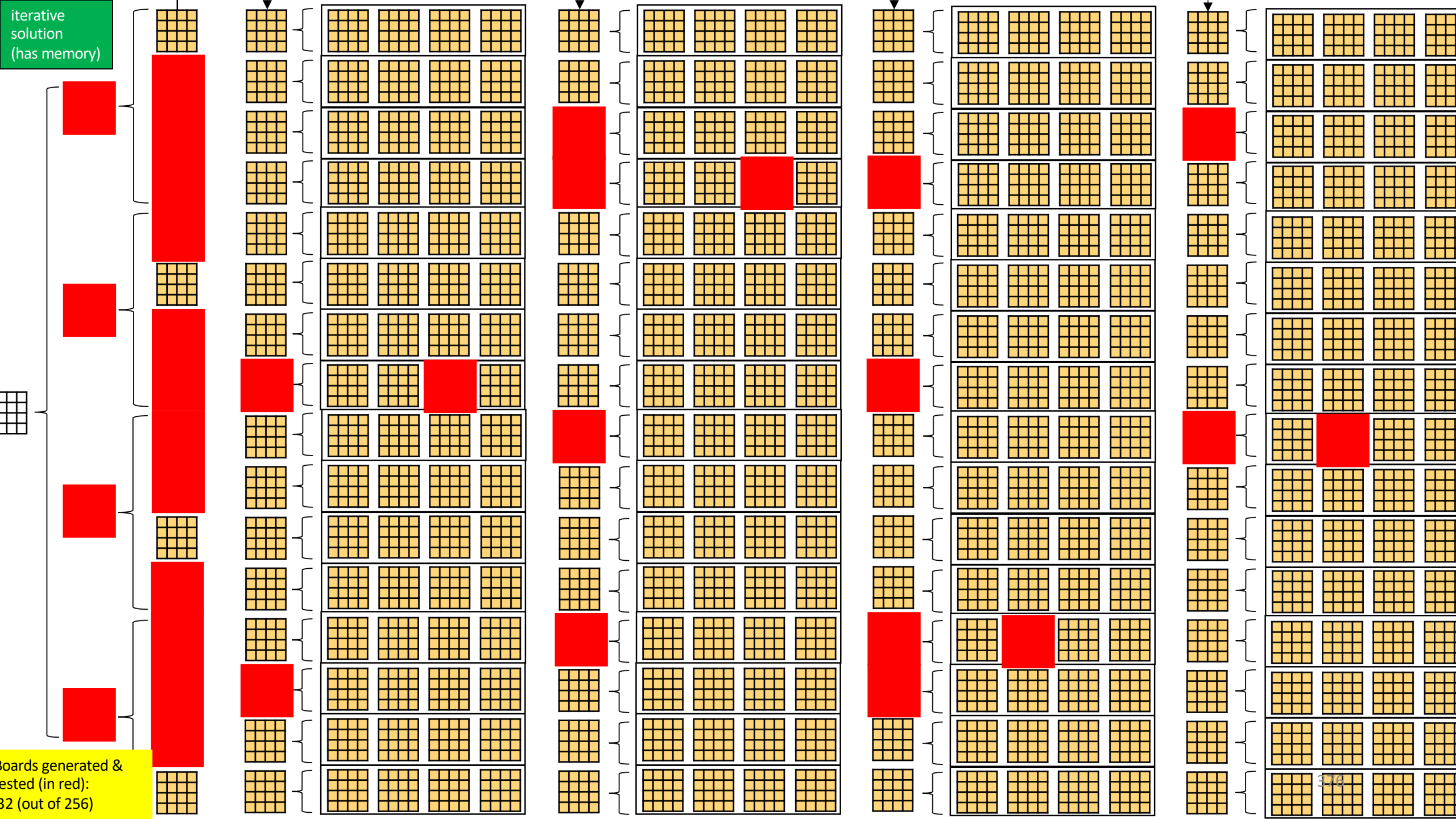
With the **isSafe** function, the iterative algorithm generates **32** boards.


iterative
solution
(has memory)

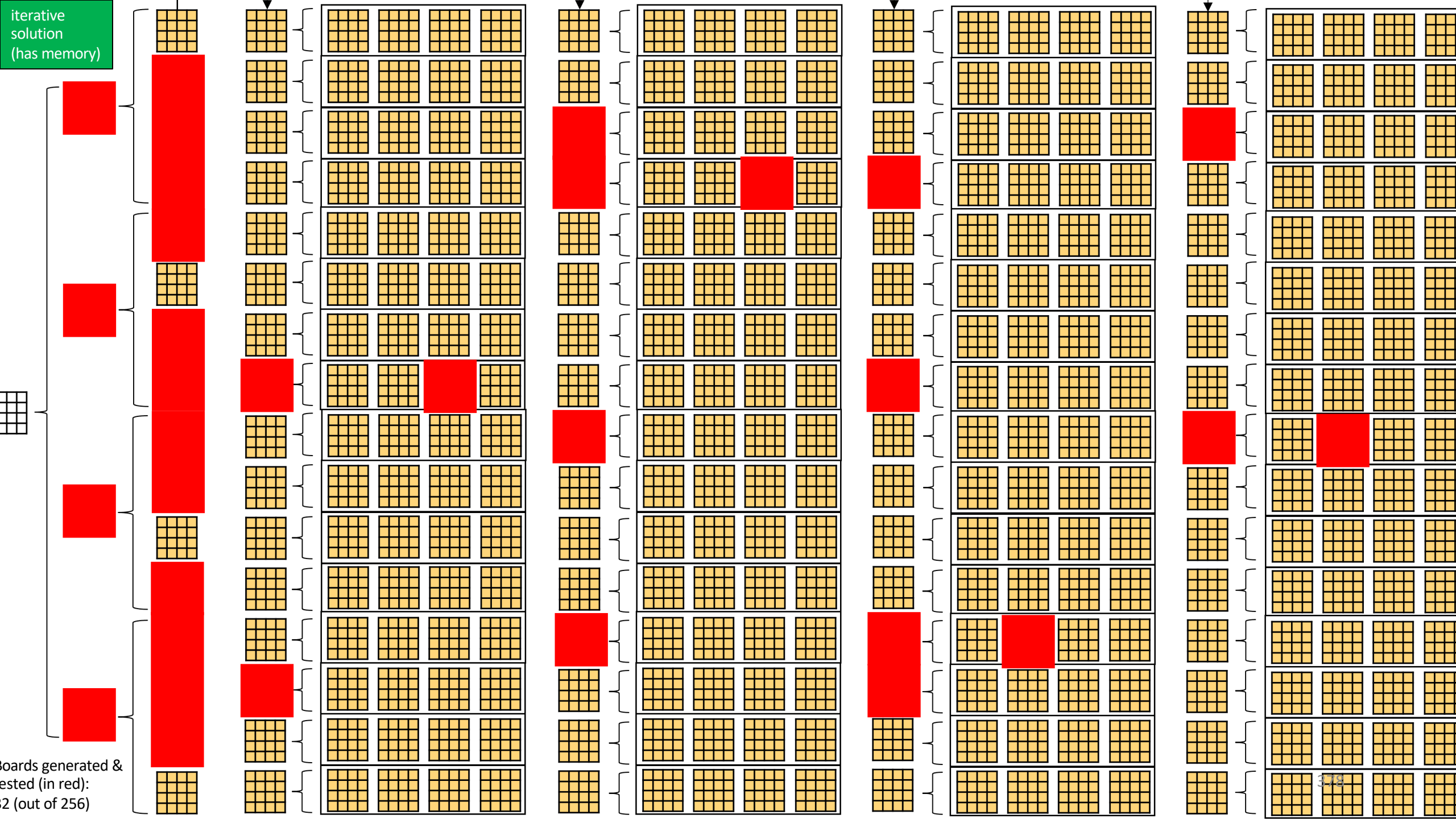

iterative
solution
(has memory)

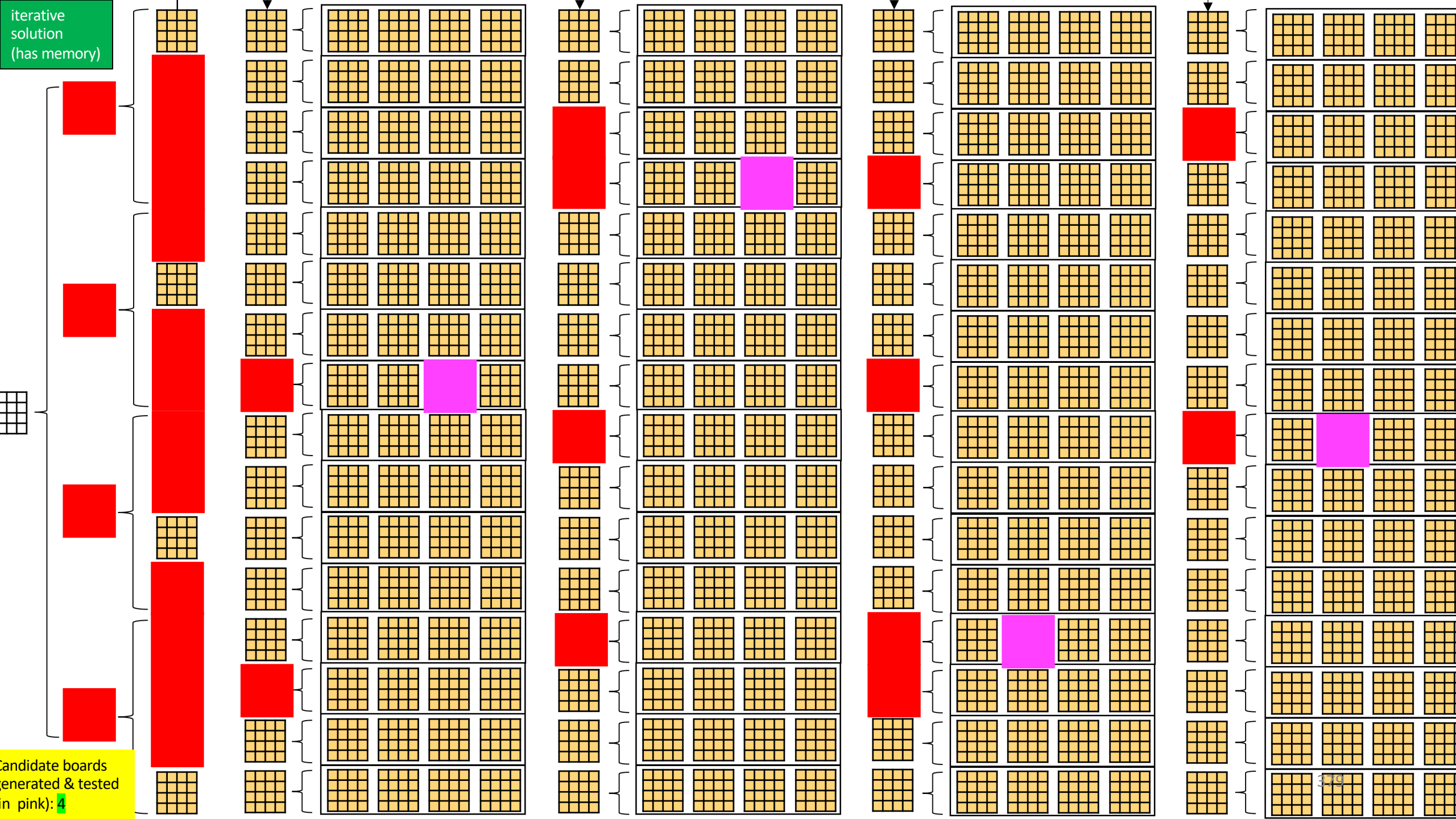

iterative
solution
(has memory)


iterative
solution
(has memory)


iterative
solution
(has memory)
Boards generated &
ested (in red):
32 (out of 256)

Of the **32** boards generated, **4** are candidate solution boards.


iterative
solution
(has memory)
Boards generated &
ested (in red):
2 (out of 256)


iterative
solution
(has memory)
Candidate boards
generated & tested
in pink): 4

N=4; 2 solutions

Algorithm	**without isSafe function**	**with isSafe function**	
	Boards Generated (All of them candidate solutions)	Boards Generated	Candidate Boards Generated
Recursive	**256 (4^4)**	**60**	**16**
Iterative	**24 (4!)**	**32**	**4**

N=5; 10 solutions

Algorithm	without isSafe function	with isSafe function	
	Boards Generated (All of them candidate solutions)	Boards Generated	Candidate Boards Generated
Recursive	3,125 (5^5)	220	60
Iterative	120 (5!)	101	12

N=6; 4 solutions

Algorithm	without isSafe function	with isSafe function	
	Boards Generated (All of them candidate solutions)	Boards Generated	Candidate Boards Generated
Recursive	46,656 (6^6)	893	240
Iterative	720 (6!)	356	40

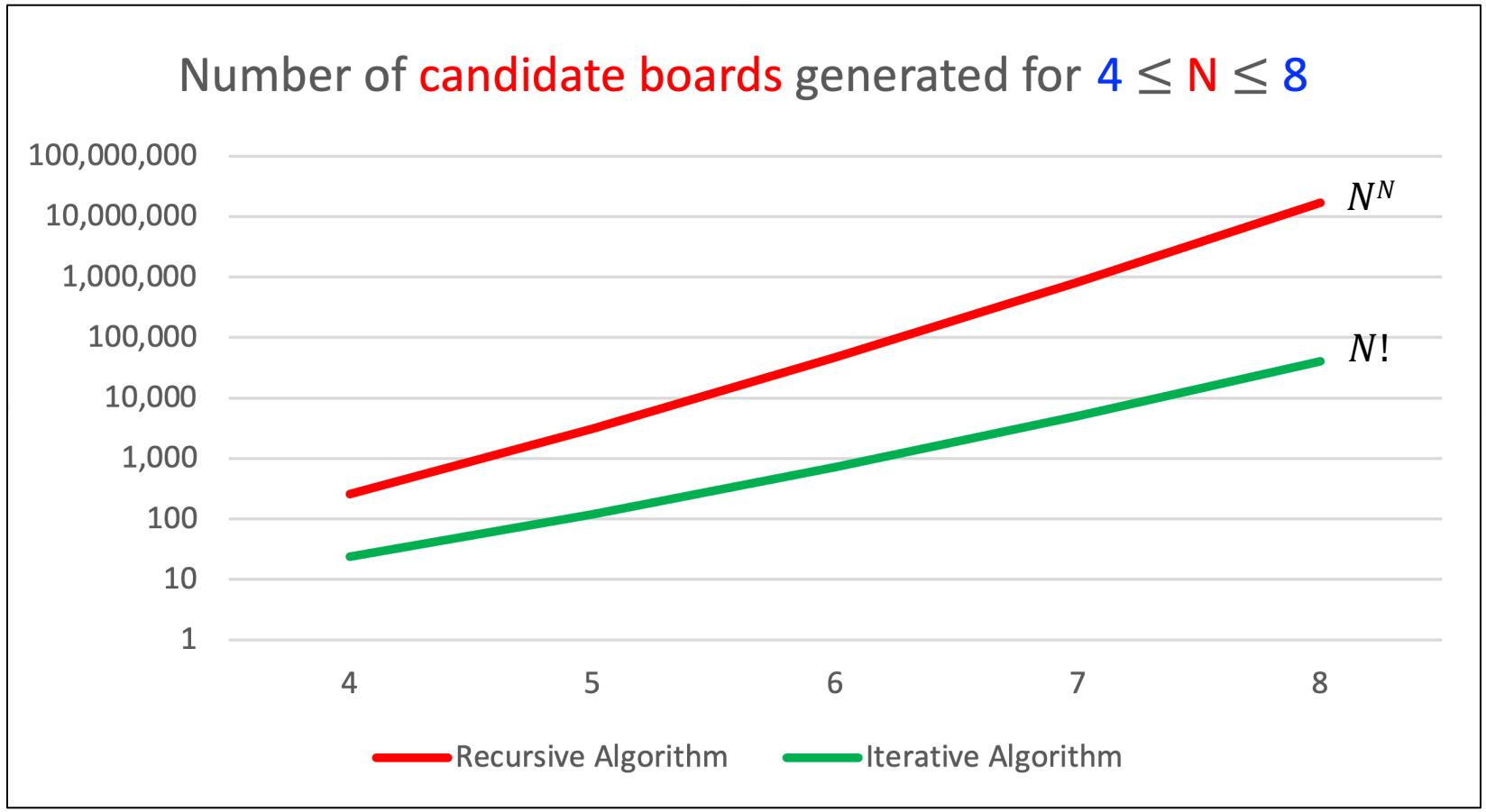
N=7; 40 solutions

Algorithm	without isSafe function	with isSafe function	
	Boards Generated (All of them candidate solutions)	Boards Generated	Candidate Boards Generated
Recursive	823,543 (7^7)	3,581	657
Iterative	5,040 (7!)	1,344	93

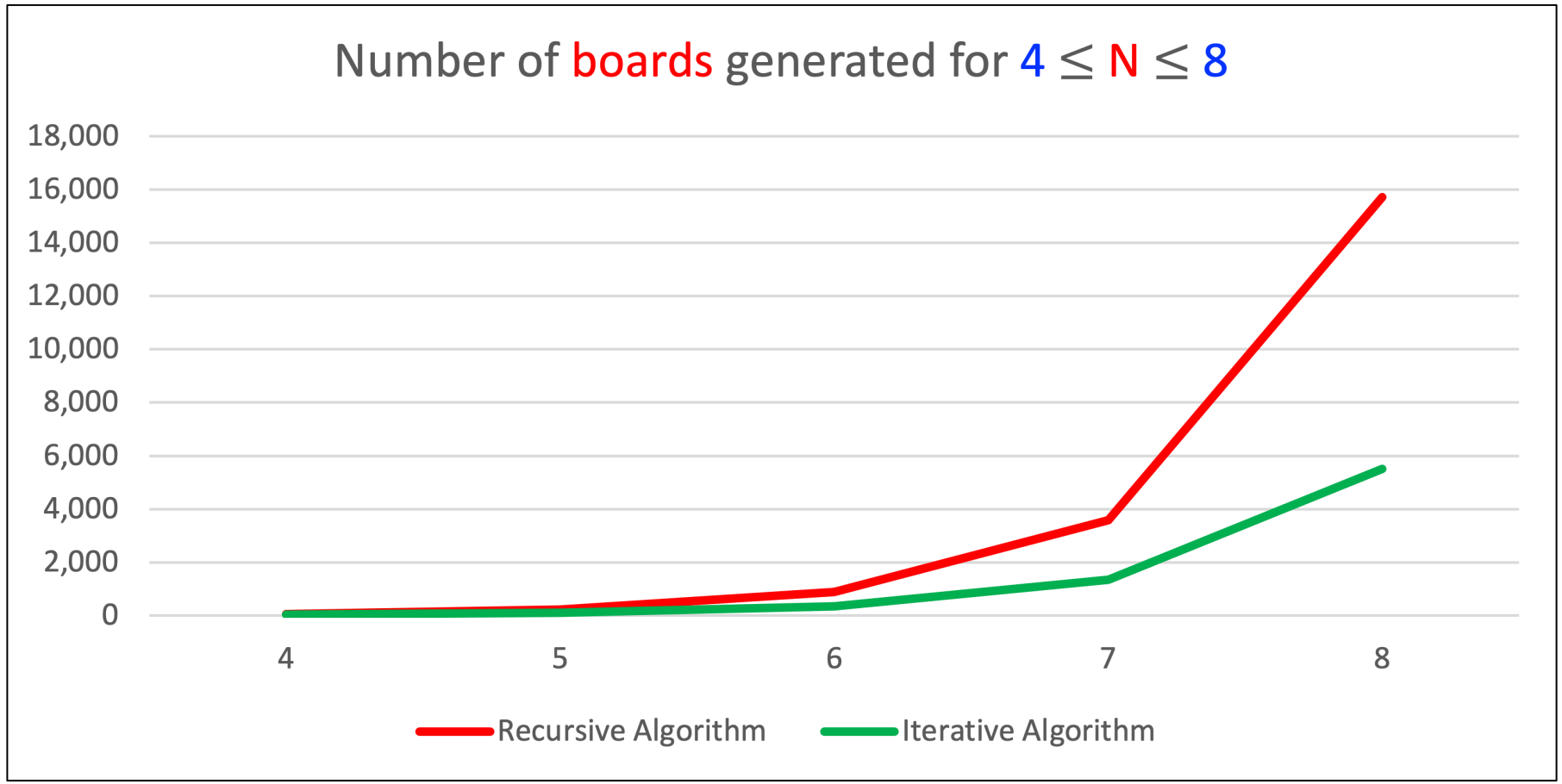
N=8; 92 solutions

Algorithm	without isSafe function	with isSafe function	
	Boards Generated (All of them candidate solutions)	Boards Generated	Candidate Boards Generated
Recursive	16,777,216 (8^8)	15,718	2,495
Iterative	40,320 (8!)	5,506	312

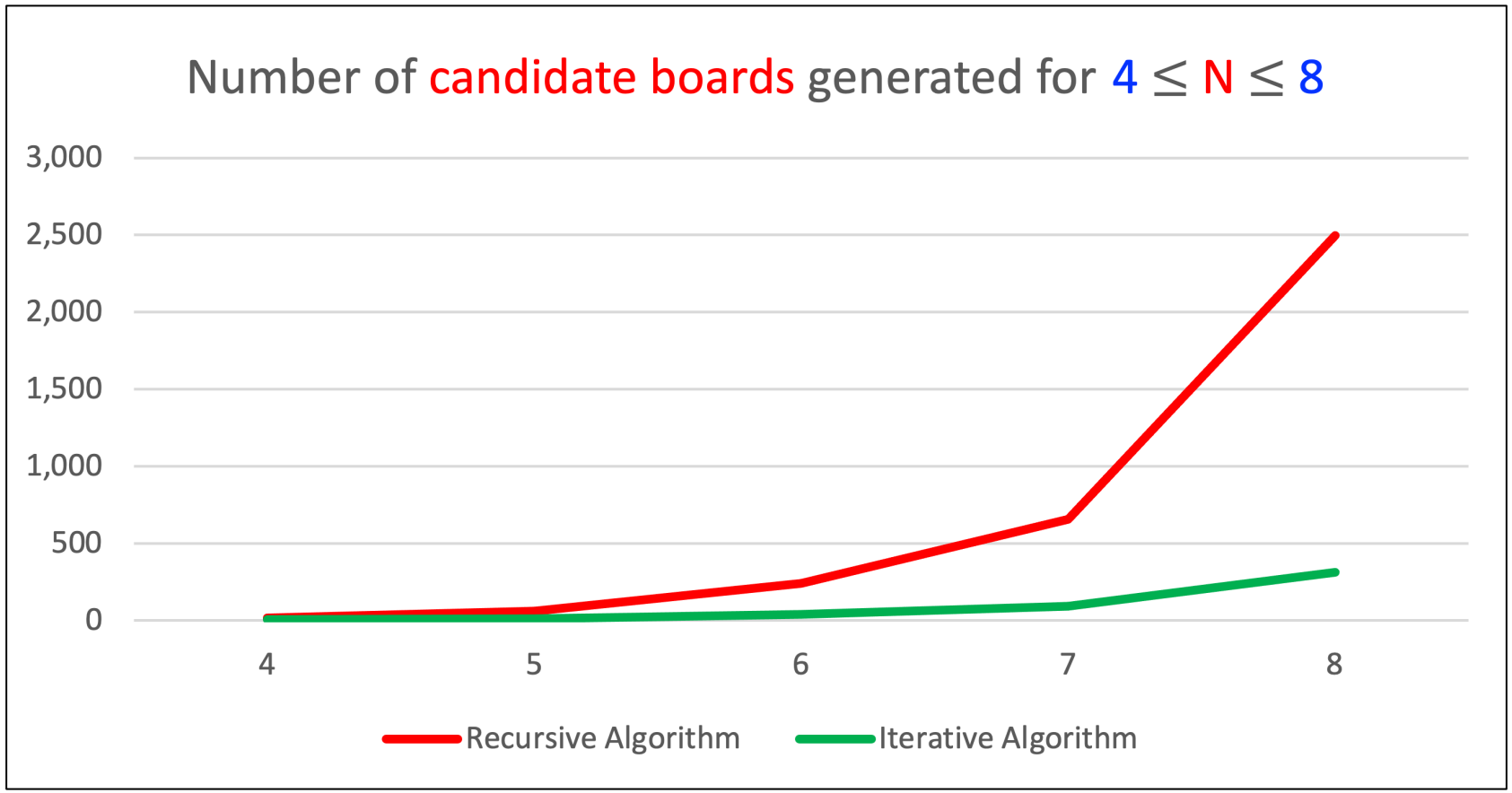
Without using **isSafe** function



Using **isSafe** function



Using **isSafe** function




That's all Folks!