15-150

Principles of Functional Programming

Lecture 13

February 27, 2025

Michael Erdmann

Exceptions, n-Queens, & more success continuations

Exceptions

Declaring

Raising

Handling

Exceptions are useful for signaling errors, including violations of invariants.

Exceptions can also be useful for control flow, analagous to continuations.

Declaring

```
exception Silly
```

↑ keyword

↑ an exception constructor

This line of code declares a new exception (constructor) called Silly.

(If Silly already exists, this new Silly will shadow the earlier Silly.)

Raising

raise Silly

↑ keyword

↑ an existing exception

Example:

if 3=4 then raise Silly else 9

- What is the type of this expression?
- What is the value to which the expression reduces?

Types

```
Silly : exn
```

↑ the type for exceptions

```
raise Silly : 'a
```

So `if 3=4 then raise Silly else 9 : int`

instantiate 'a to be int

and the expression reduces to value 9.

What about the type & value of

```
if 3=3 then raise Silly else 0
```

?

The type is int.

The expression does not reduce to a value. Instead SML will print

uncaught exception Silly.

We will discuss handling exceptions shortly.

Declaring exceptions that take arguments

```
exception Rdiv of real
```

- `Rdiv`: exception constructor
- `real`: type of argument

Now

```
Rdiv : real -> exn
Rdiv (2.1) : exn
raise Rdiv (2.1) : 'a
```

Handling

General form of an exception handler for expression e:

$$e \ \underline{\text{handle}} \ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 \mid \vdots \mid p_n \Rightarrow e_n$$

$e, e_1, \ldots, e_n$ are expressions; must have the same type.

$p_1, \ldots, p_n$ are patterns; must match exceptions.

If e reduces to a value v, that value is returned.

If e instead raises an **uncaught** exception E, the handler will try to match E against the patterns $p_1, \ldots, p_n$ (in sequential order). If E matches p_i, then SML will evaluate e_i. If no pattern matches, E remains uncaught.

```
fun  f(x,0) = raise Rdiv (x * x)
   | f (x,n) = if n<0 then raise Silly
               else x/(real n)

fun g(x,n) = f(x,n) handle Silly ⇒ 0.0
                         | Rdiv(v) ⇒ v
```

What are the types of f & g ?

Both have type

real * int → real

```
fun  f(x,0) = raise Rdiv (x * x)
   | f (x,n) = if n<0 then raise Silly
               else x/(real n)

fun g(x,n) = f(x,n) handle Silly ⇒ 0.0
                         | Rdiv(v) ⇒ v
```

What are the values of:

$g(3.0, 0) \hookrightarrow 9.0$

$g(3.0, 2) \hookrightarrow 1.5$

$g(3.0, \sim 1) \hookrightarrow 0.0$

```
fun f(x,0) = raise Rdiv (x * x)
  | f (x,n) = if n<0 then raise Silly
              else x/(real n)

fun g(x,n) = f(x,n) handle Silly ⇒ 0.0
```

Suppose g does not handle Rdiv.

What now are the values of:

g (3.0, 0) **no** value; uncaught exception Rdiv

g (3.0, 2) ↪ 1.5

g (3.0, ~1) ↪ 0.0

n-Queens

Place n queens on a square $n \times n$ board without any two queens threatening each other.

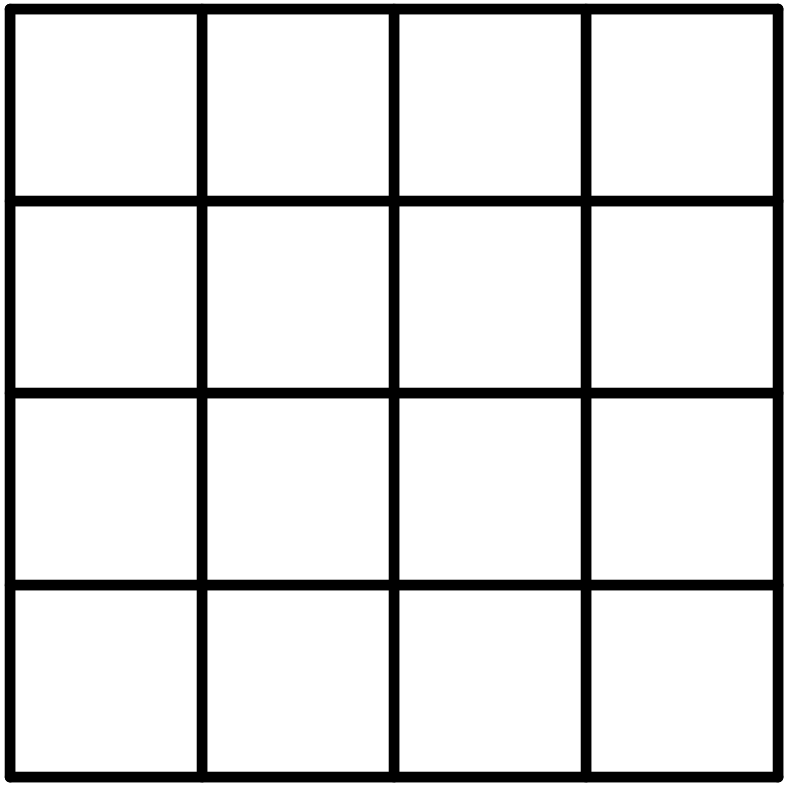
(A queen threatens all locations in the same column, in the same row, and on lines with slope ± 1 that pass through the queen's position.)

Three Implementations

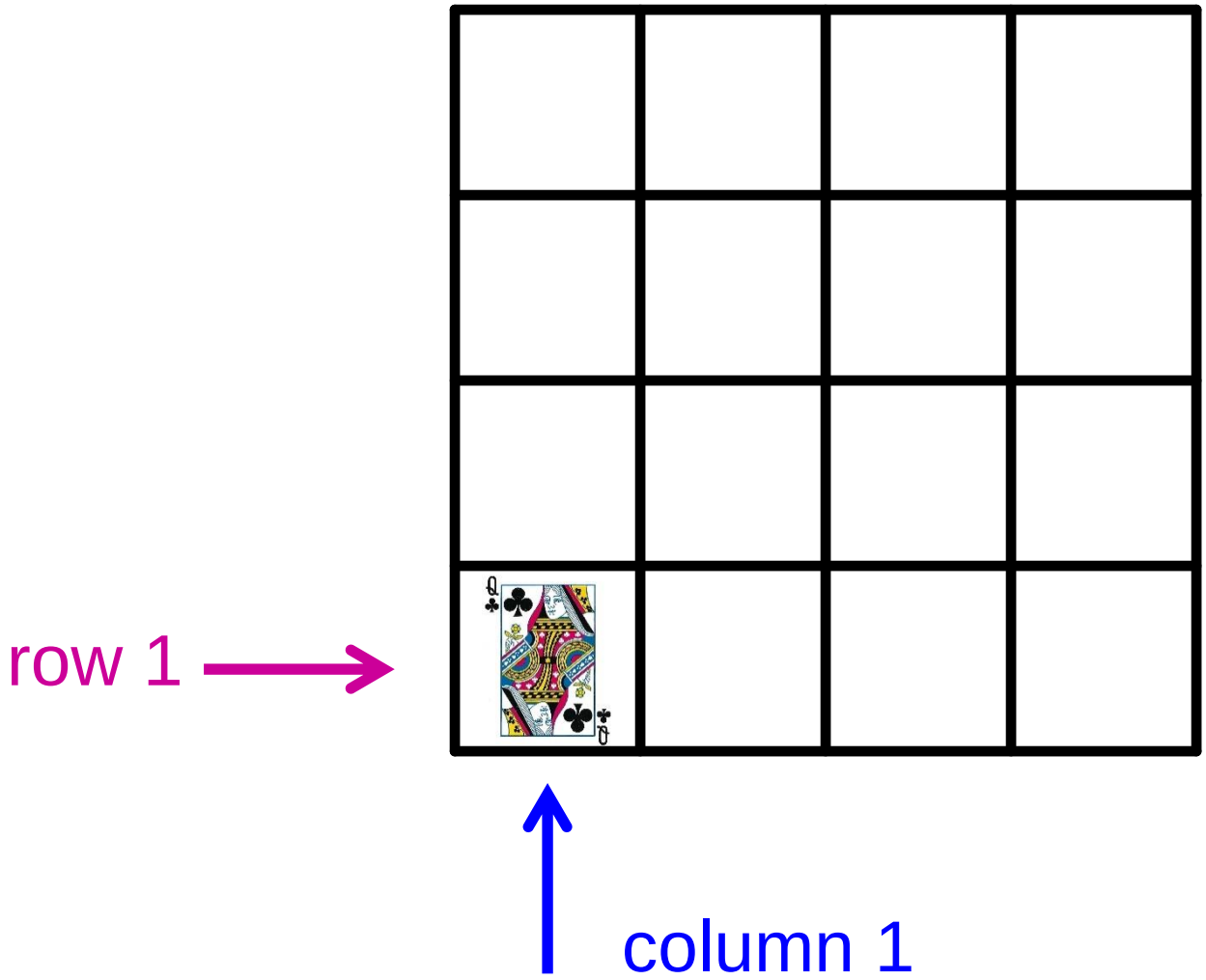
- Exceptions
- Options
- Continuations

We will use these programming styles for search

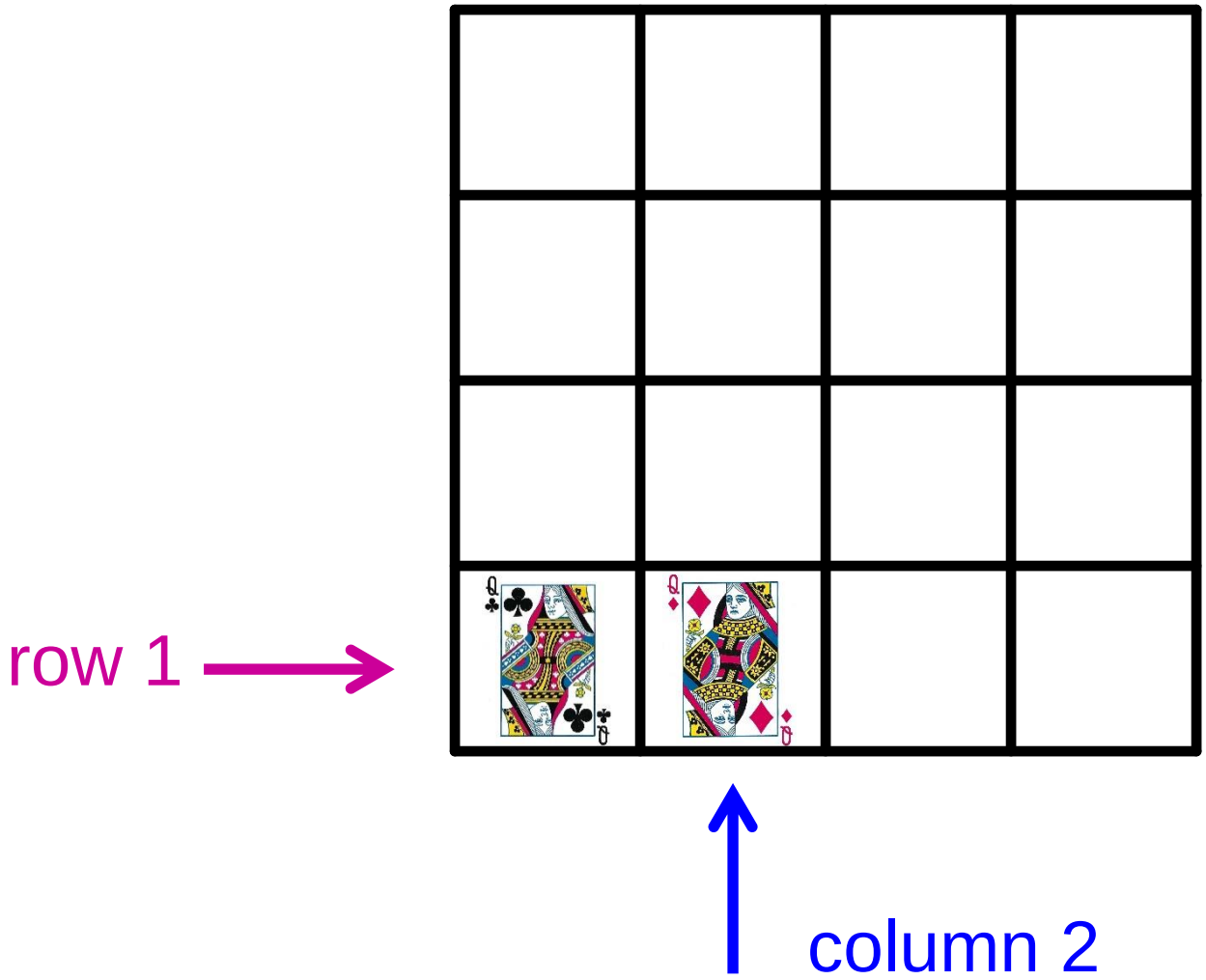
Example: Place 4 queens on a 4x4 board:



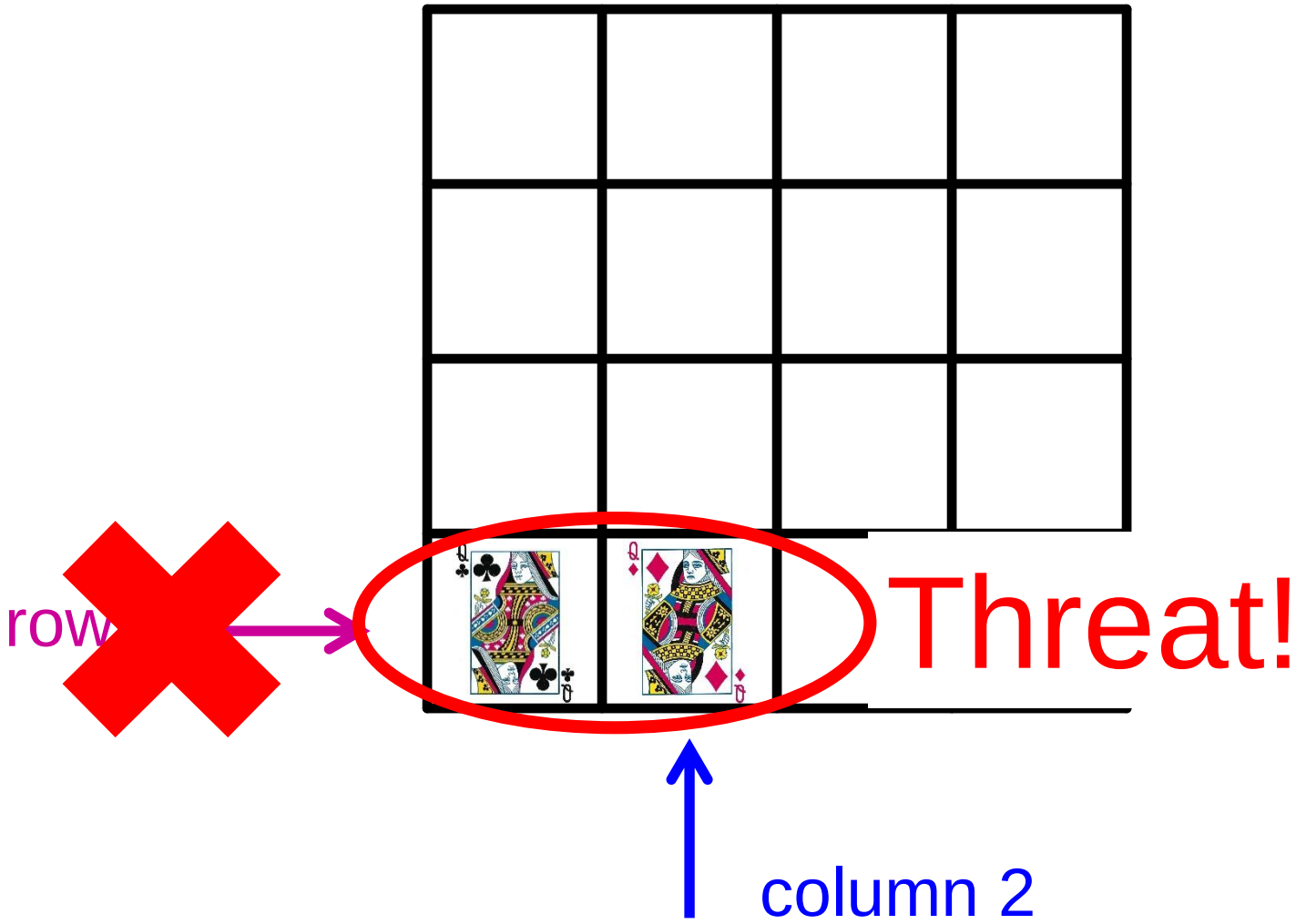
Start by placing a queen in column 1 and row 1:



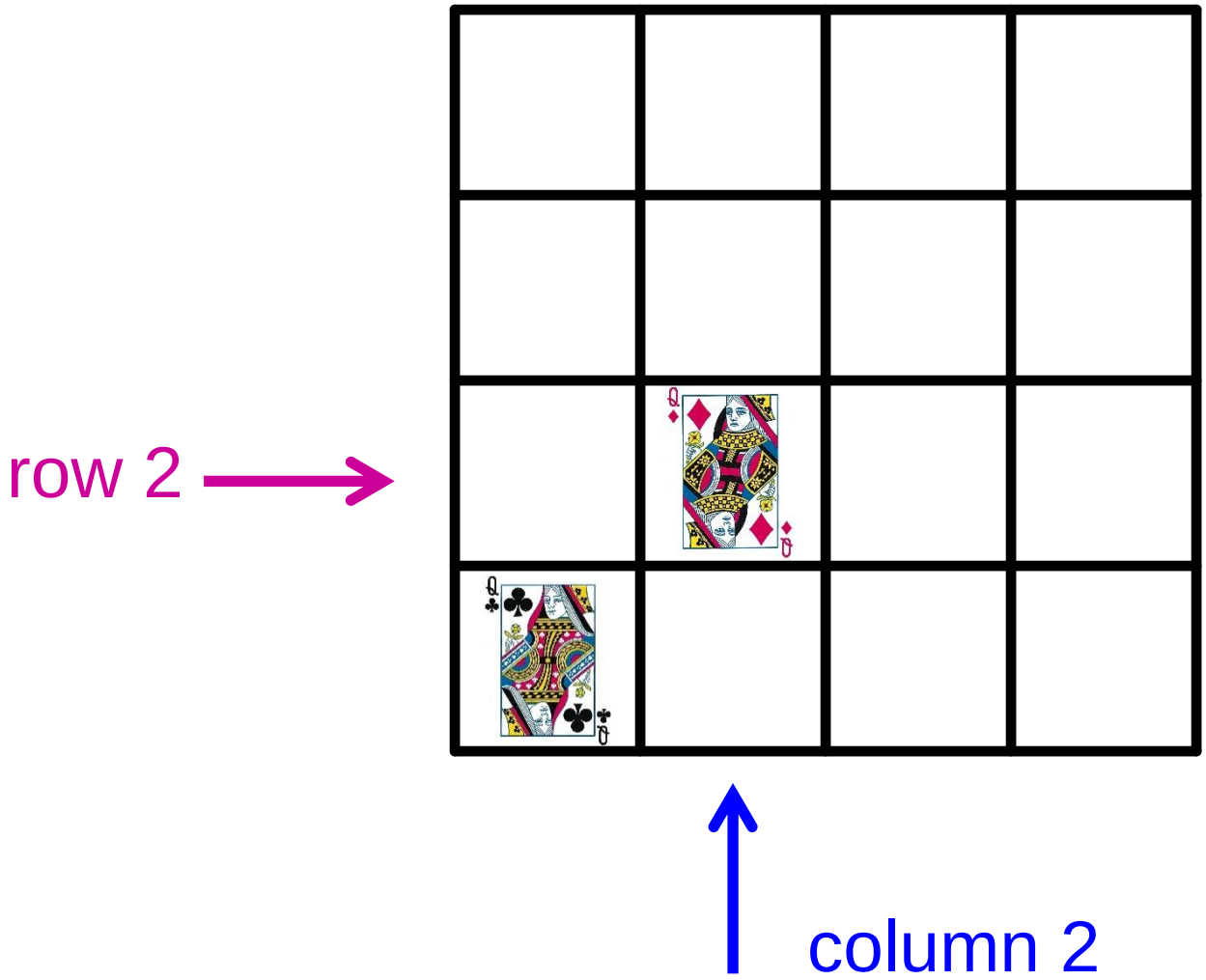
Add a queen to column 2 by trying different rows:



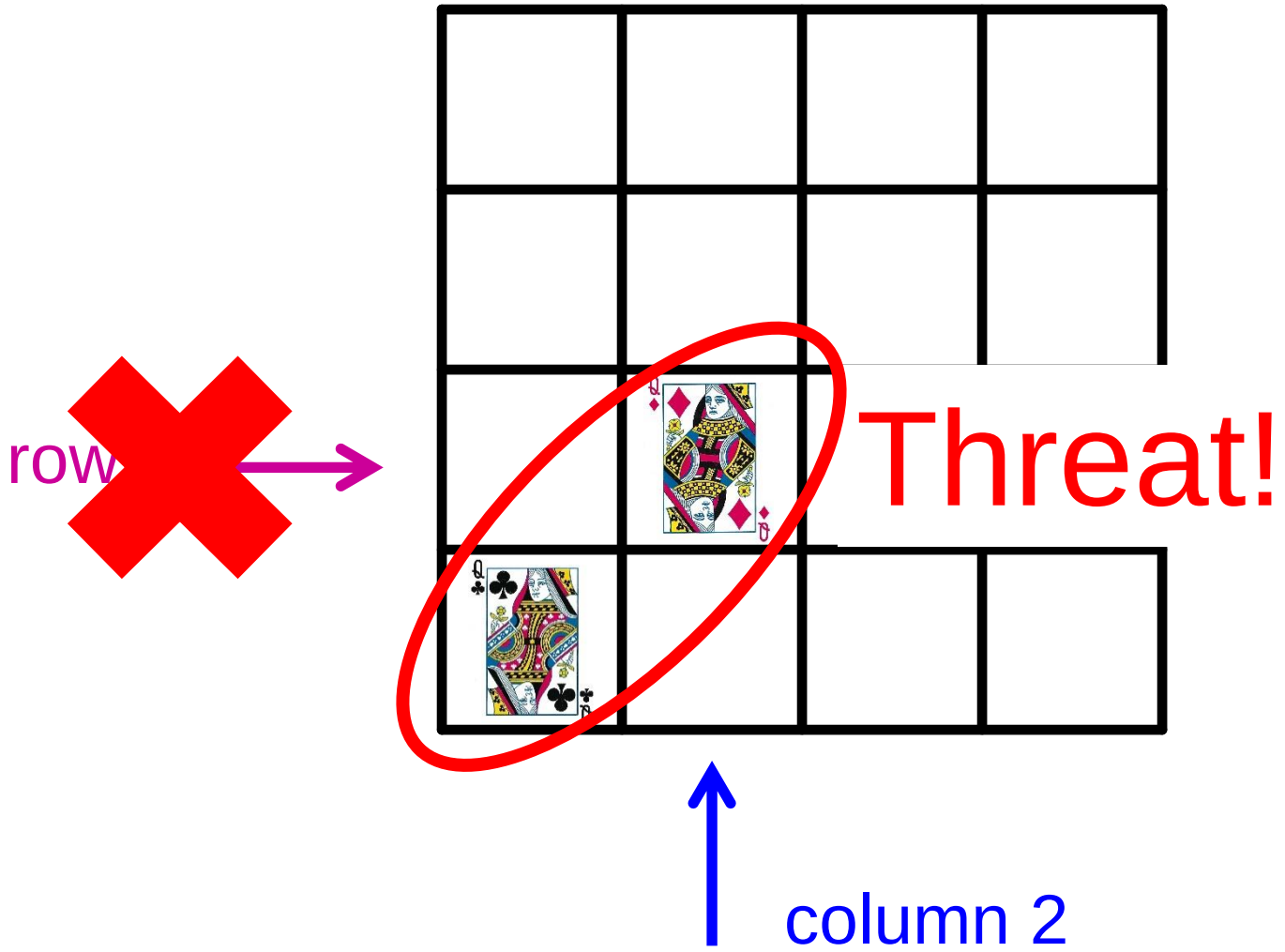
Add a queen to column 2 by trying different rows:



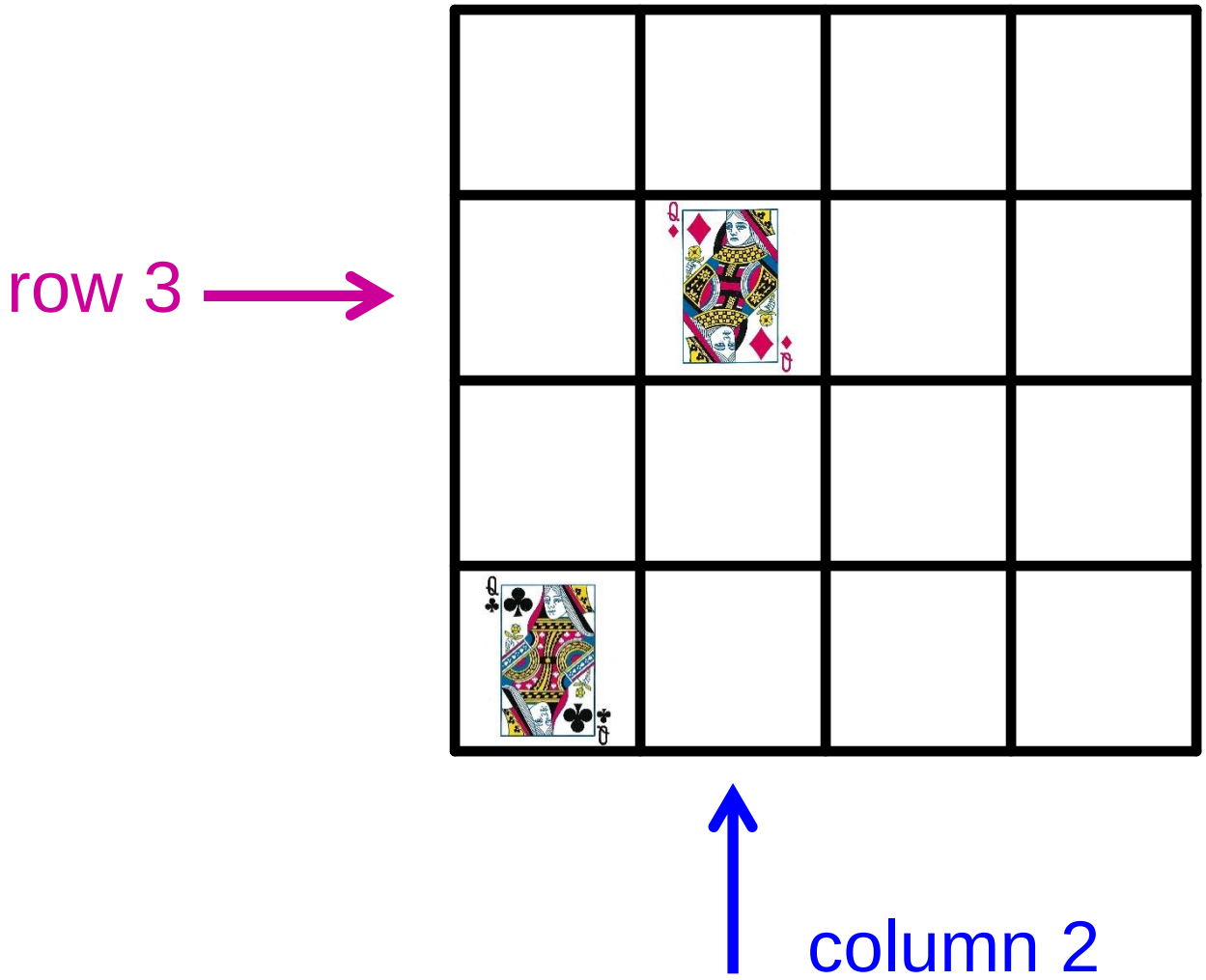
Add a queen to column 2 by trying different rows:



Add a queen to column 2 by trying different rows:

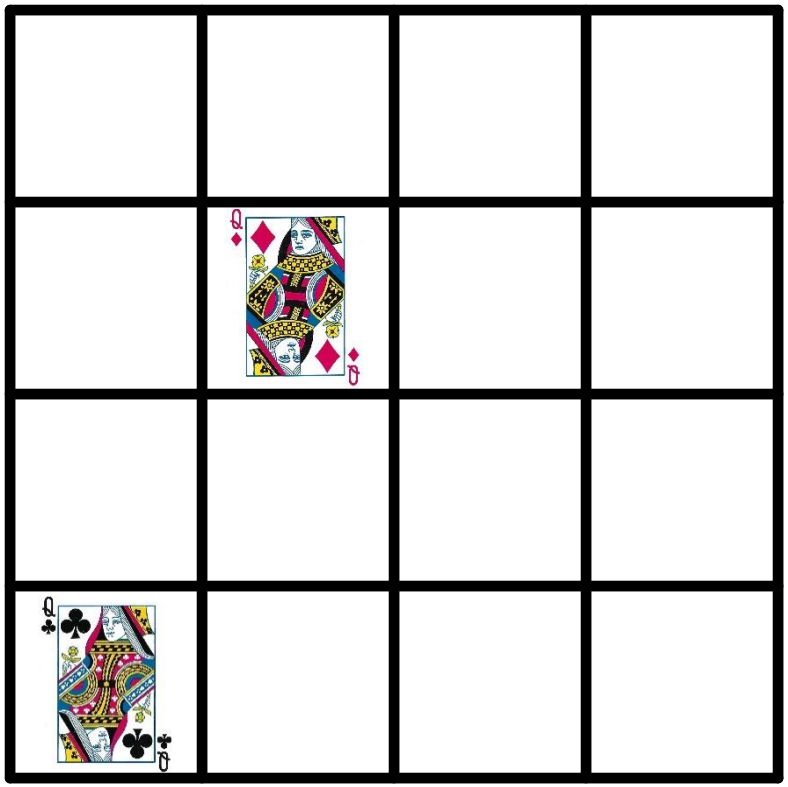


Add a queen to column 2 by trying different rows:



Add a queen to column 2 by trying different rows:

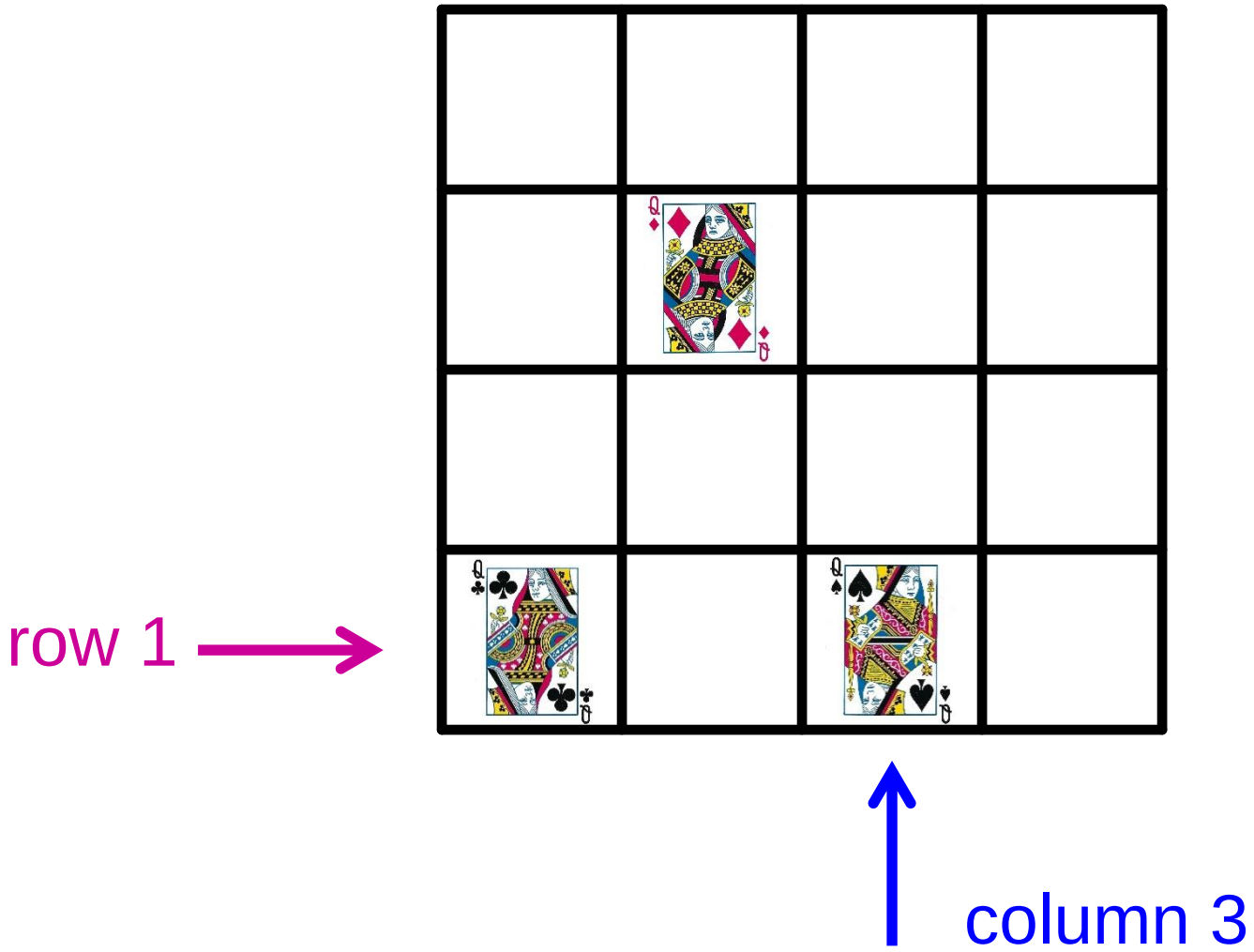
row 3 →



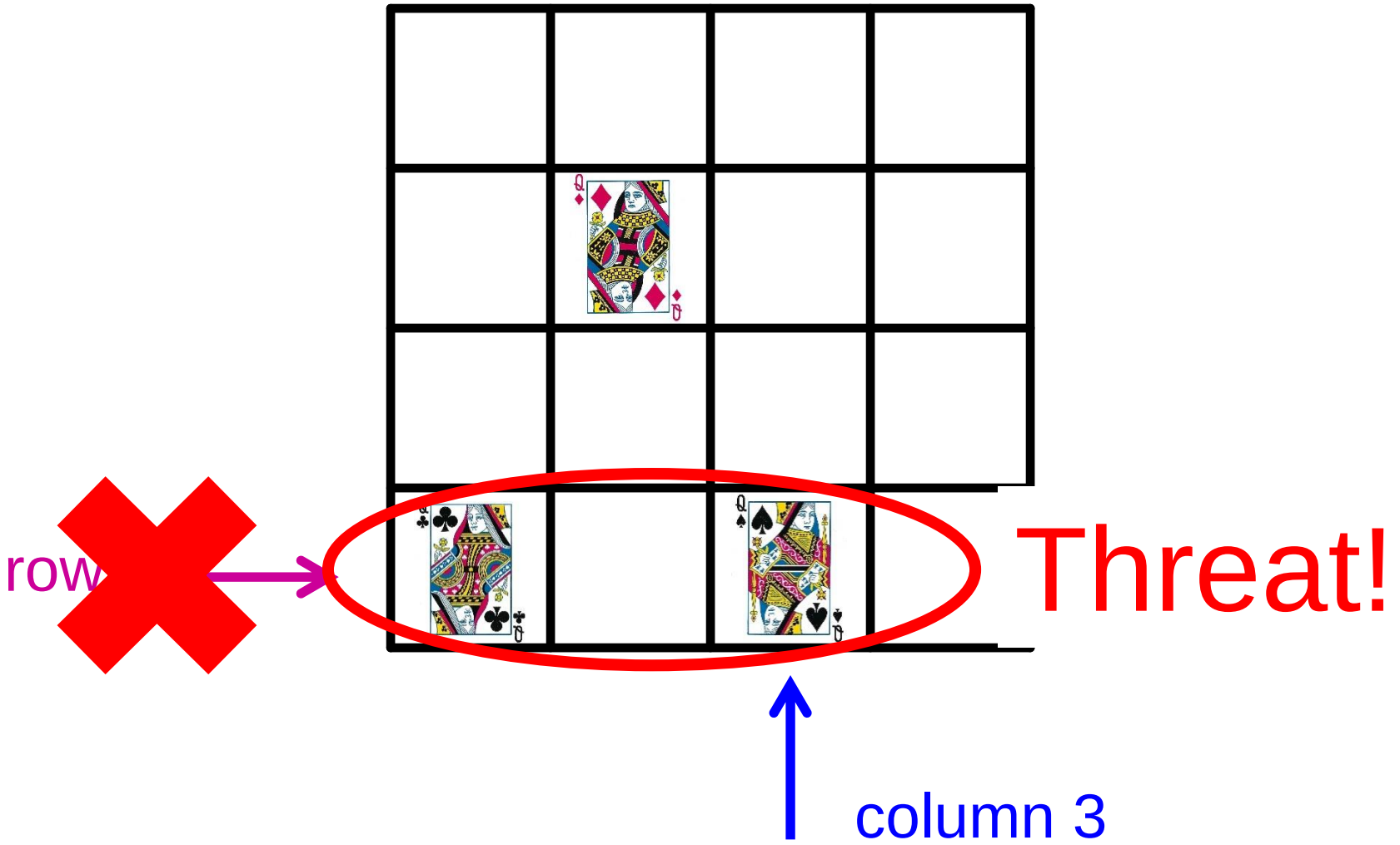
That succeeds!

↑ column 2

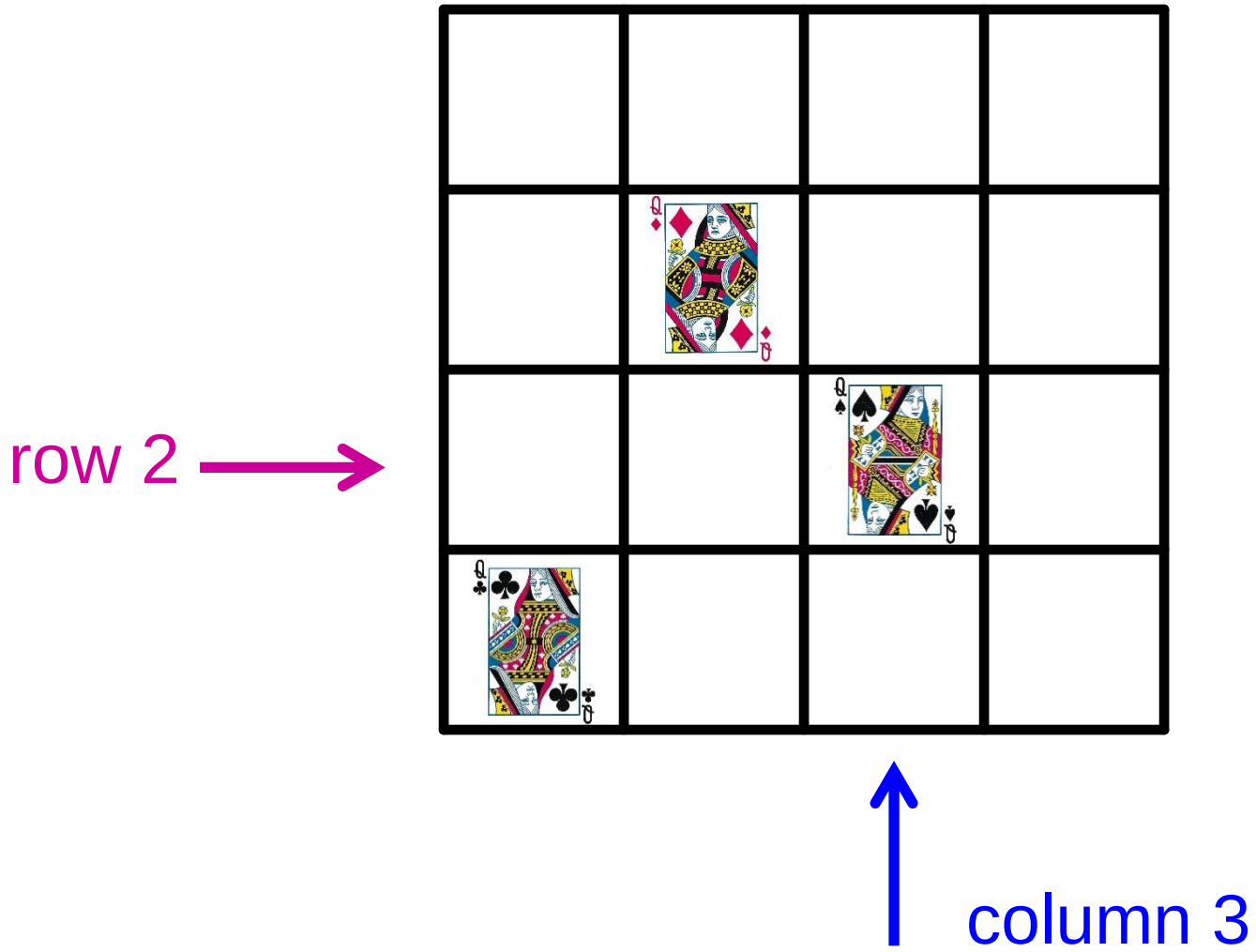
Add a queen to column 3 by trying different rows:



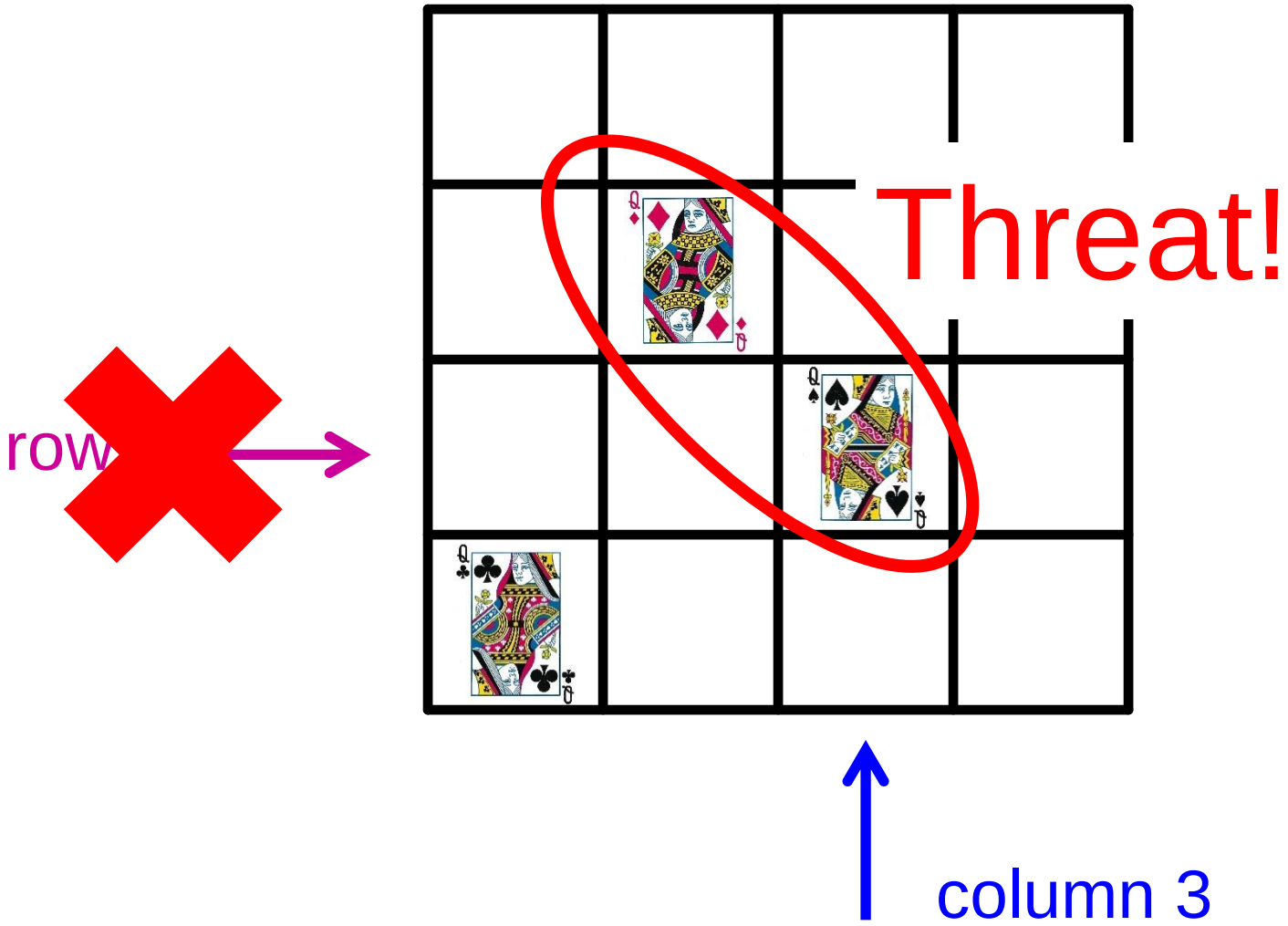
Add a queen to column 3 by trying different rows:



Add a queen to column 3 by trying different rows:

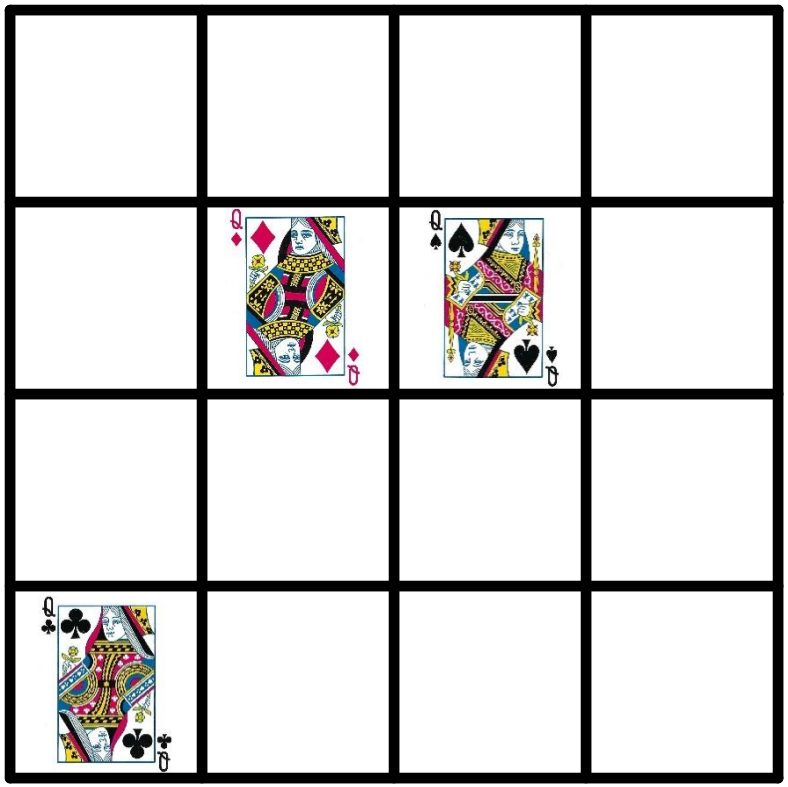


Add a queen to column 3 by trying different rows:



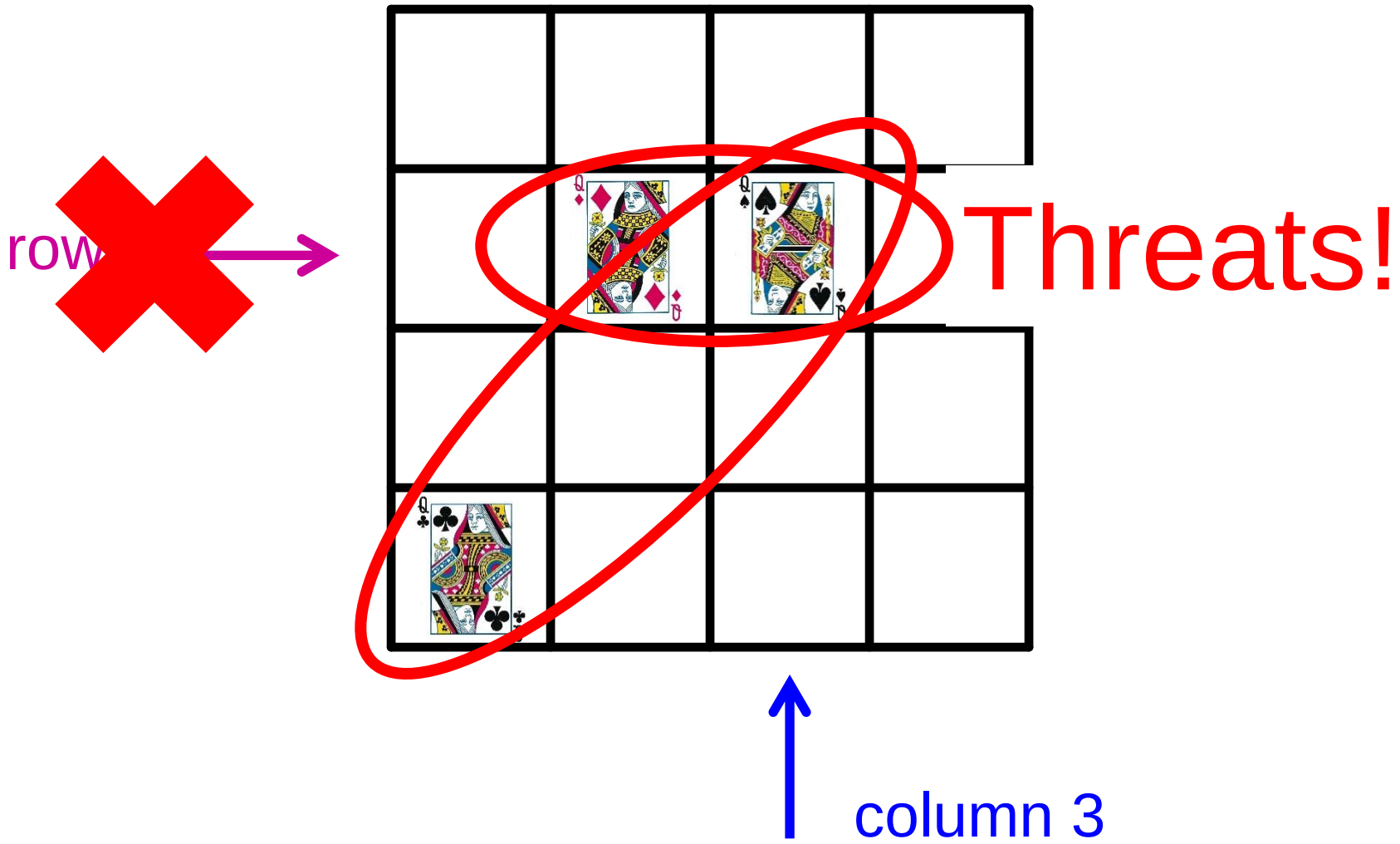
Add a queen to column 3 by trying different rows:

row 3 →



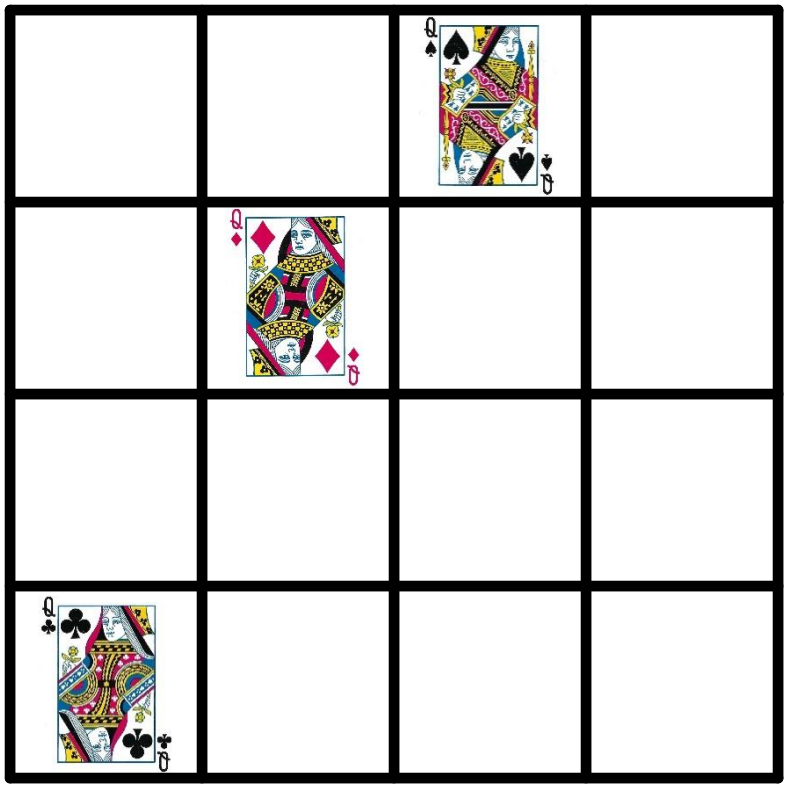
↑ column 3

Add a queen to column 3 by trying different rows:



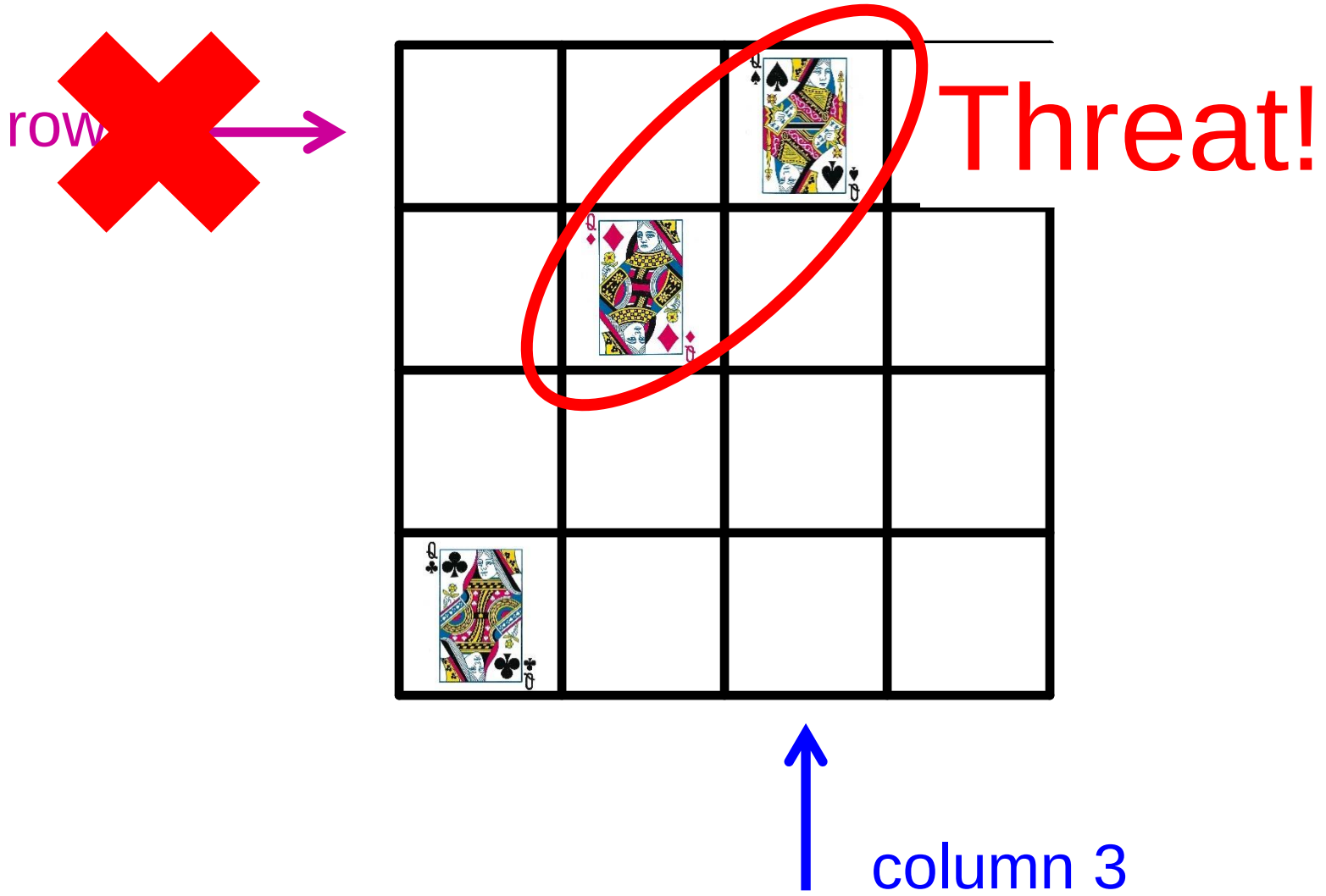
Add a queen to column 3 by trying different rows:

row 4 →



↑ column 3

Add a queen to column 3 by trying different rows:

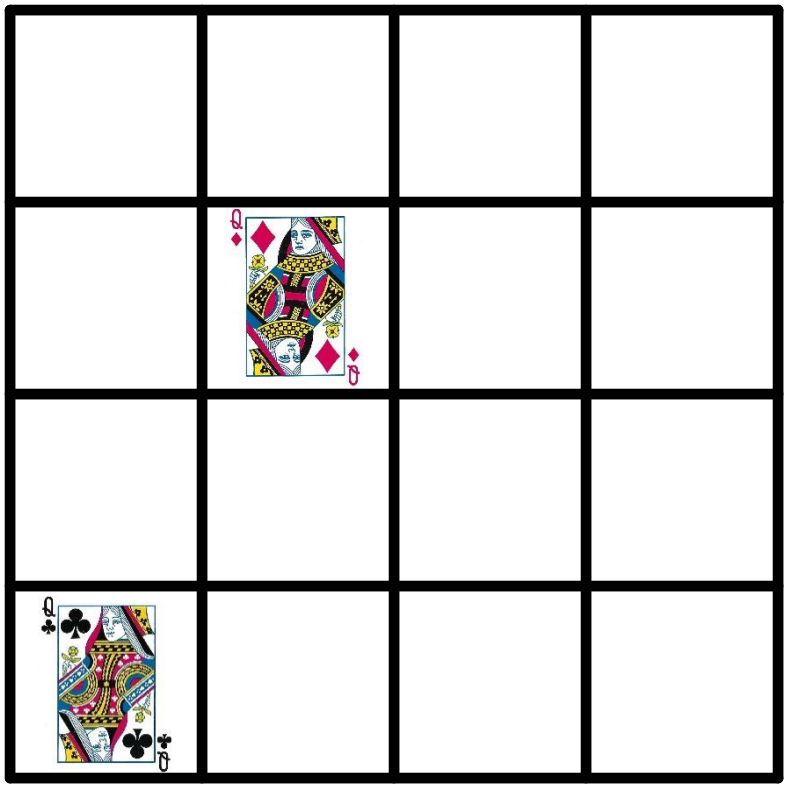


OH NO! We cannot place the third queen!

column 3

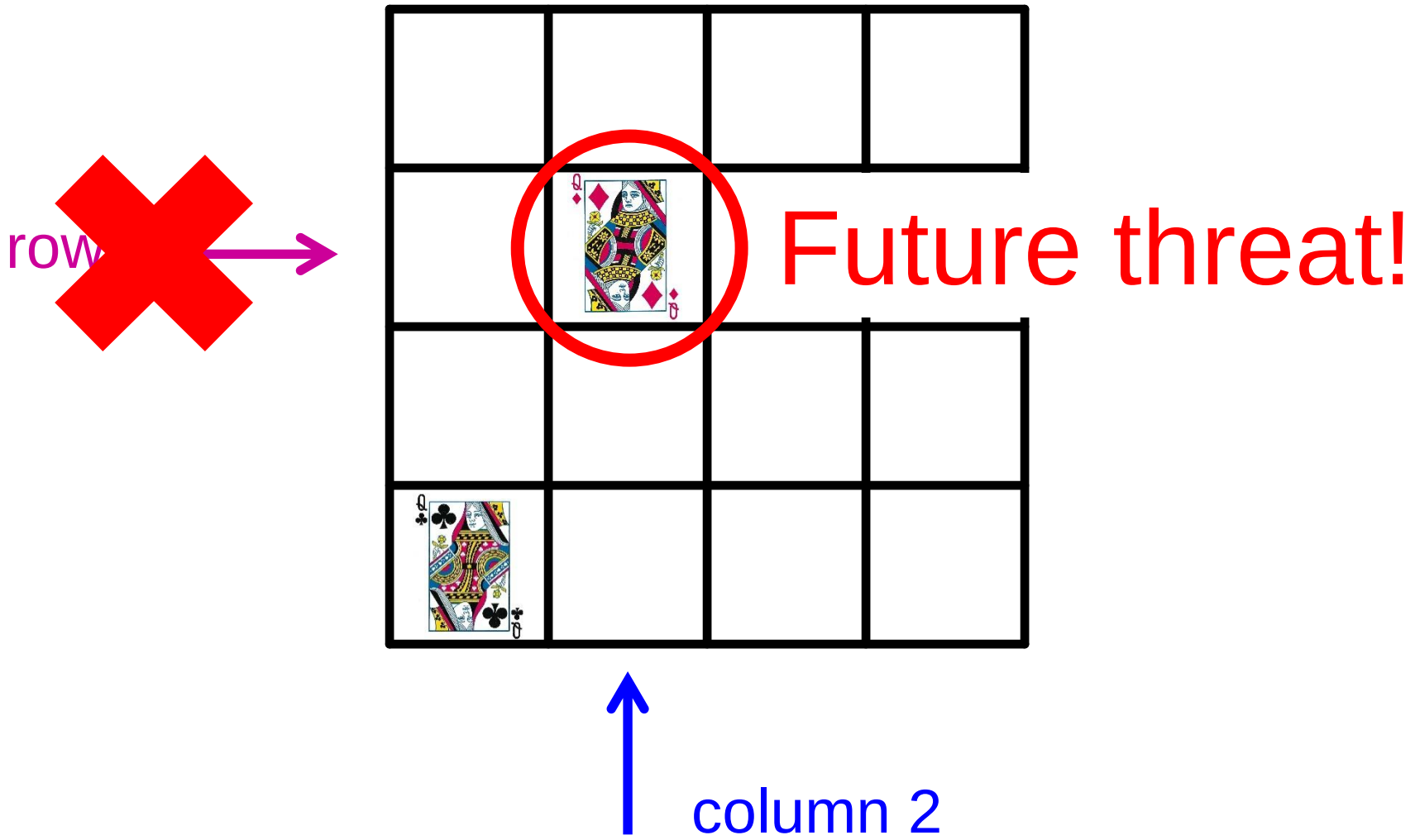
OH NO! We cannot place the third queen!

Let's backtrack to the placement of second queen.

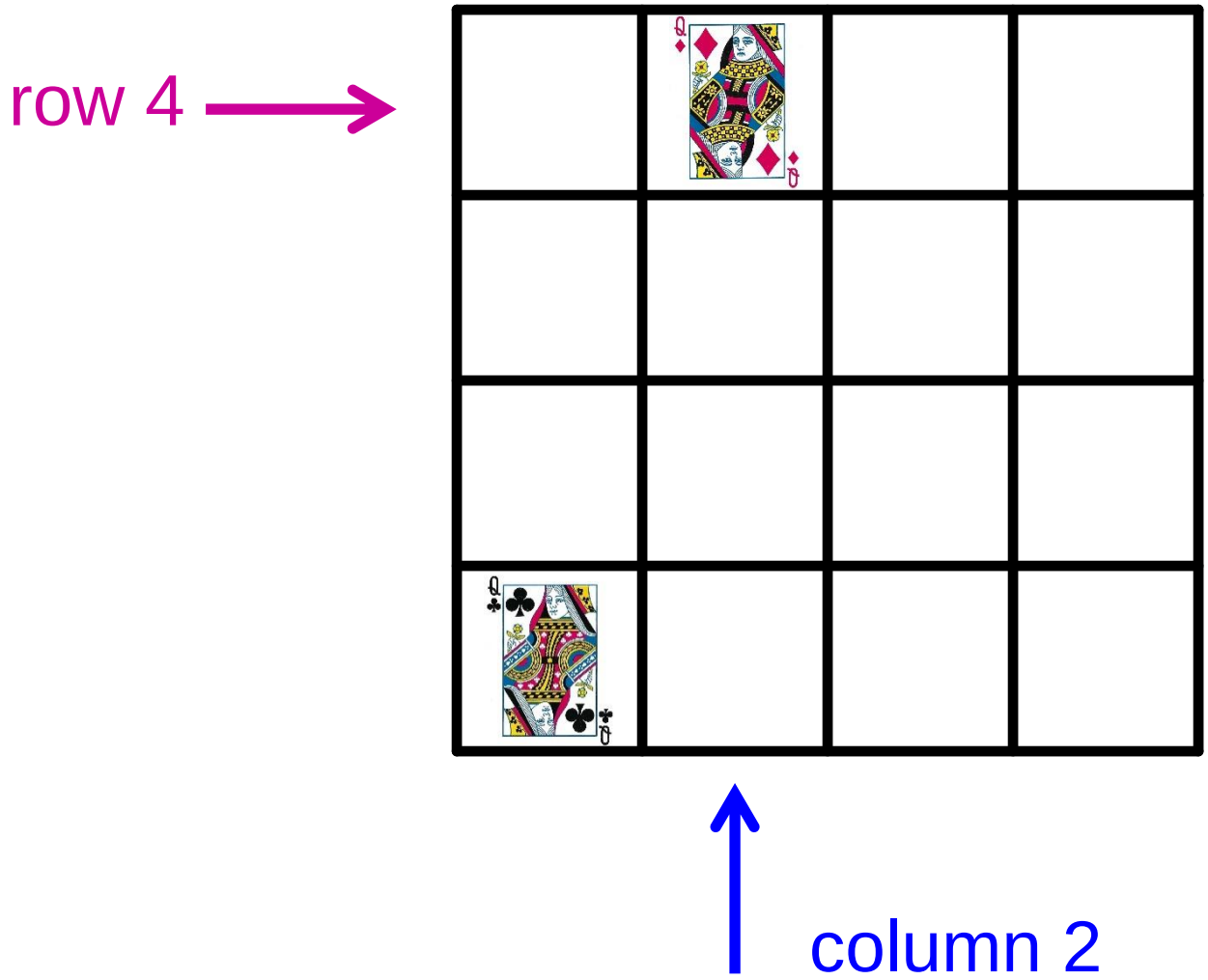


column 2

Let's backtrack to the placement of second queen.

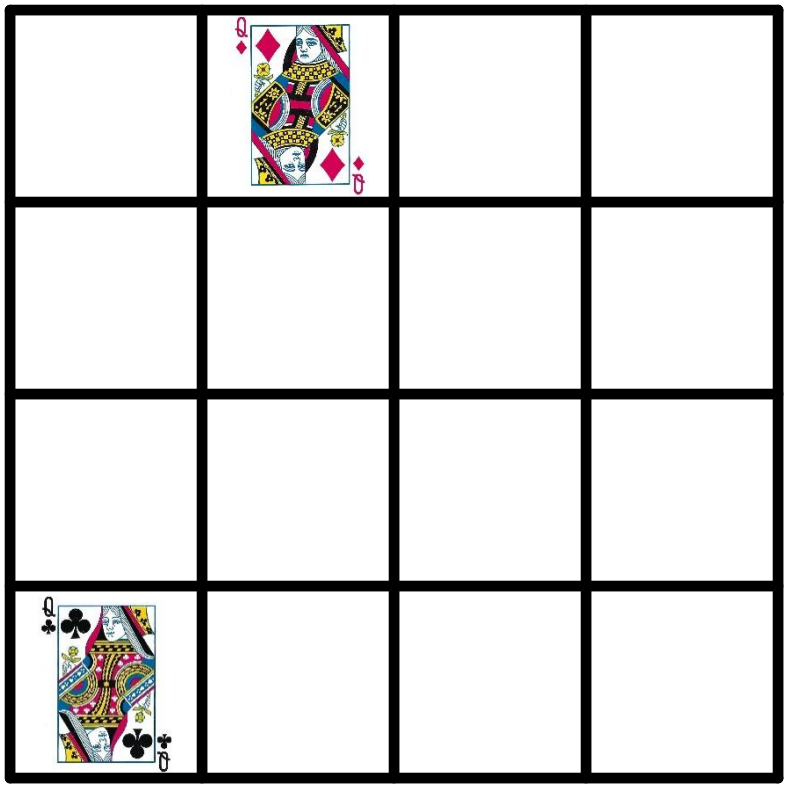


Let's try a new placement for the second queen.



Let's try a new placement for the second queen.

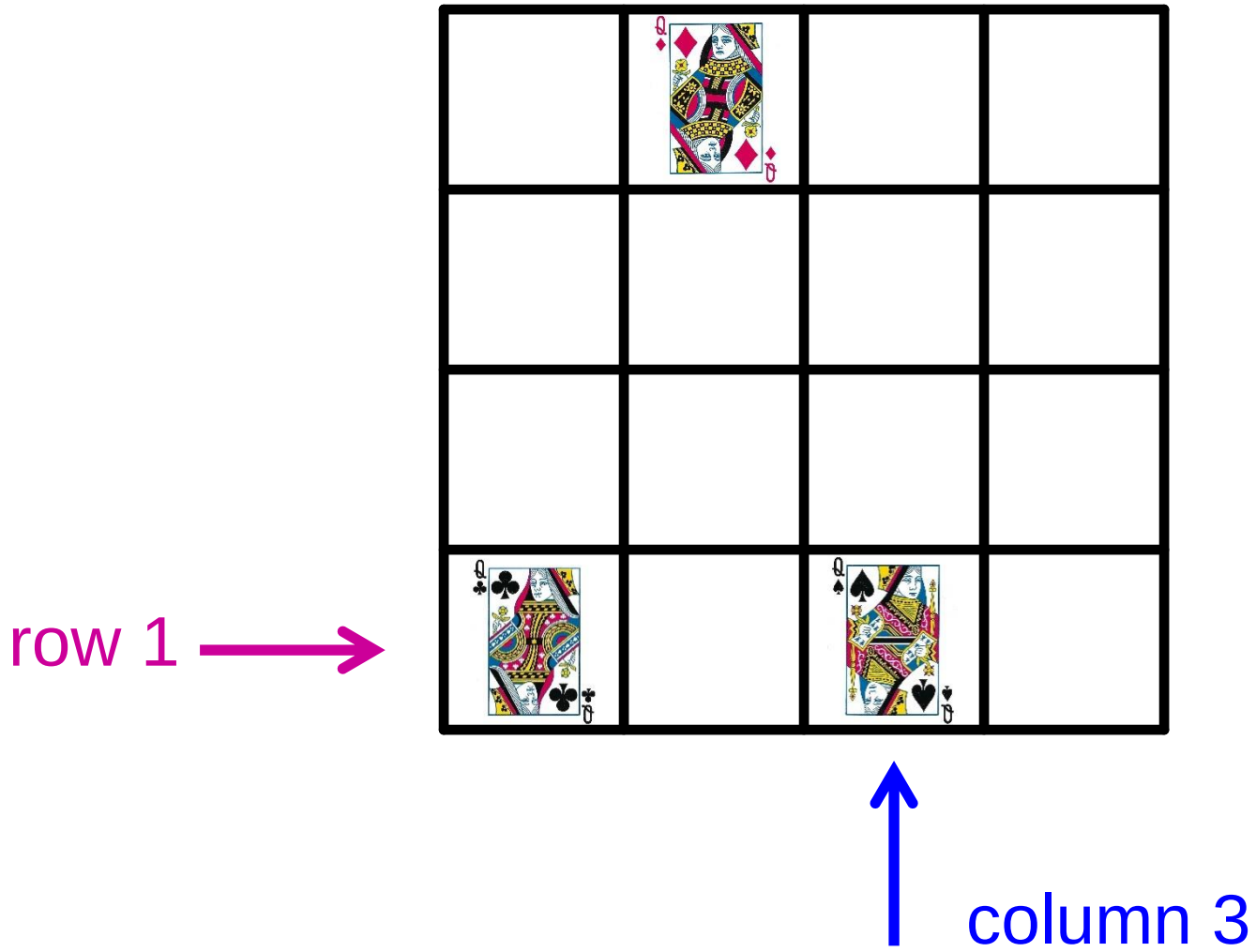
row 4 →



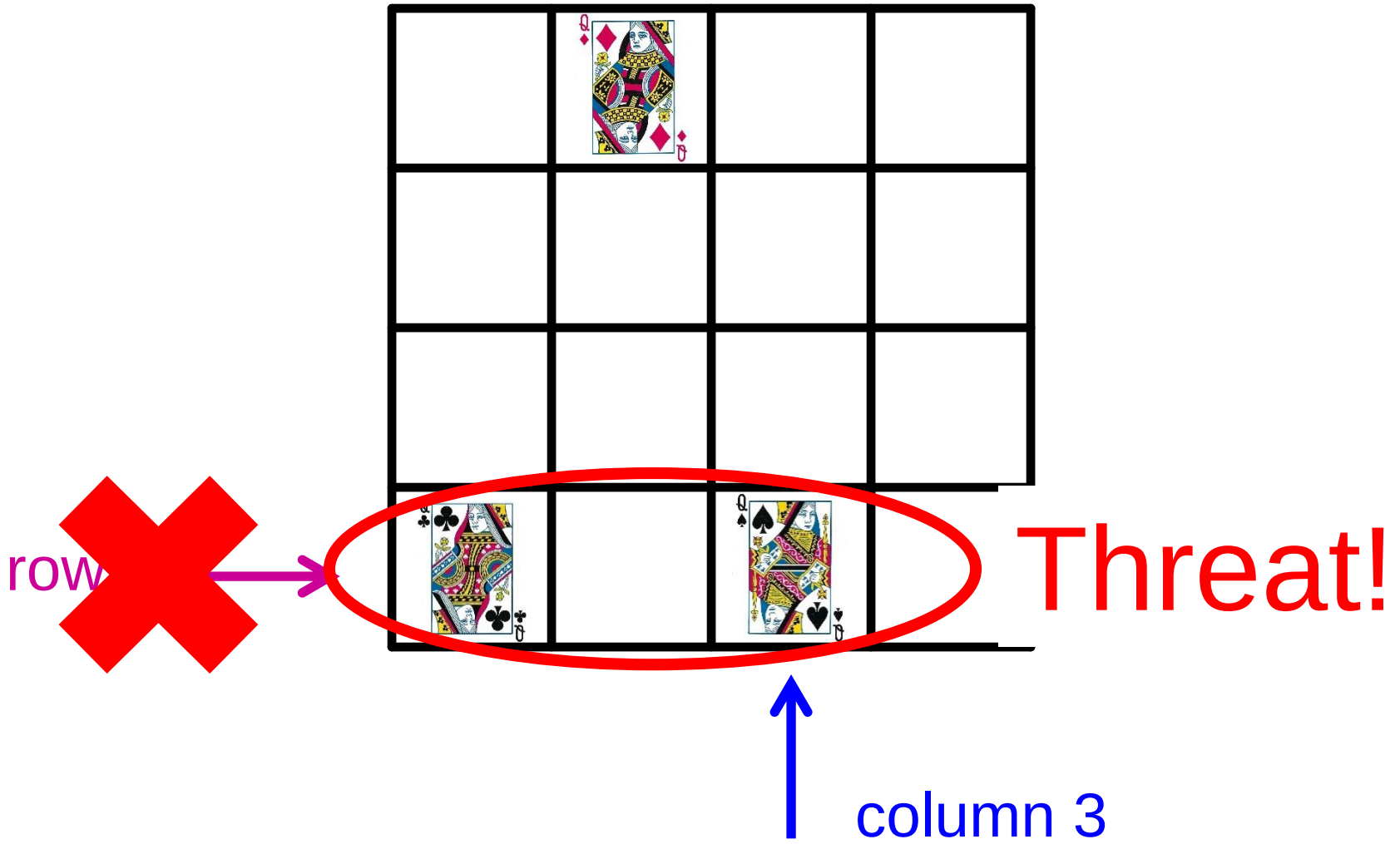
That succeeds!

↑ column 2

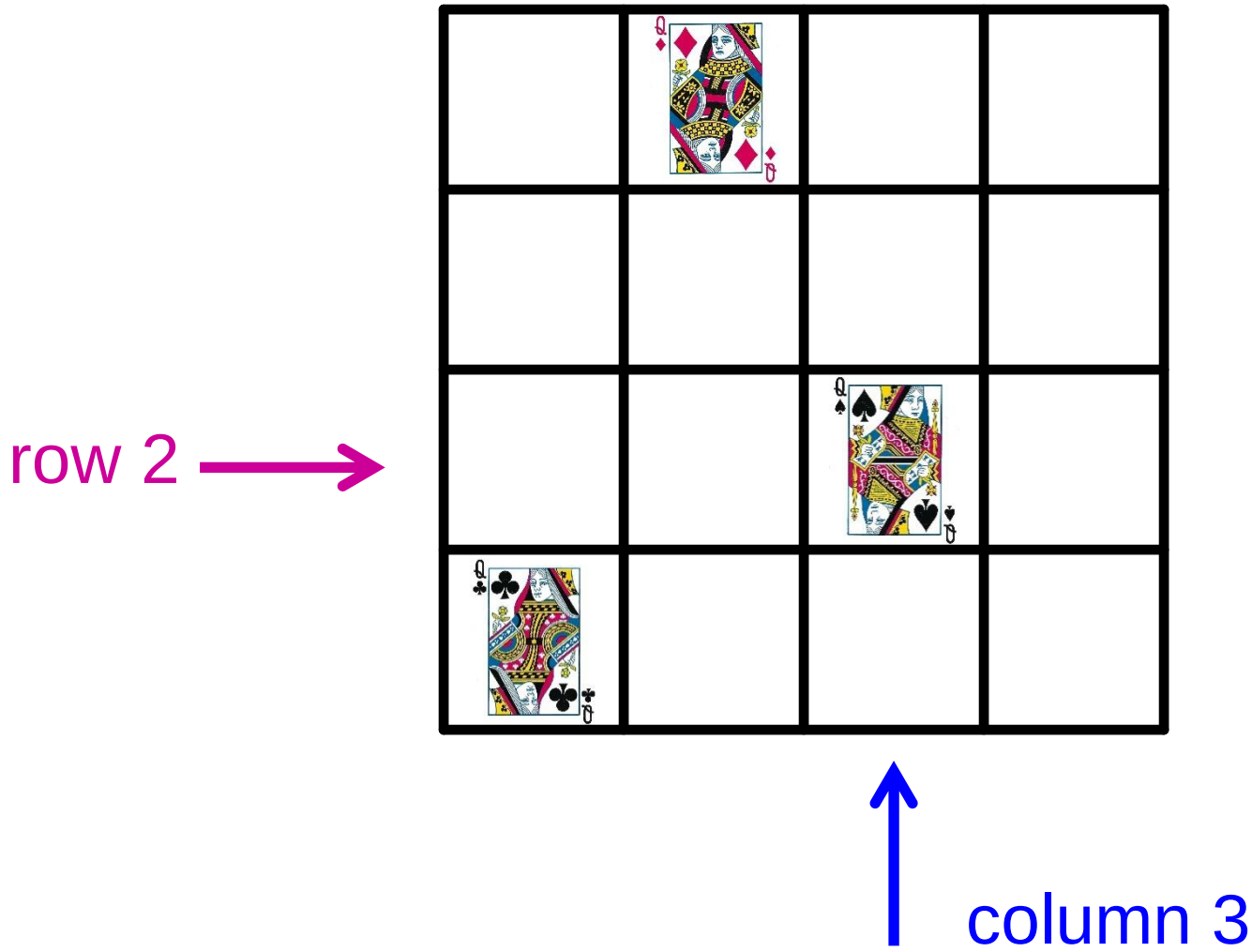
Again add a queen to column 3 by trying different rows:



Again add a queen to column 3 by trying different rows:

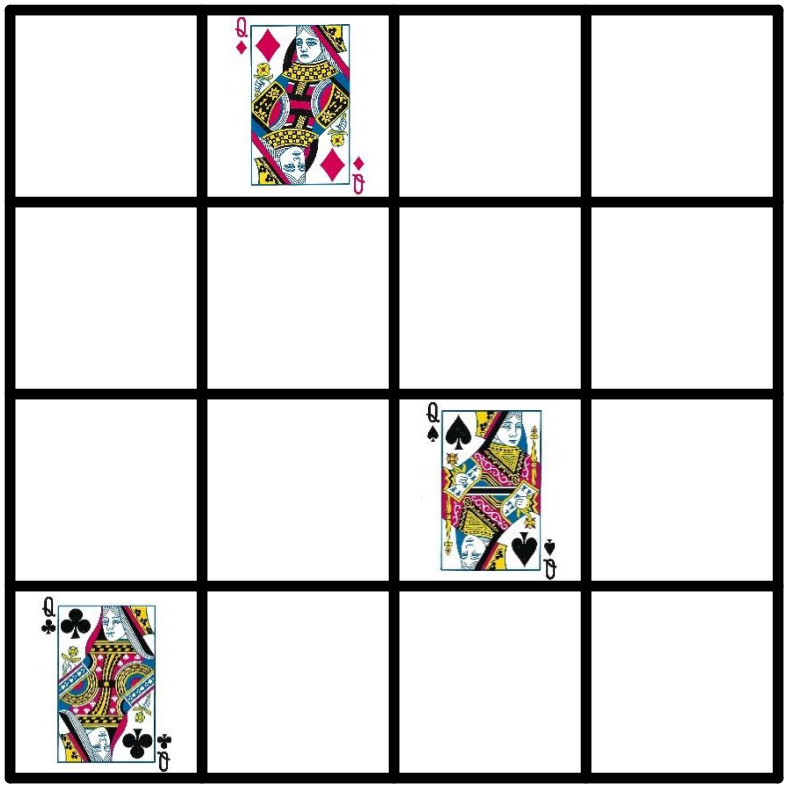


Again add a queen to column 3 by trying different rows:



Again add a queen to column 3 by trying different rows:

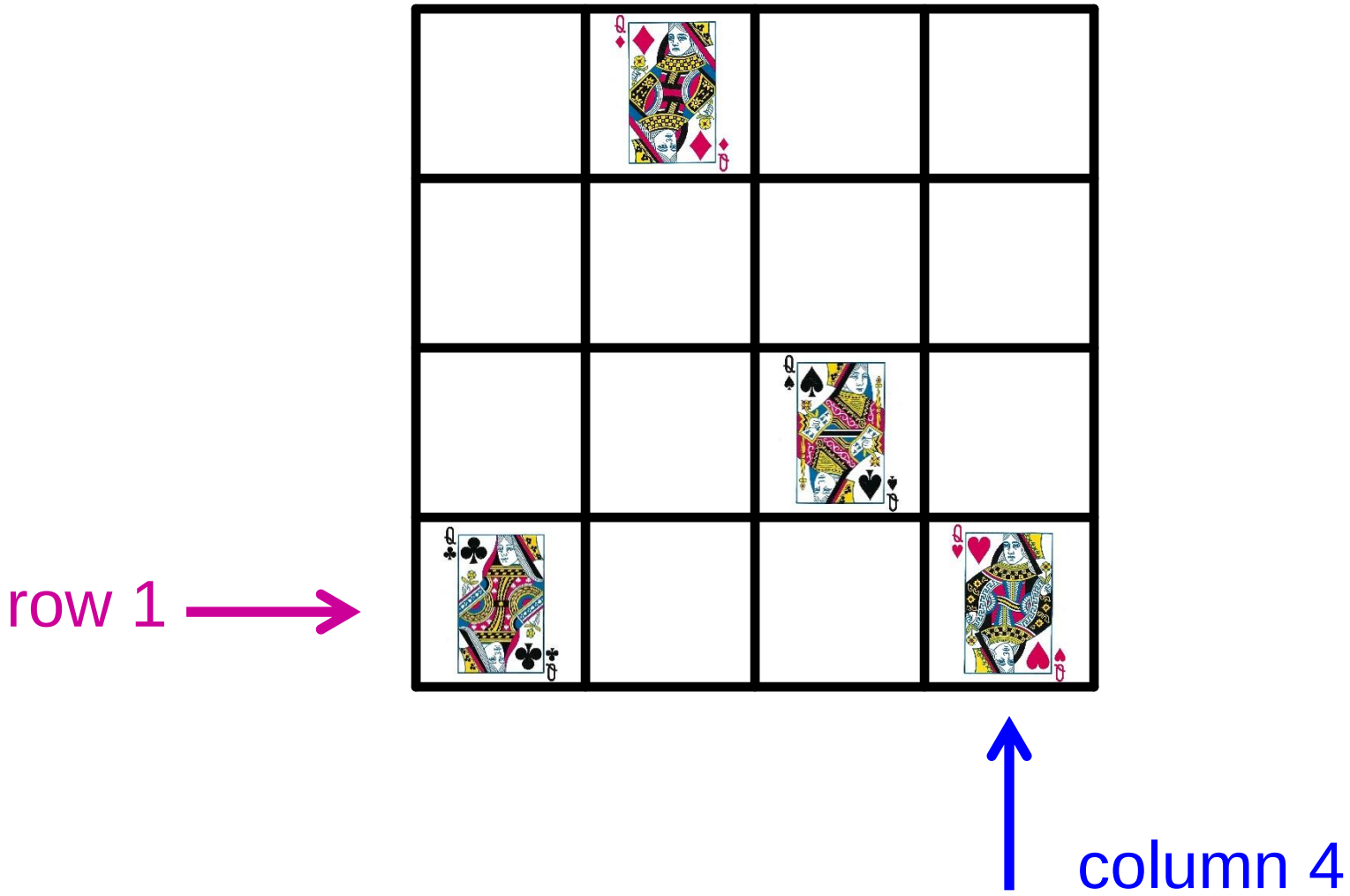
row 2 →



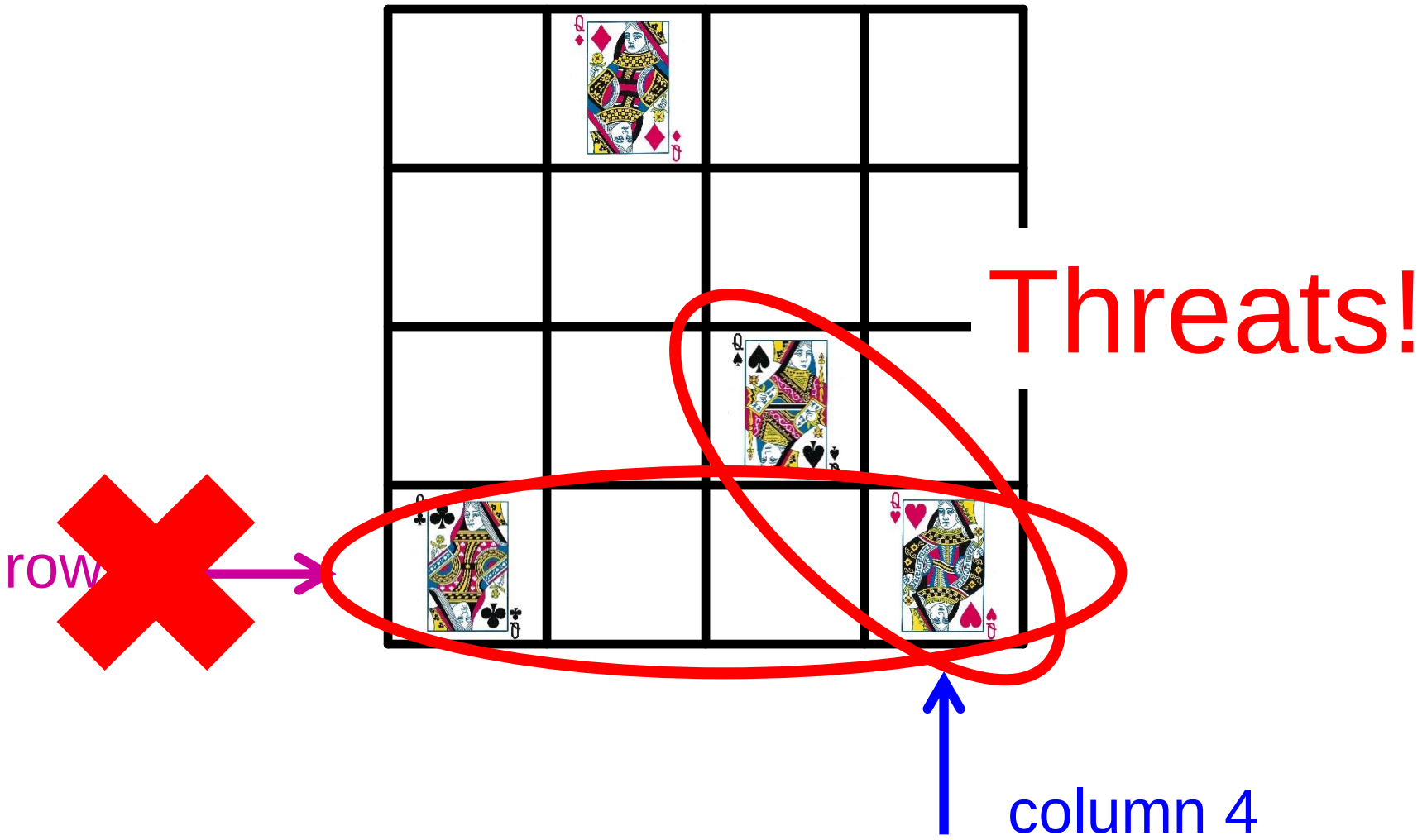
That succeeds!

↑ column 3

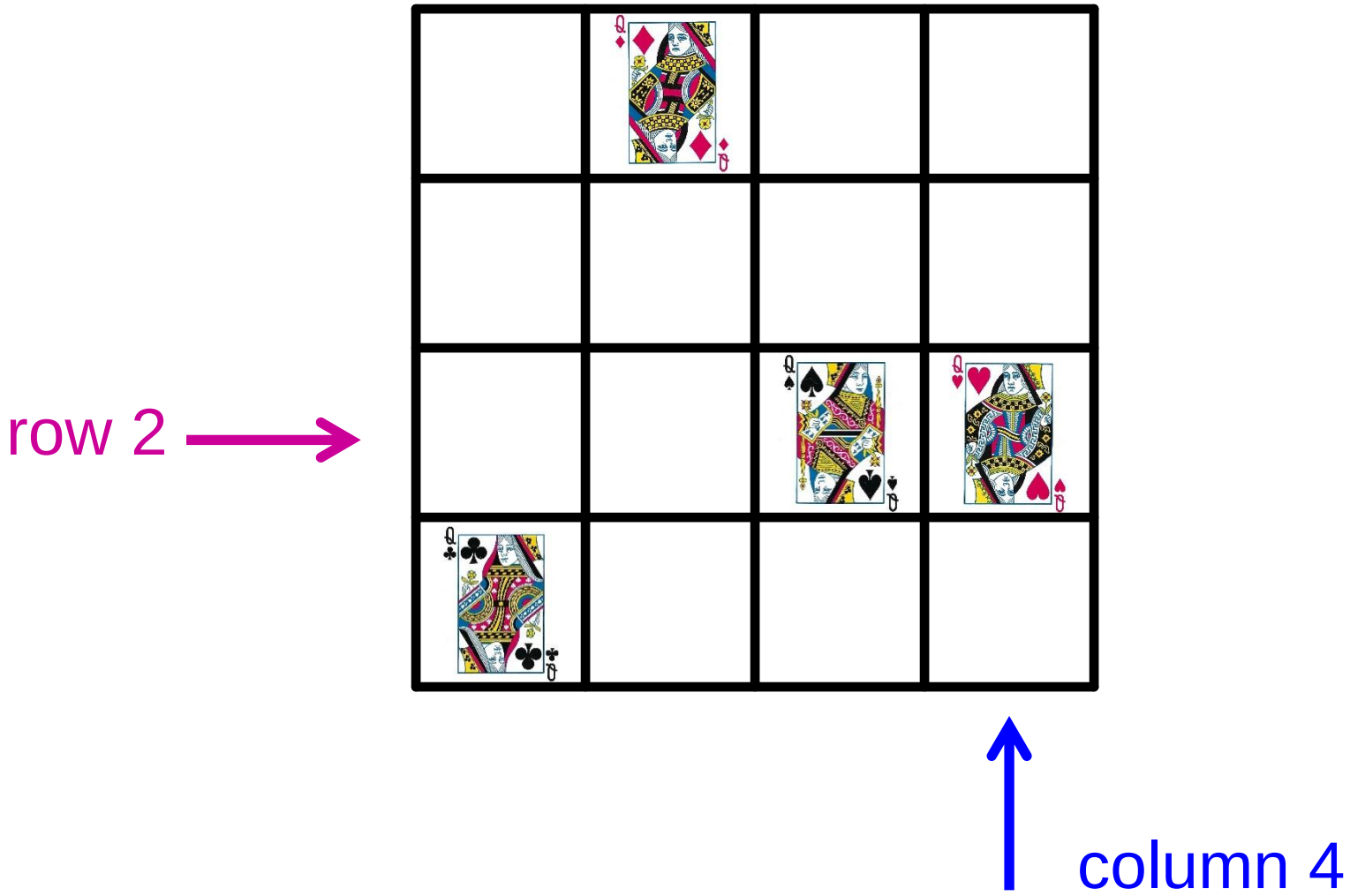
Add a queen to column 4 by trying different rows:



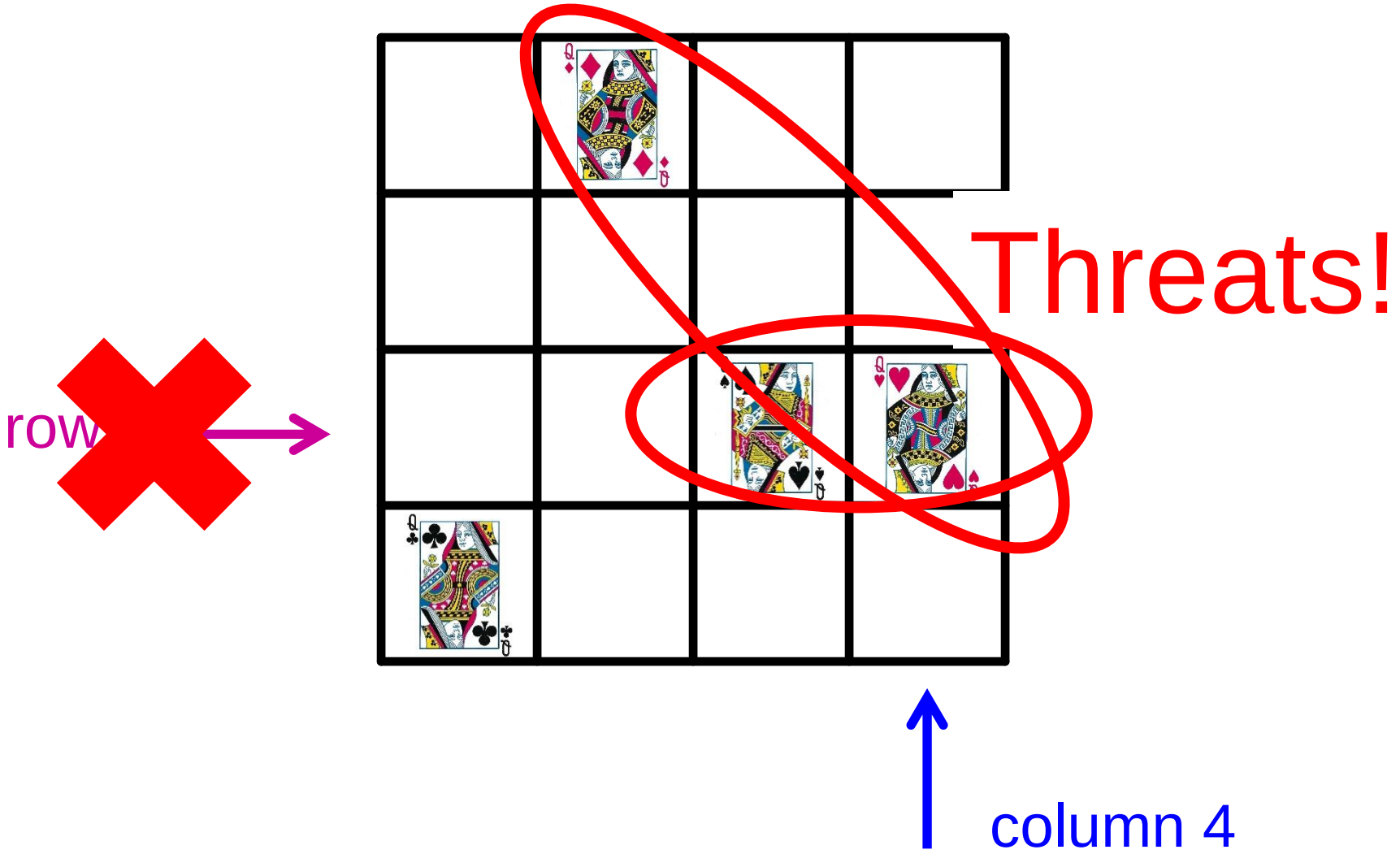
Add a queen to column 4 by trying different rows:



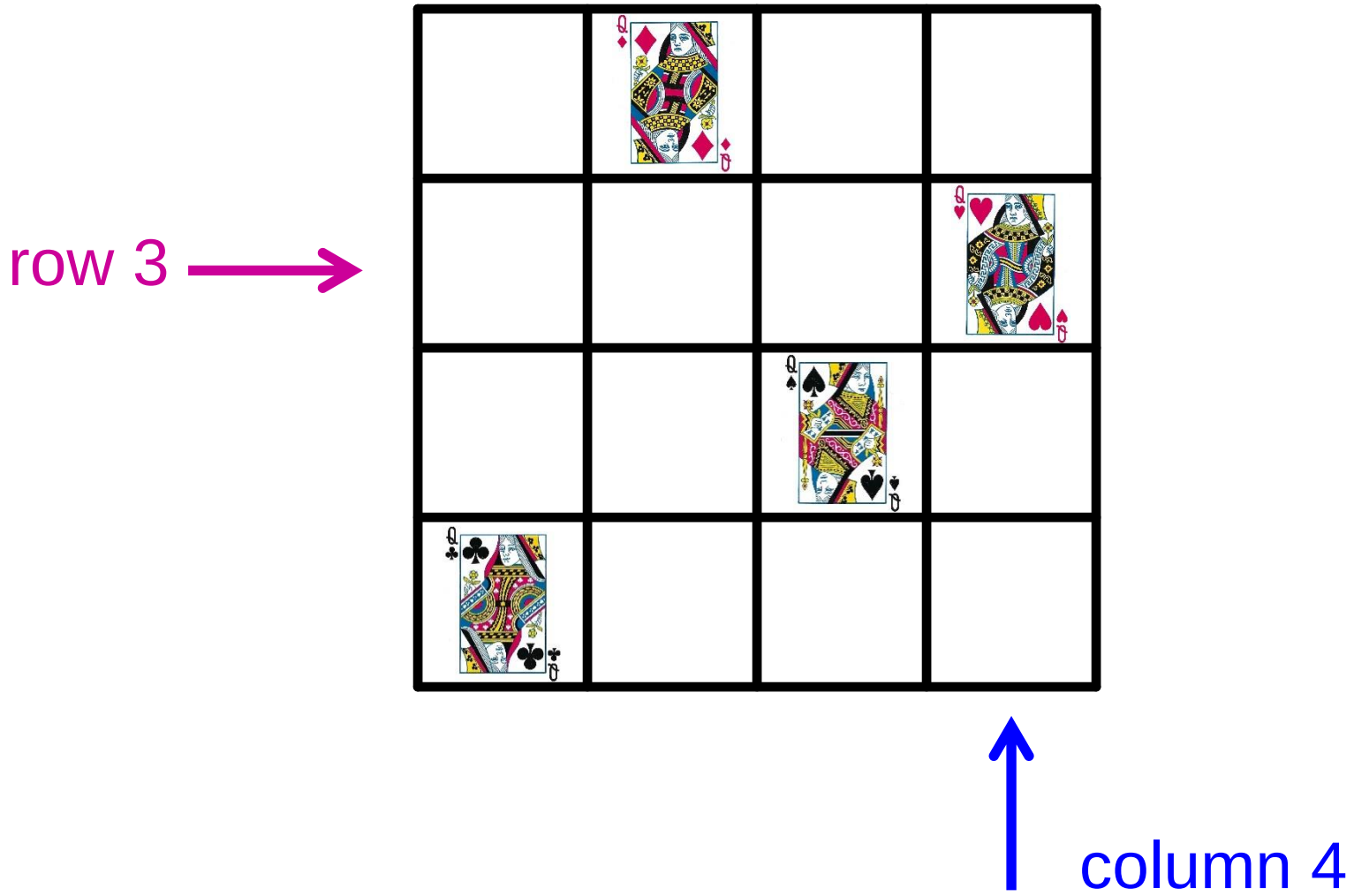
Add a queen to column 4 by trying different rows:



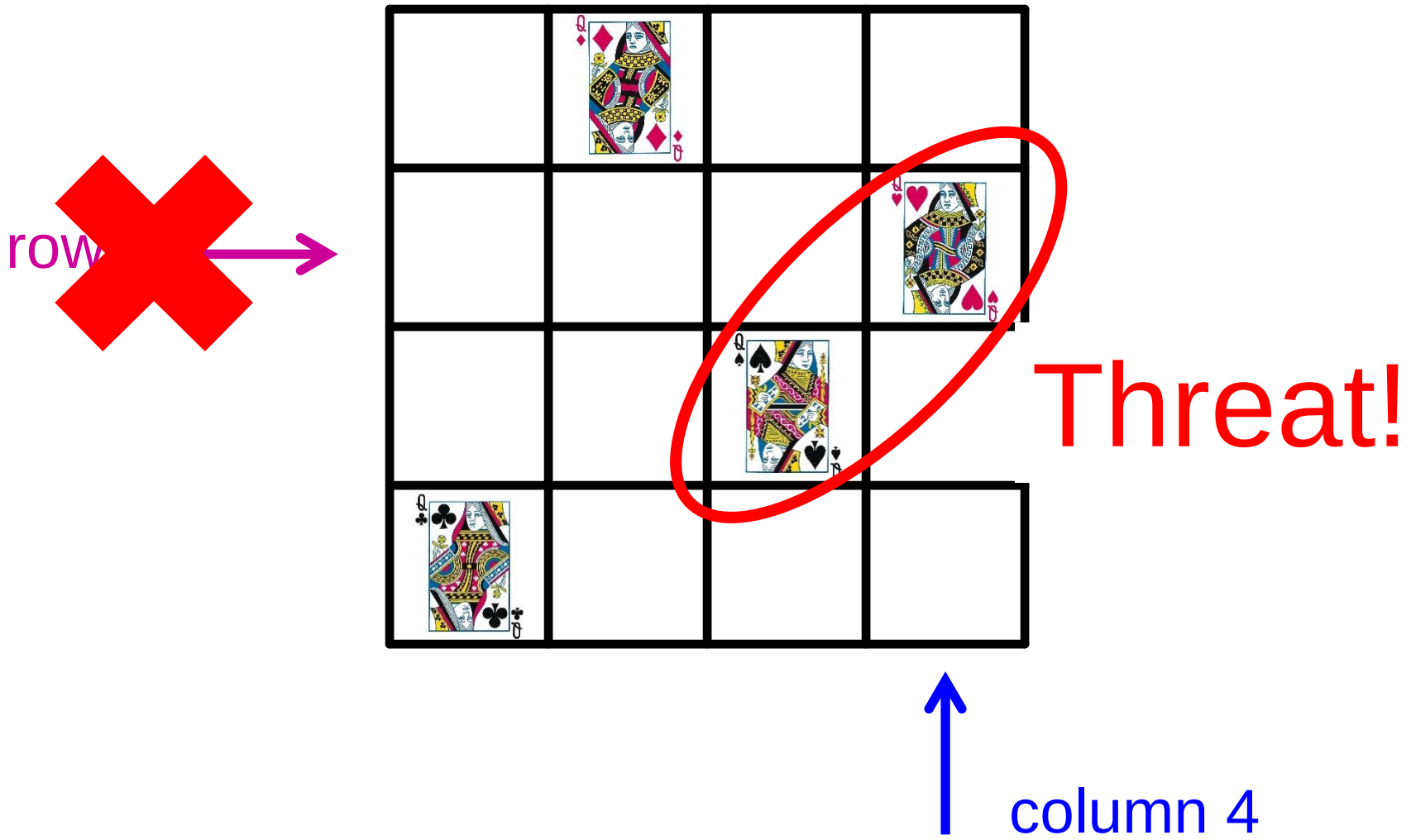
Add a queen to column 4 by trying different rows:



Add a queen to column 4 by trying different rows:

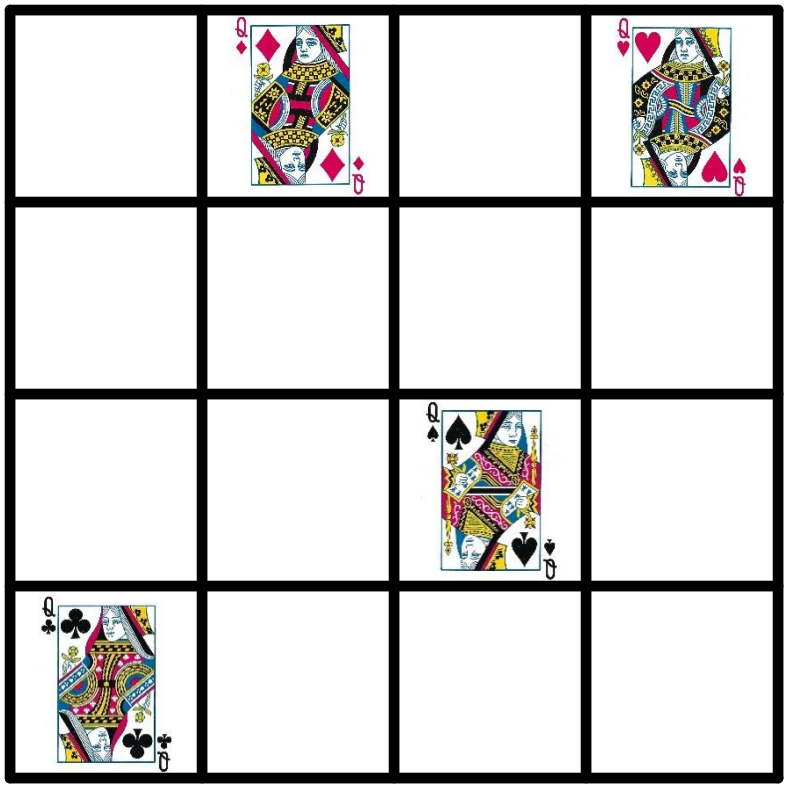


Add a queen to column 4 by trying different rows:



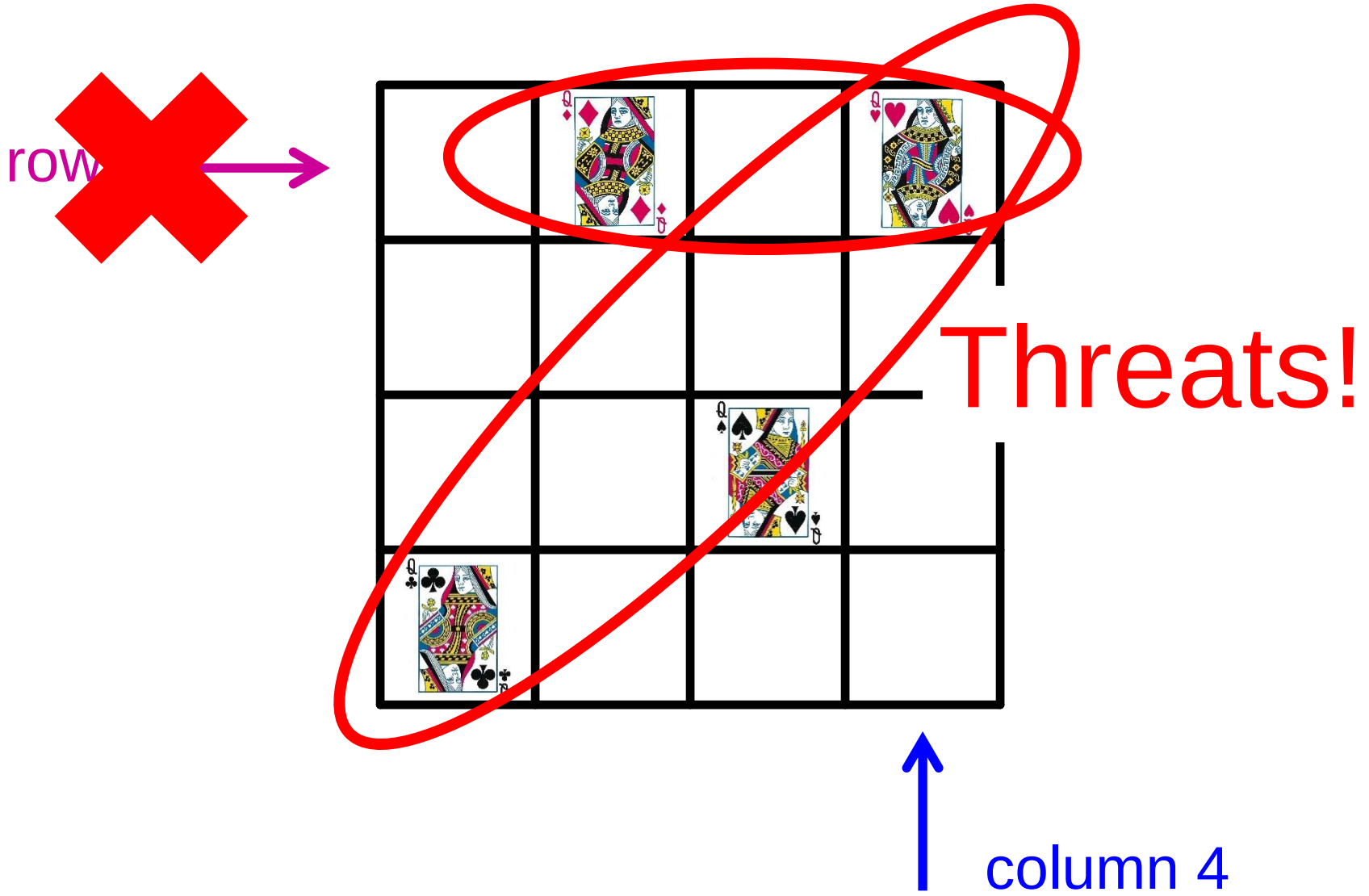
Add a queen to column 4 by trying different rows:

row 4 →

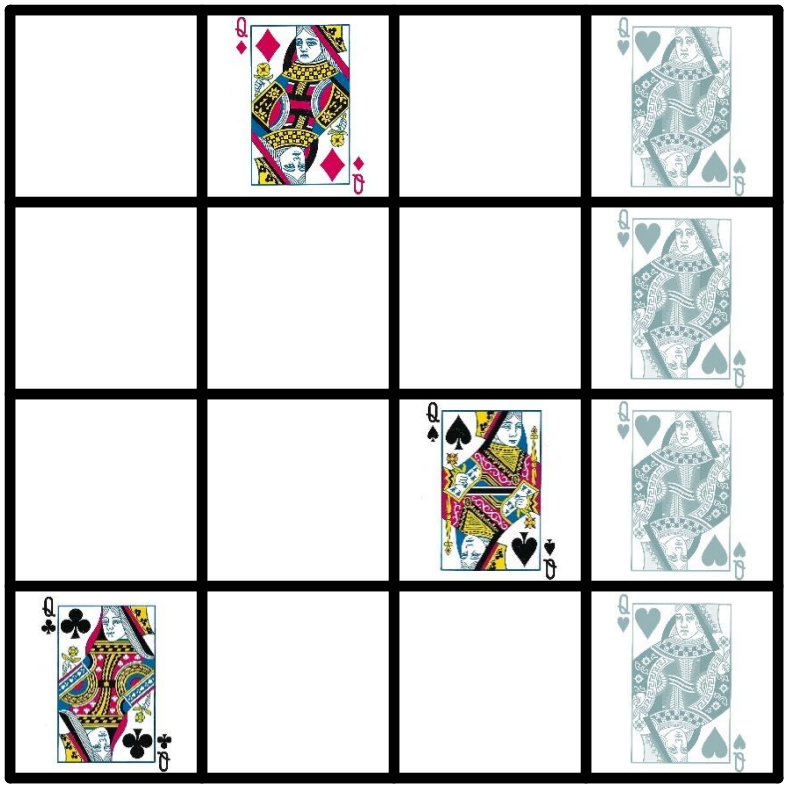


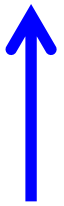
↑ column 4

Add a queen to column 4 by trying different rows:



OH NO! We cannot place the fourth queen!

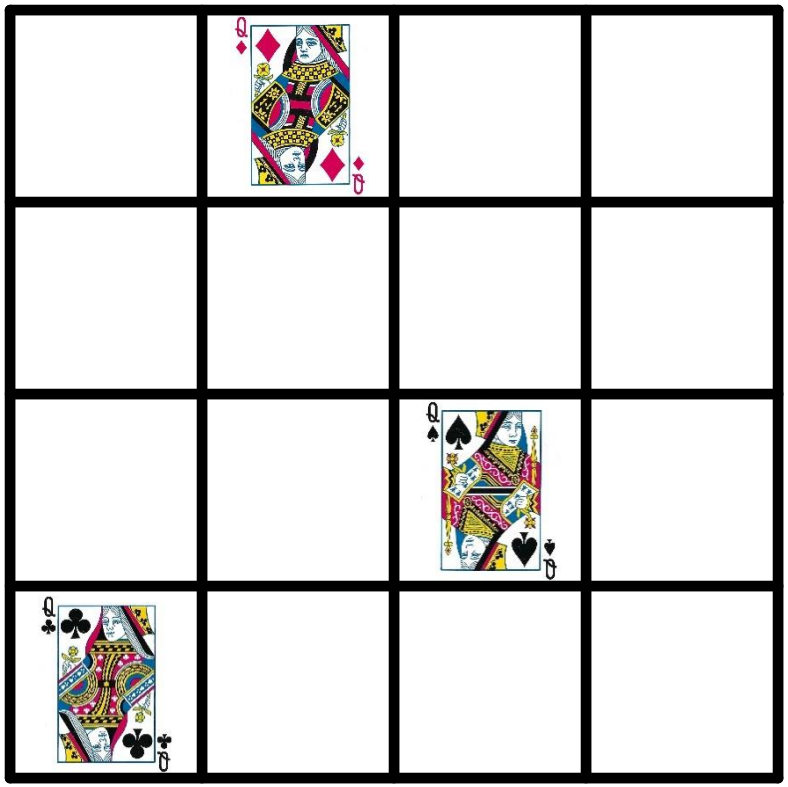




column 4

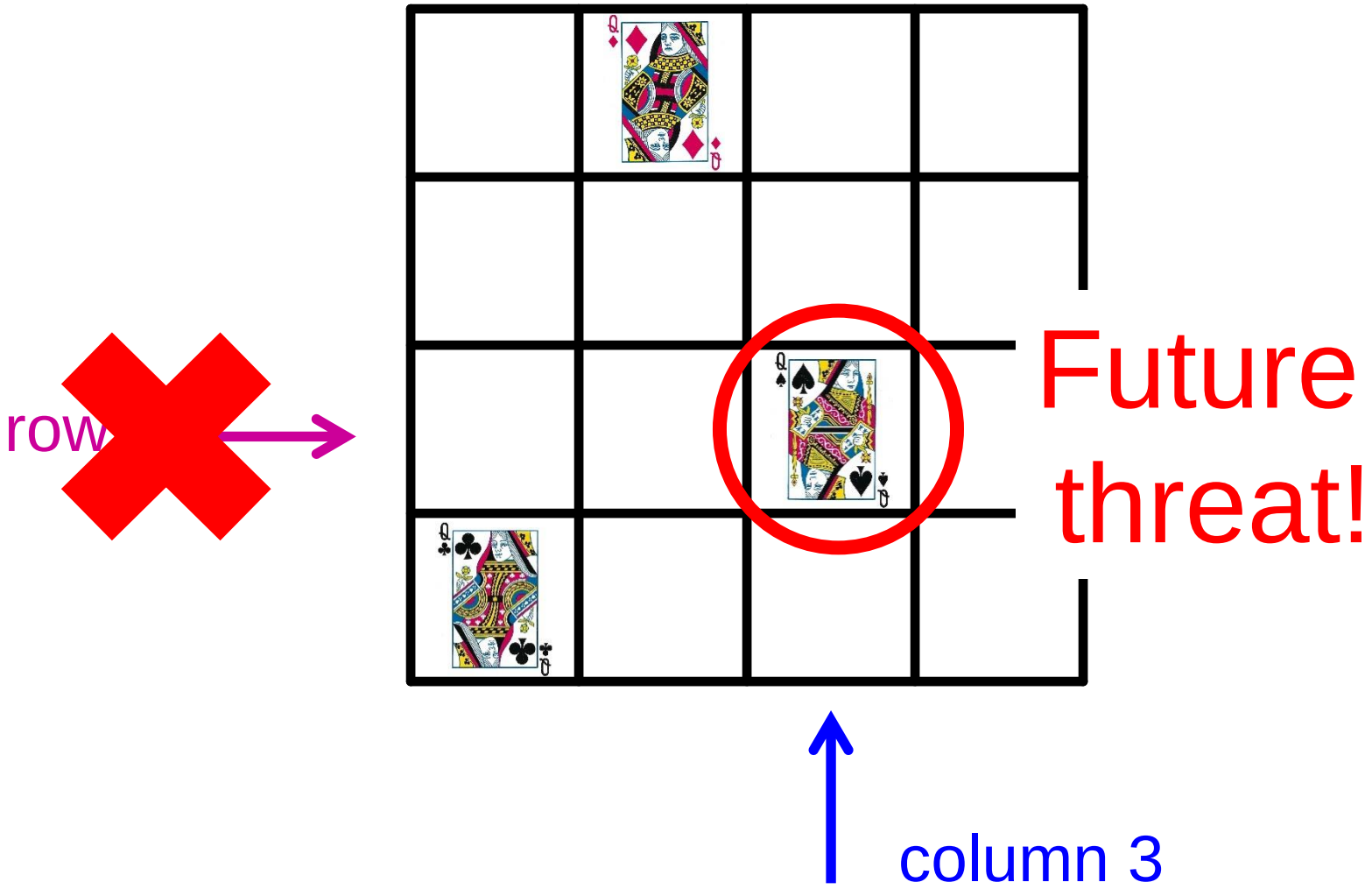
OH NO! We cannot place the fourth queen!

Let's backtrack to the placement of third queen.



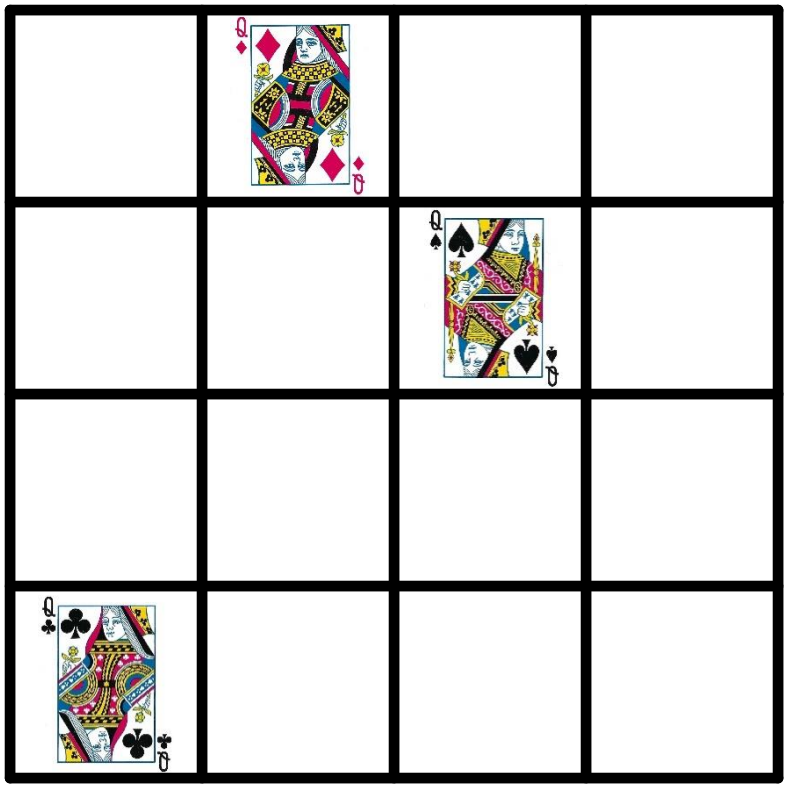
column 3

Let’s backtrack to the placement of third queen.



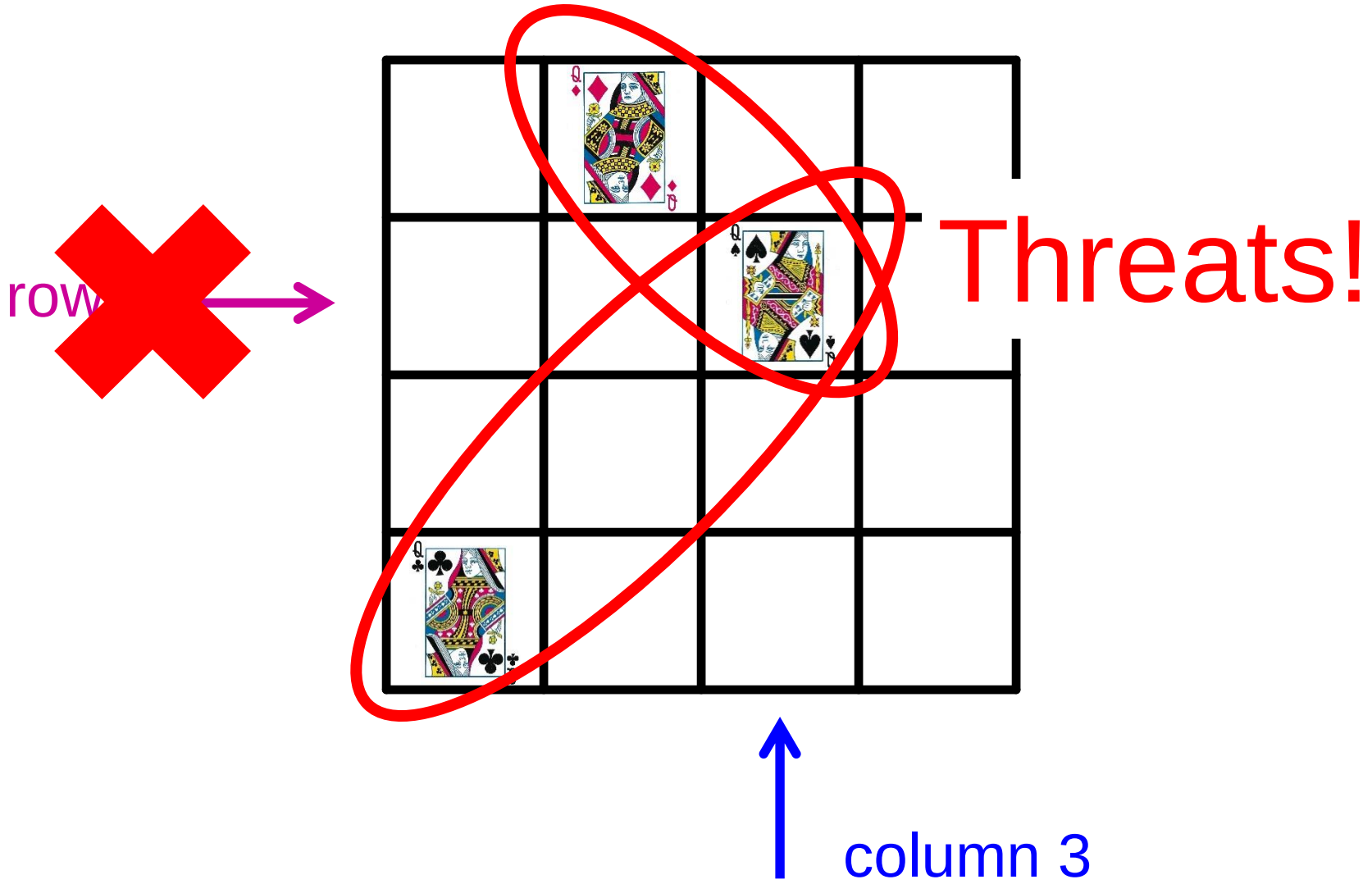
Let's try new placements for the third queen.

row 3 →



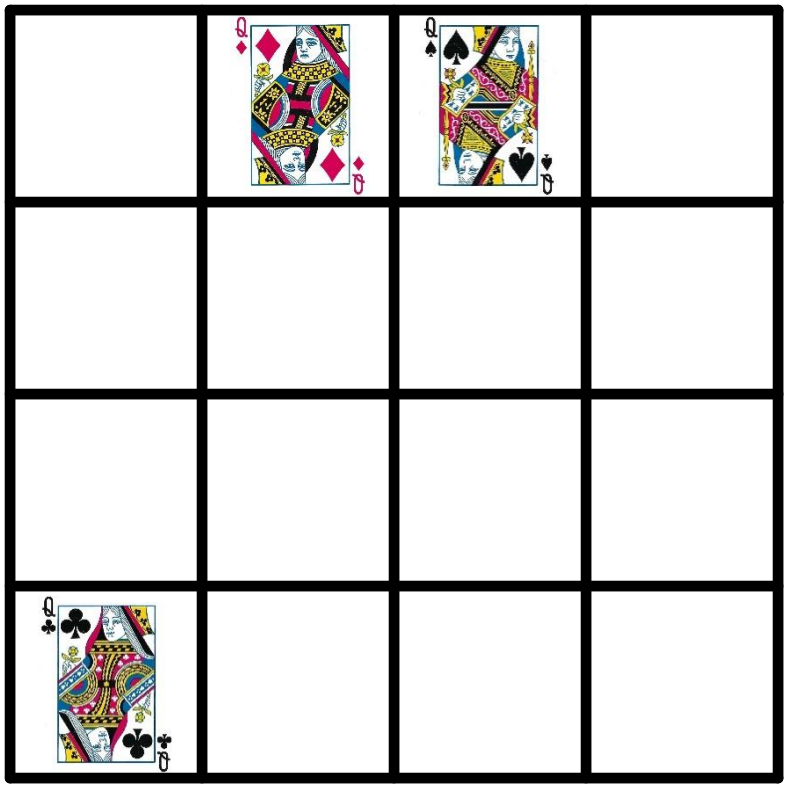
↑ column 3

Let's try new placements for the third queen.



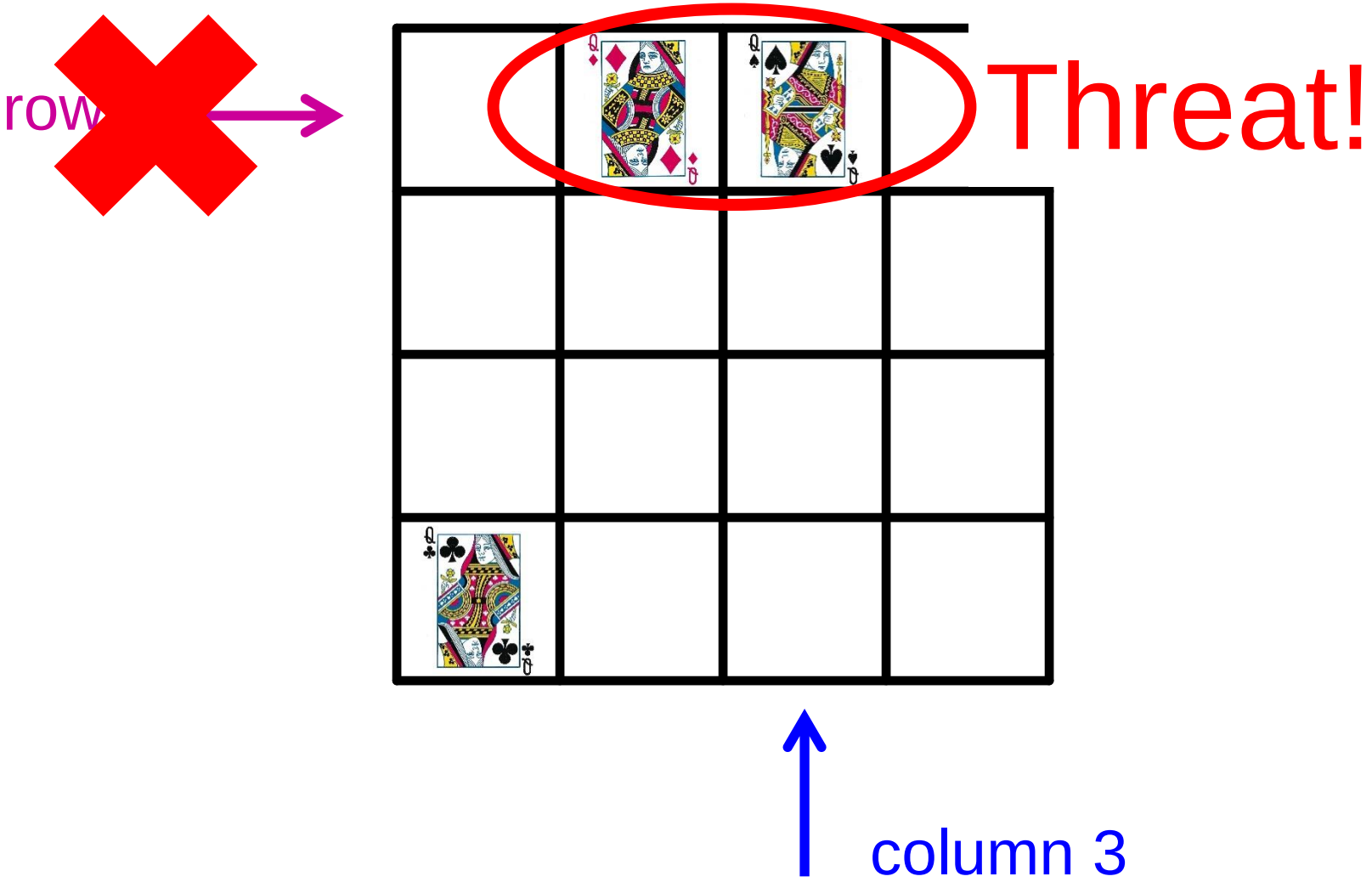
Let's try new placements for the third queen.

row 4 →



↑ column 3

Let's try new placements for the third queen.

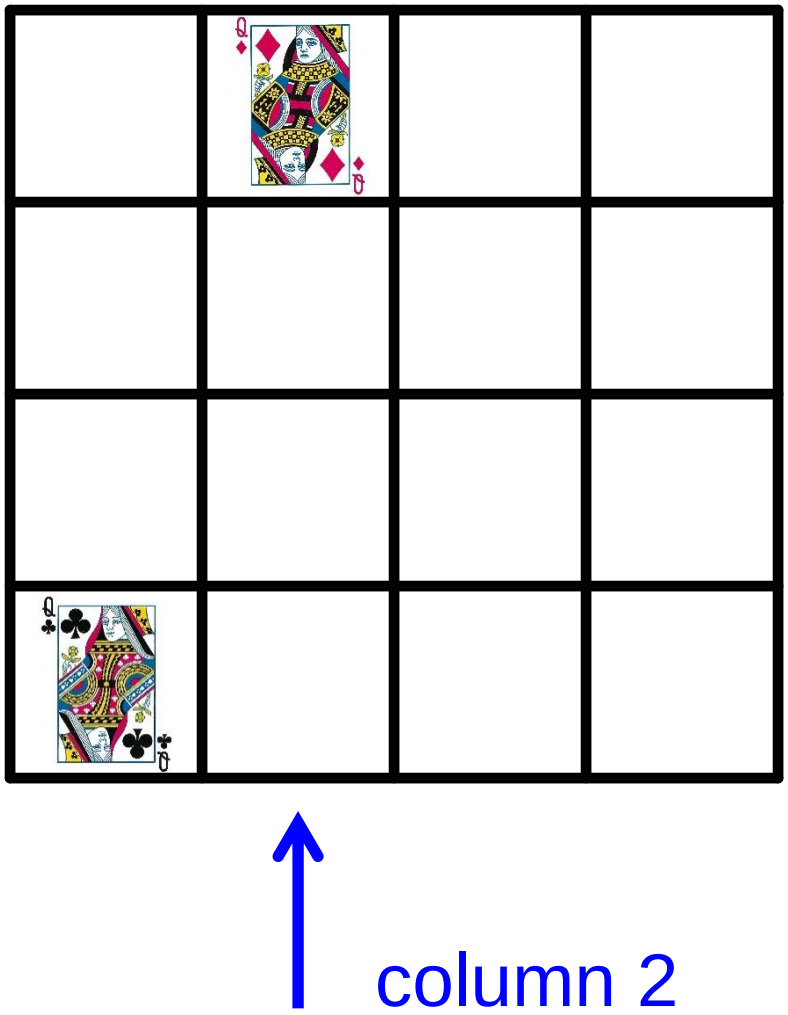


OH NO! Again cannot place the third queen!

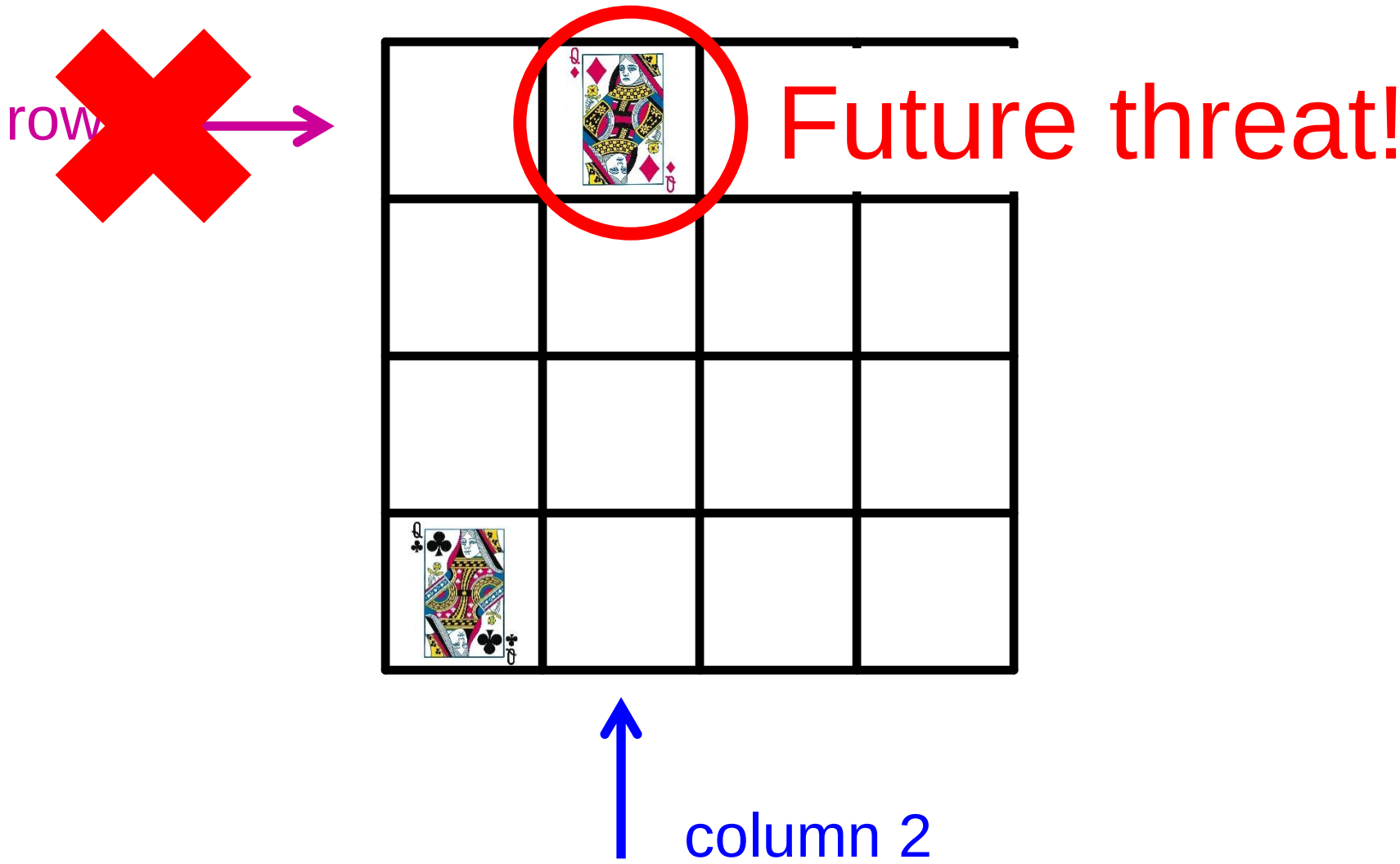
column 3

OH NO! Again cannot place the third queen!

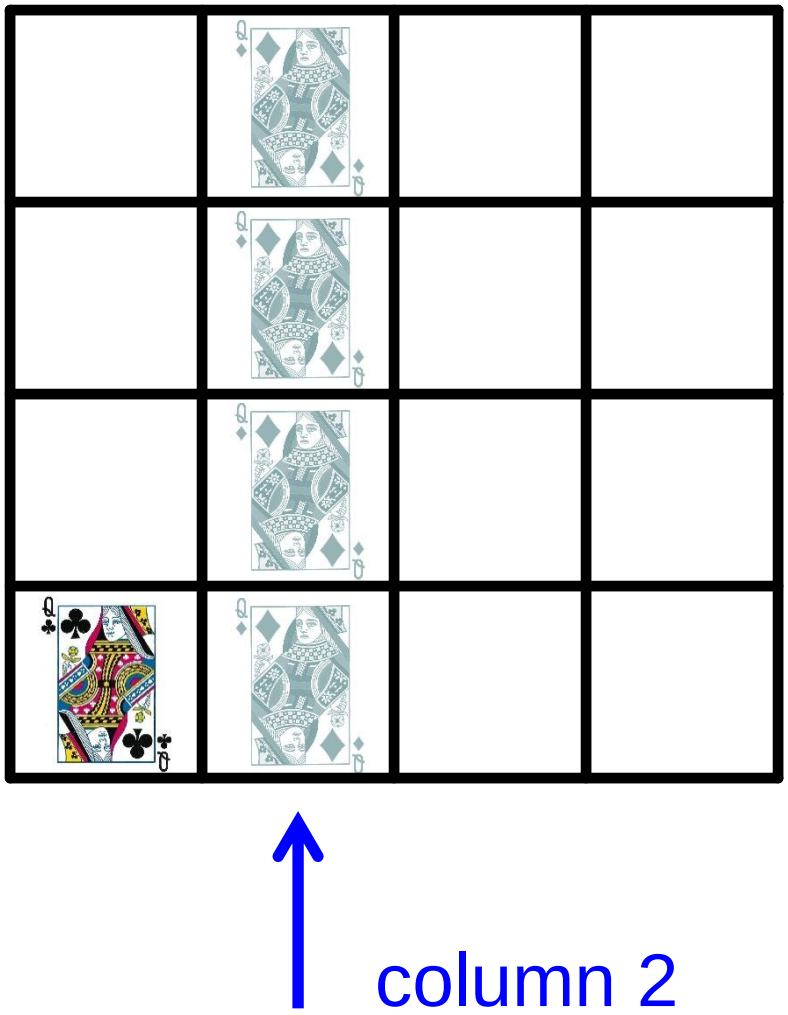
Again backtrack to the placement of second queen.



Again backtrack to the placement of second queen.

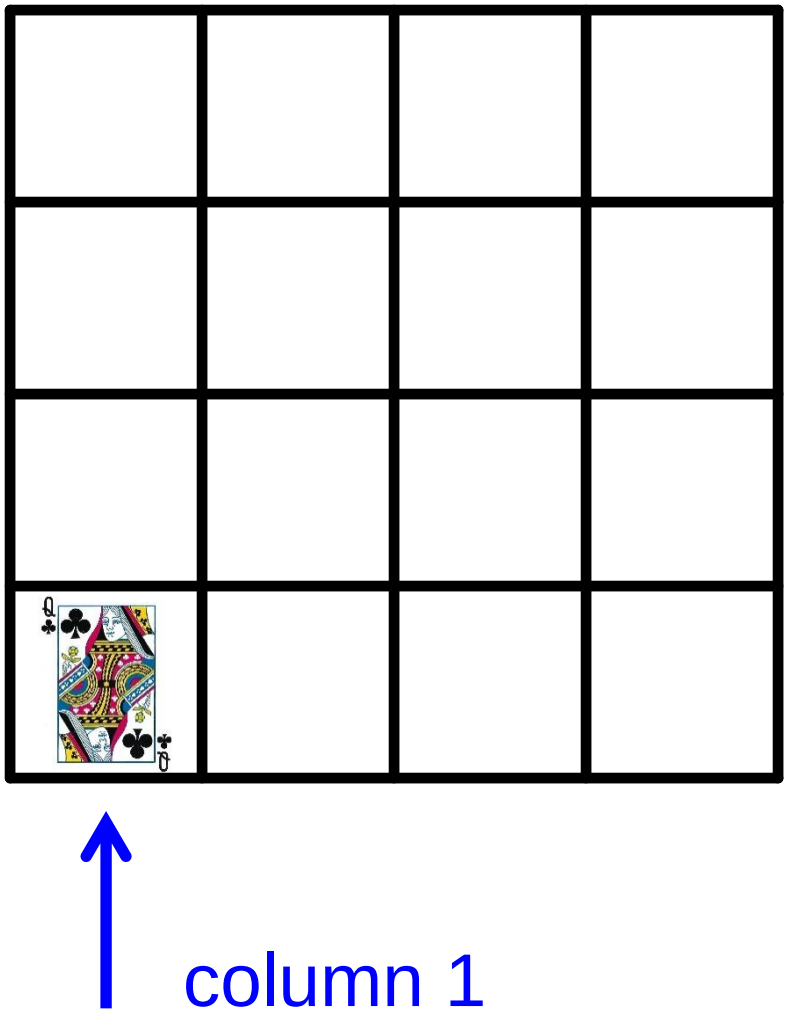


OH NO! We cannot place the second queen!

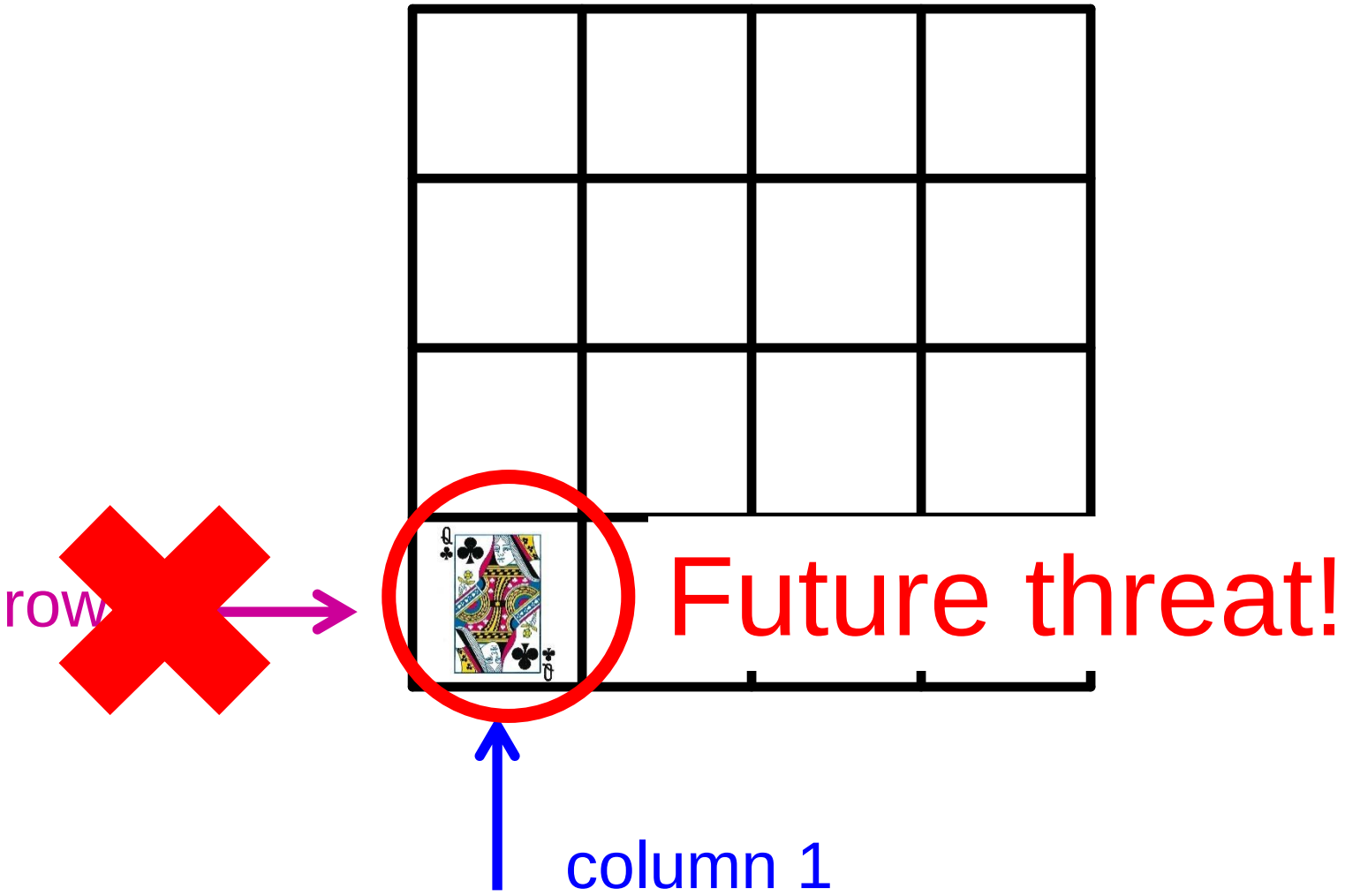


OH NO! We cannot place the second queen!

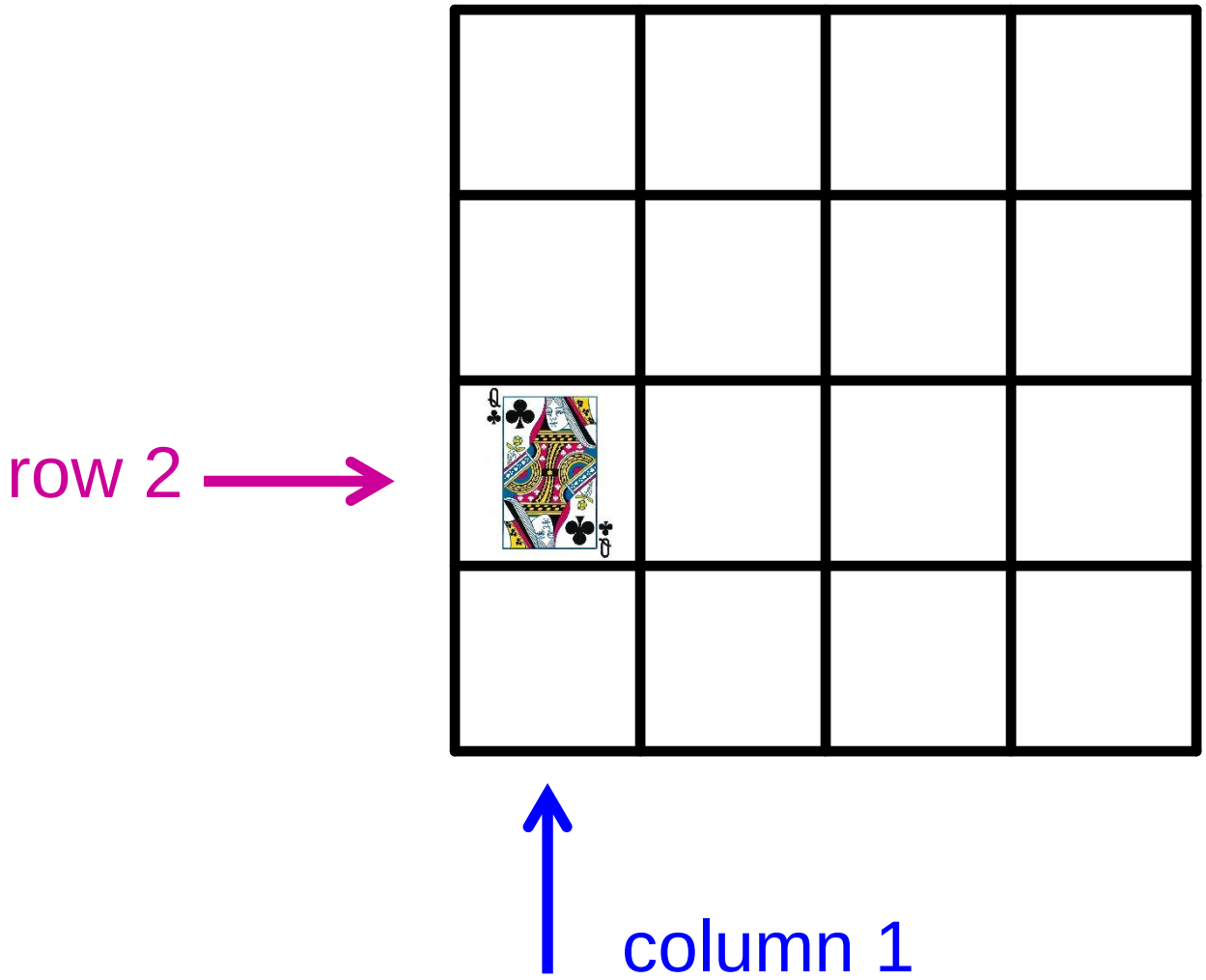
Let's backtrack to the placement of ***first*** queen.



Let's backtrack to the placement of ***first*** queen.

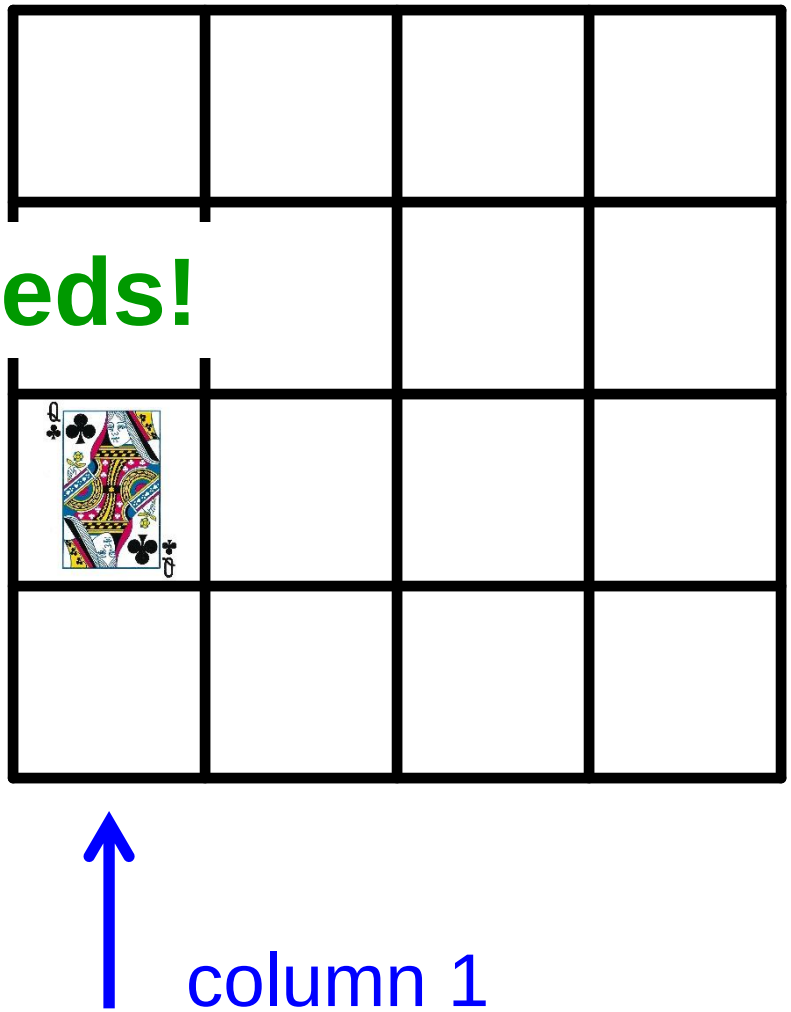


Let’s try a new placement for the first queen.

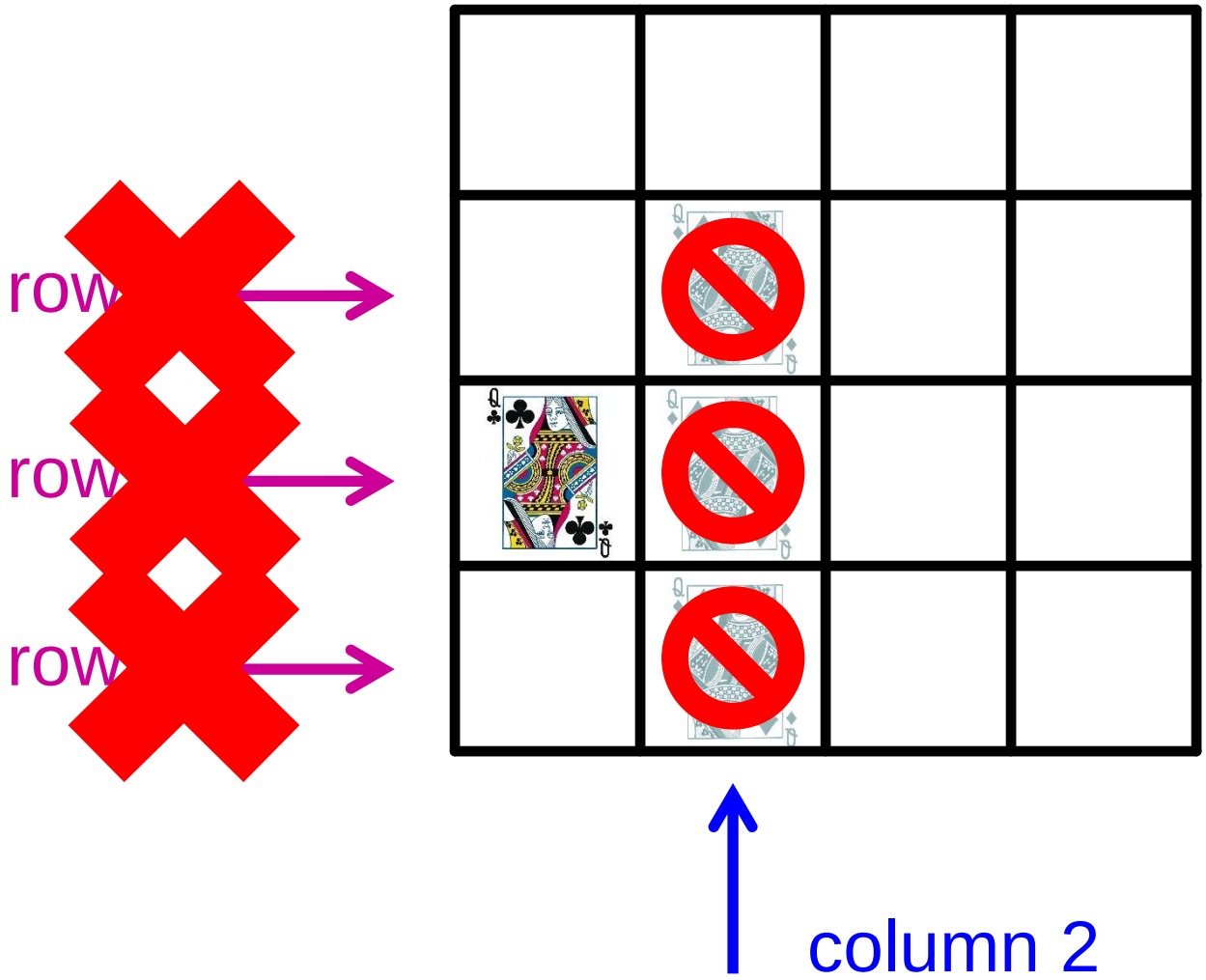


Let’s try a new placement for the first queen.

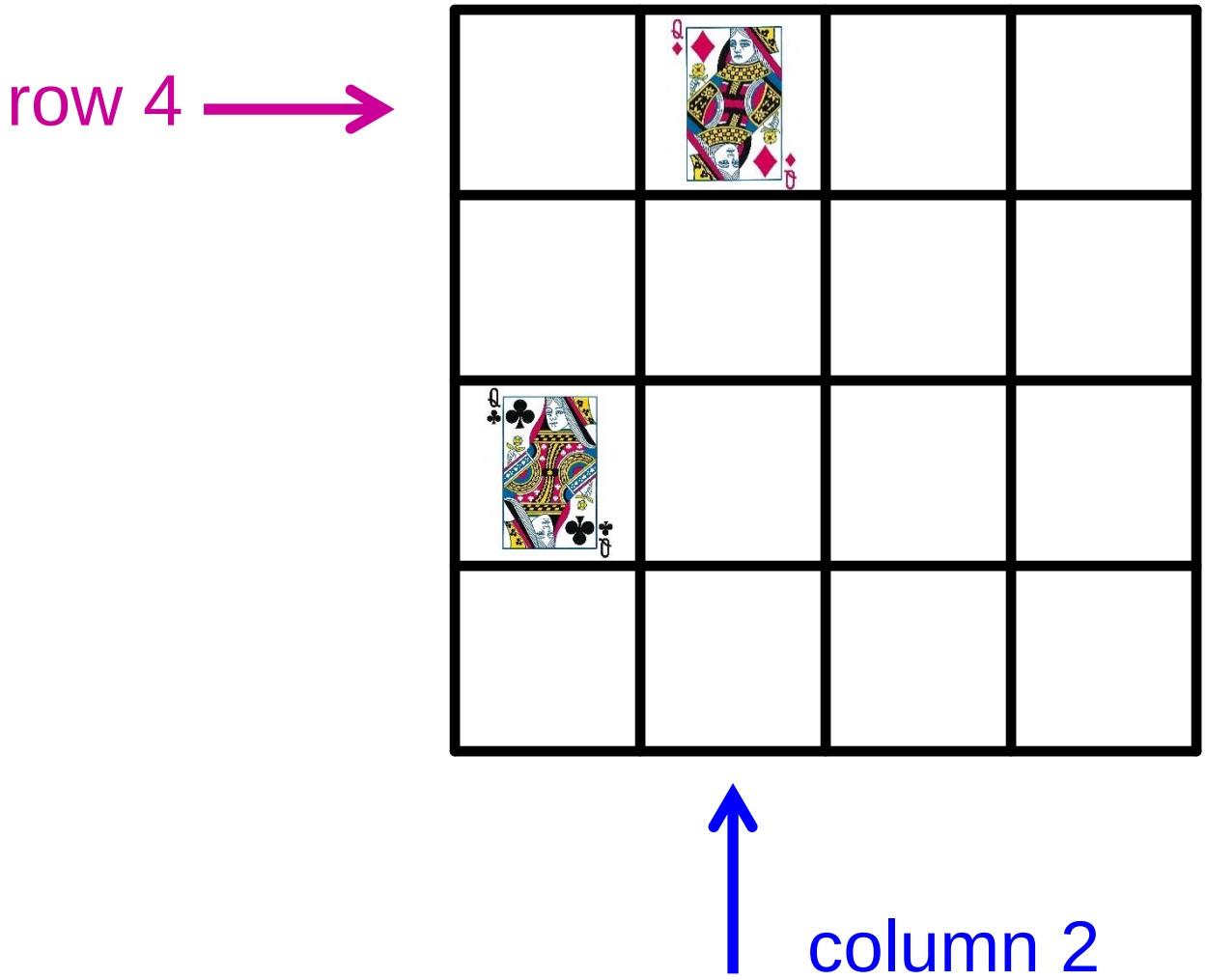
That succeeds!



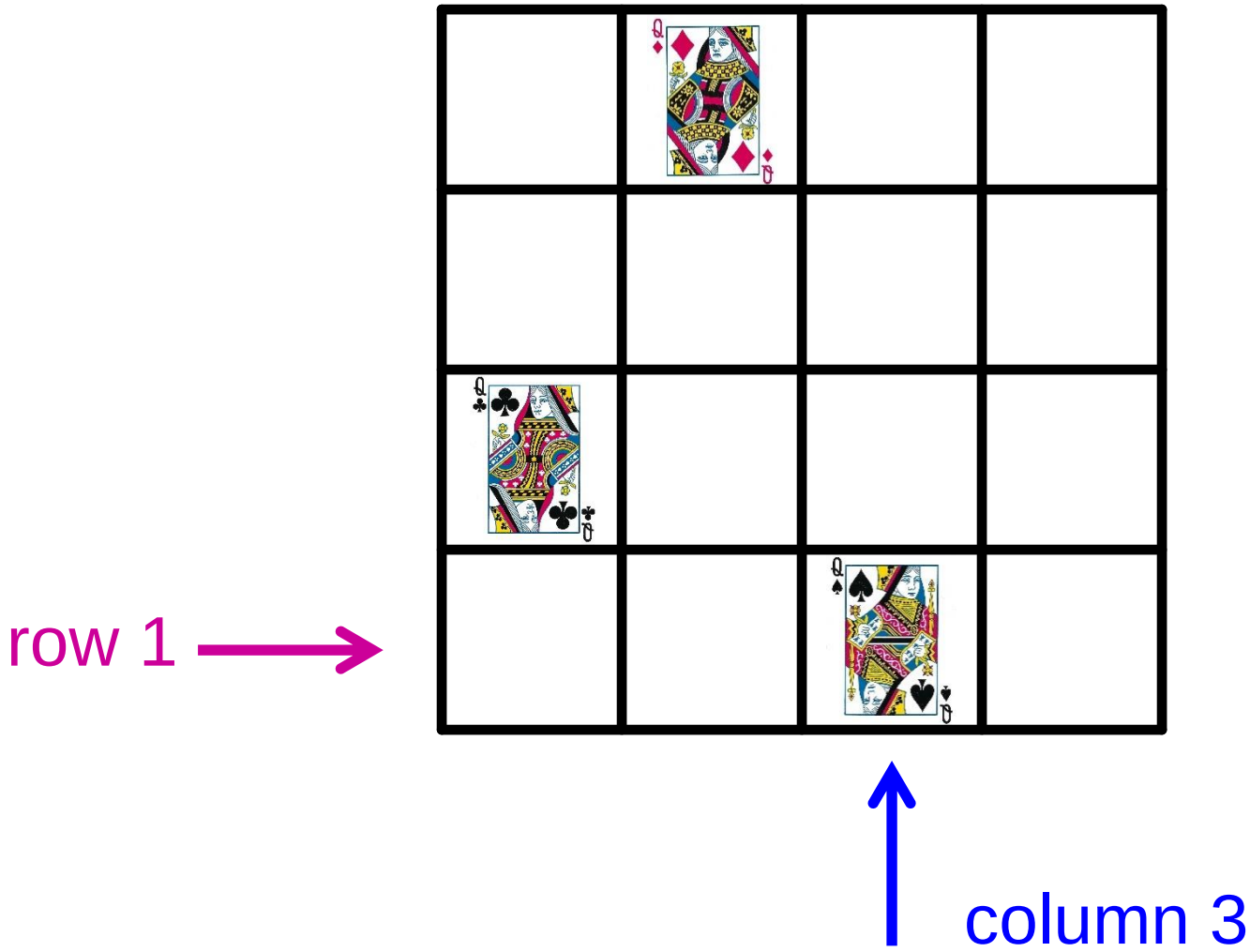
Again add a queen to column 2 by trying different rows:



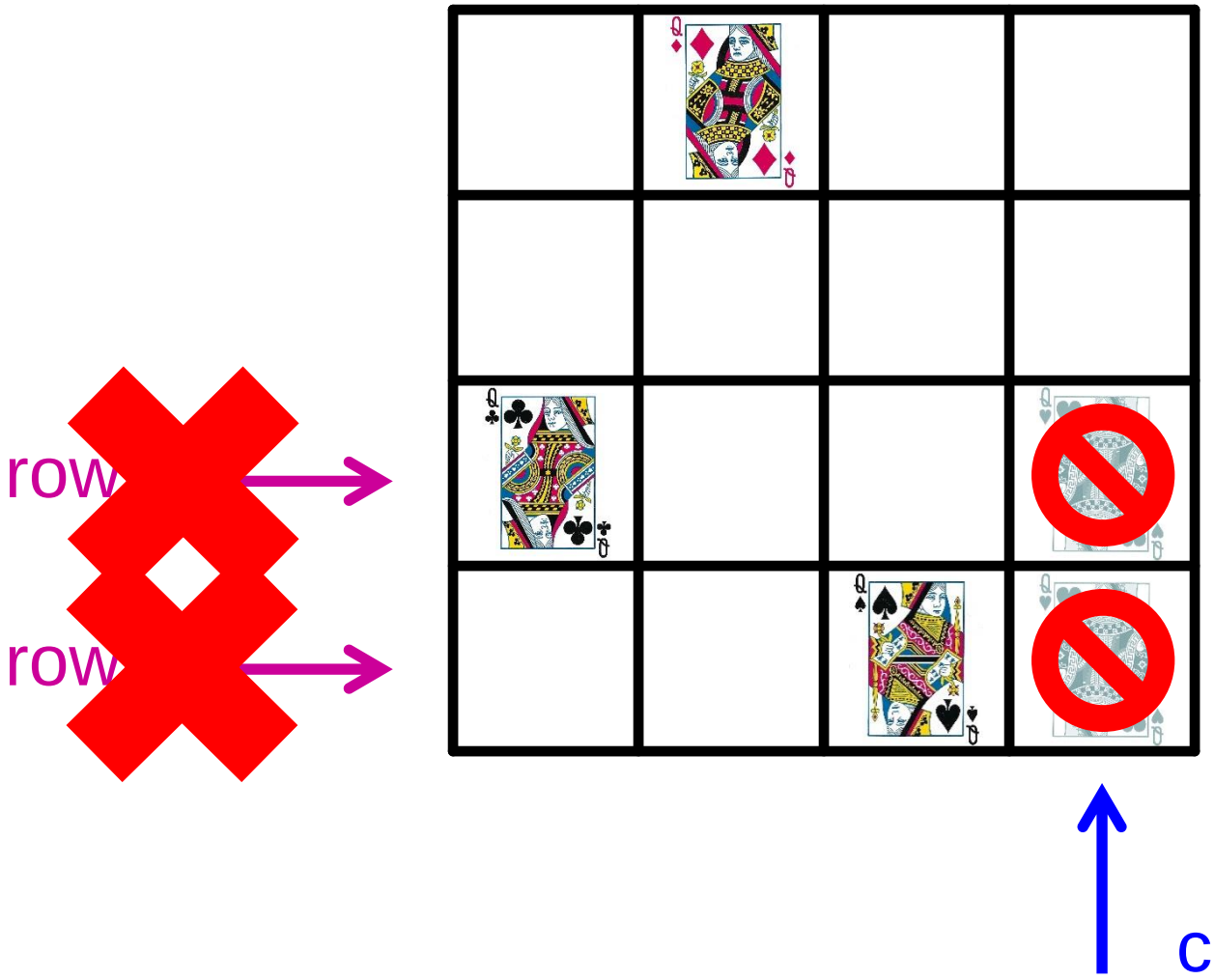
Eventually, placement of **second** queen **succeeds**:



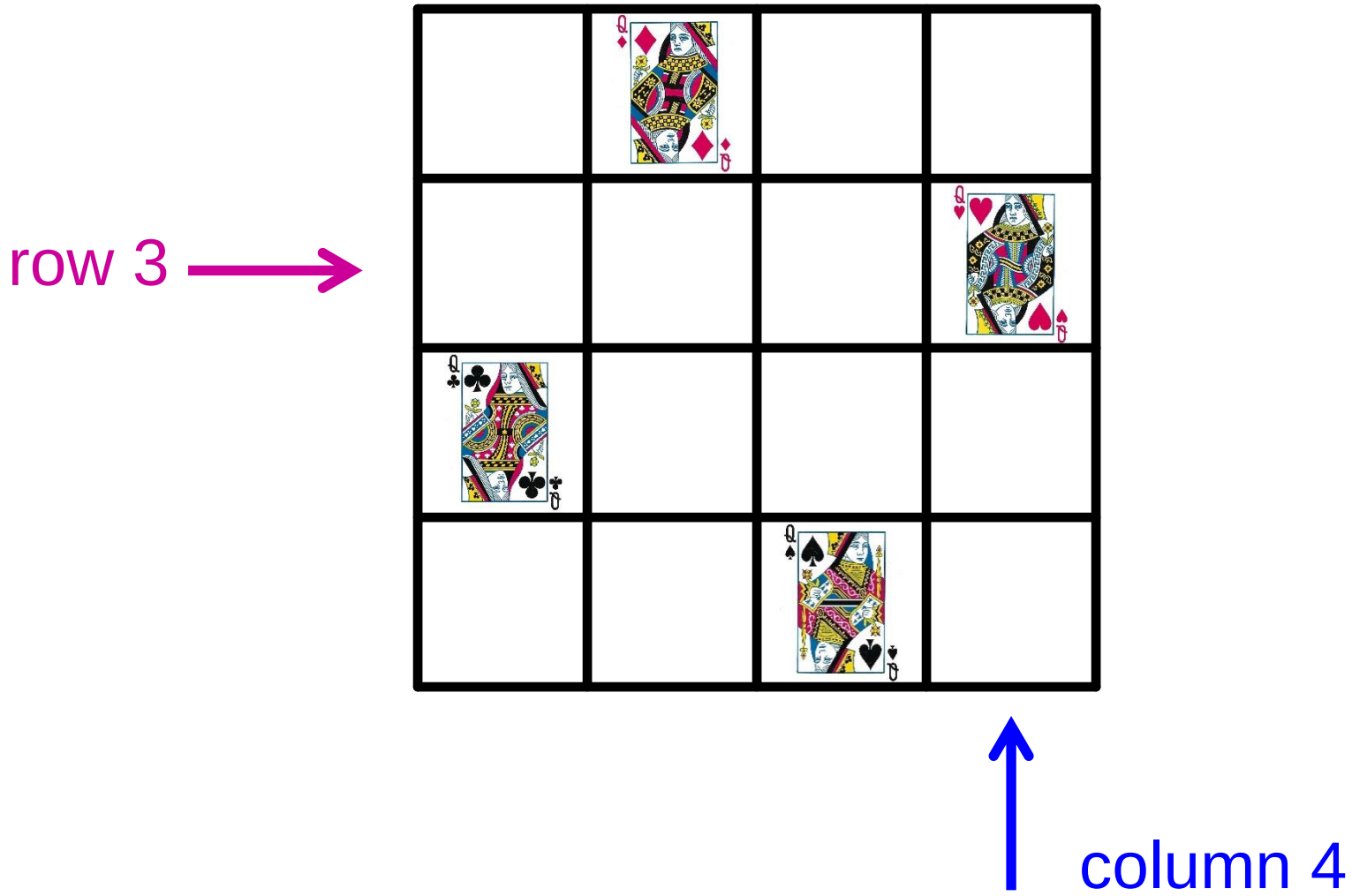
Then placement of **third** queen **succeeds**:



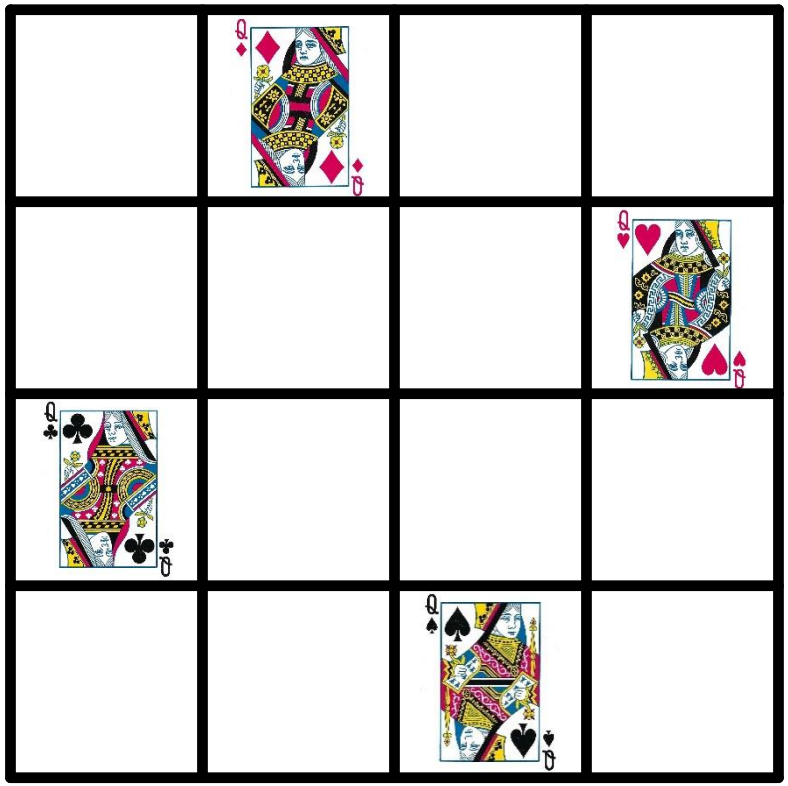
Again add a queen to column 4 by trying different rows:



Eventually, placement of **fourth** queen **succeeds**:



Solution obtained:



Code Overview

- (i,j) refers to i^{th} column & j^{th} row.
- Try to add a queen to column i, given threat-free queen placements in columns $1,\ldots,i-1$.
- Try successive rows, i.e., positions $(i,1),(i,2)\ldots$.
- If position (i,j) is threat-free, place i^{th} queen there and move on to column $i+1$.
- If no position is threat-free in column i, backtrack to column $i-1$, undo the prior placement of a queen in that column and search for a new placement.

```
(* threat : int*int -> int*int -> bool
     Decide whether two queen positions
     threaten each other.
*)

fun threat (x,y) (a,b) =
      x=a orelse y=b orelse x+y=a+b orelse x-y=a-b
```

```
(* threat : int*int -> int*int -> bool *)
fun threat (x,y) (a,b) =
    x=a orelse y=b orelse x+y=a+b orelse x-y=a-b
```

```
(* conflict : int*int -> (int*int) list -> bool
   Decide whether a given queen position
   is threatened by any queen position in a list of
   queen positions.
*)
fun conflict p = List.exists (threat p)
```

```
List.exists : ('a -> bool) -> 'a list -> bool
```

```
(* addqueen : int * int * (int * int) list -> (int*int) list
      try : int -> (int * int) list
   queens : int -> (int * int) list
```

- addqueen (i,n,Q) tries to place all remaining queens on an nxn board, starting in column i, assuming Q describes conflict-free queen placements in columns 1,...,i-1.
- addqueen uses local helper function try. try(j) starts its search from position (i,j).
- queens(n) tries to place all queens on an nxn board.
- These functions raise exception Conflict when unsuccessful.

```
*)
```

```
exception Conflict
fun addqueen (i, n, Q) =
    let
        fun try j =
            (if conflict (i,j) Q then raise Conflict
             else if i=n then (i,j) :: Q
             else addqueen(i+1, n, (i,j) :: Q))
               handle Conflict => if j=n
                                  then raise Conflict
                                  else try(j+1)
    in try 1
    end

fun queens n = addqueen (1, n, [])
```

queens 4 ⟶ [(4,3),(3,1),(2,4),(1,2)]

queens 1 ⟶ [(1, 1)]

queens 2 does not return a value.
Instead, exception Conflict
is uncaught at top level.

Implementation using options

```
(* addqueen : int * int * (int * int) list
                  -> (int * int) list option

   try : int -> (int * int) list option
queens: int -> (int * int) list option

*)
```

```
fun addqueen (i, n, Q) =
    let
        fun try j =
            (case (if conflict (i,j) Q then NONE
                   else if i=n then SOME((i,j)::Q)
                   else addqueen (i+1, n, (i,j)::Q))
              of NONE => if j=n then NONE
                         else try (j+1)
               | result => result)
    in
        try 1
    end

fun queens n = addqueen (1, n, [])
```

Implementation using continuations

```
(* addqueen :  int * int * (int * int) list
                -> (((int * int) list -> 'a)
                -> (unit -> 'a)
                -> 'a

   try : int -> 'a

   queens : int -> (int * int) list option
*)
```

(* Here we have the top-level queens function again return a list option of queen placements. *)

```
fun addqueen (i,n,Q) sc fc =
    let
        fun try j =
            let
                fun fc' () =
                    if j=n then fc() else try(j+1)
            in
                if conflict (i,j) Q then fc'()
                else if i=n then sc((i,j)::Q)
                else addqueen (i+1,n,(i,j)::Q) sc fc'
            end
    in
        try 1
    end

fun queens n =
    addqueen (1,n,[]) SOME (fn () => NONE)
```

More powerful continuations

We will allow success continuations to take failure continuations as arguments.

Doing so increases expressive power.

We can then solve more problems simply by changing continuations slightly.

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* find : ('a → bool) → 'a tree
                      → ('a → (unit → 'b) → 'b)
                      → (unit → 'b)
                      → 'b
*)

fun find p Empty sc fc = fc ()
  | find p (Node(l,x,r)) sc fc =
      let fun fcnew () =
              find p l sc (fn () ⇒ find p r sc fc)
      in if p(x) then sc x fcnew
         else fcnew()
      end
```

The success continuation receives element **`x`** **and** the failure continuation (**`fcnew`** says what to do if **`p(x)`** had been **`false`**).

```
fun even n = (n mod 2 = 0)

(* find first even integer encountered in pre-order traversal *)
fun findfirst T =
      find even T (fn x => fn f => SOME(x))
                  (fn () => NONE)

(* accumulate list of all even integers *)
fun findall T =
      find even T (fn x => fn f => x :: f())
                  (fn () => [])

(* count all the even integers *)
fun count T =
      find even T (fn x => fn f => 1 + f())
                  (fn () => 0)
```

That is all.

Please enjoy Spring Break.

See you the Tuesday after, when we will talk about regular expressions.