Programming Abstractions

Lecture 15: Backtracking

Stephen Checkoway

chris
@chrsnjk

I have a good backtracking joke.
Um, no, actually it is a bad one.

11:31 AM · Jul 28, 2020

21 See chris's other Tweets

Backtracking

You've seen backtracking before

Anagram lab in CS 150!

- oberlin student:
  let none disturb
  run no bed titles
  let us not rebind
  trust line on bed
  but not red lines
  bound in letters
  let in; runs to bed

Backtracking

A method to search for all possible elements in the solution space of many problems

- Not efficient: often exponential time

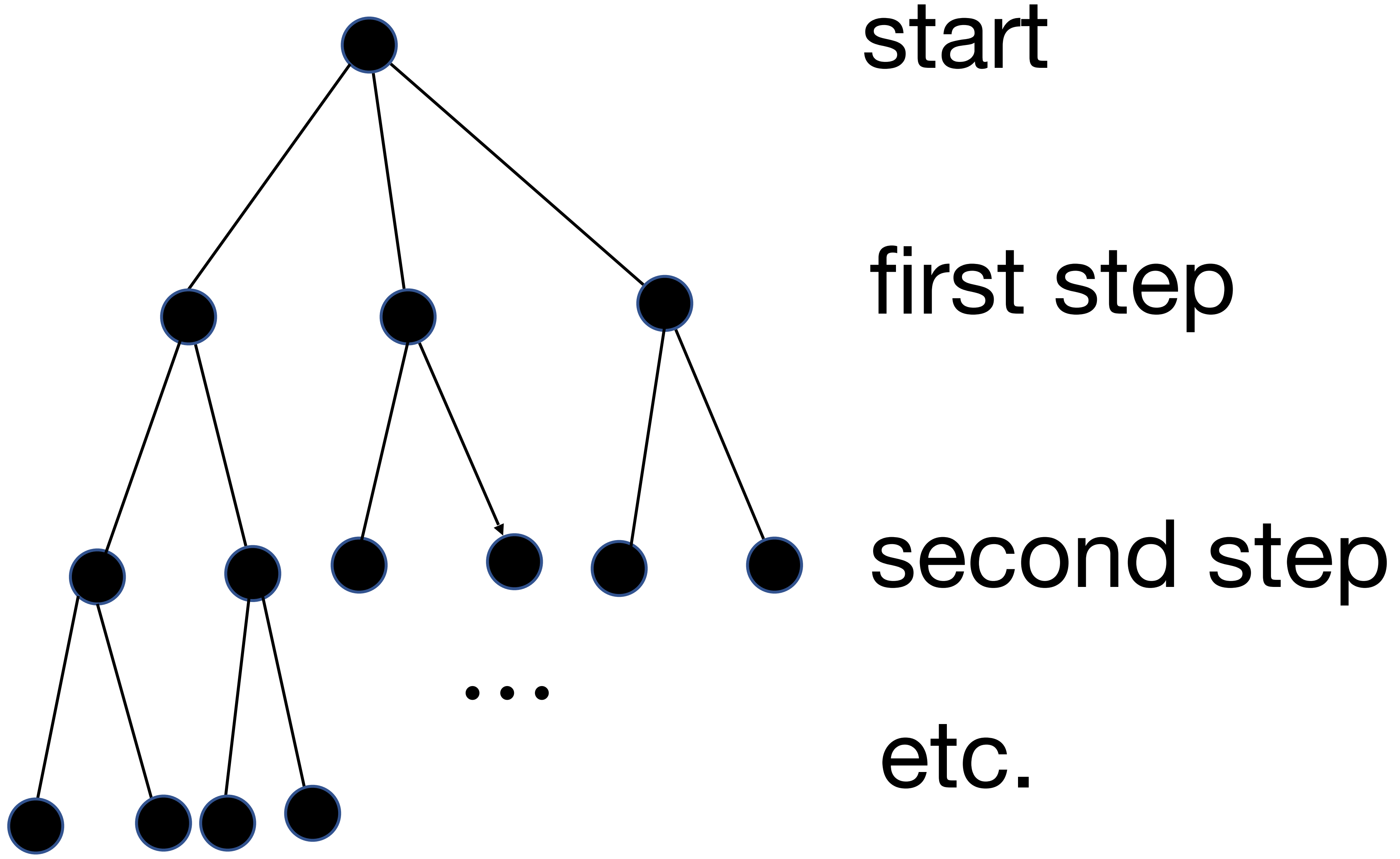
‣ Thus it only works on small problems

+ Fairly easy to implement for a wide class of problems

Types of problems

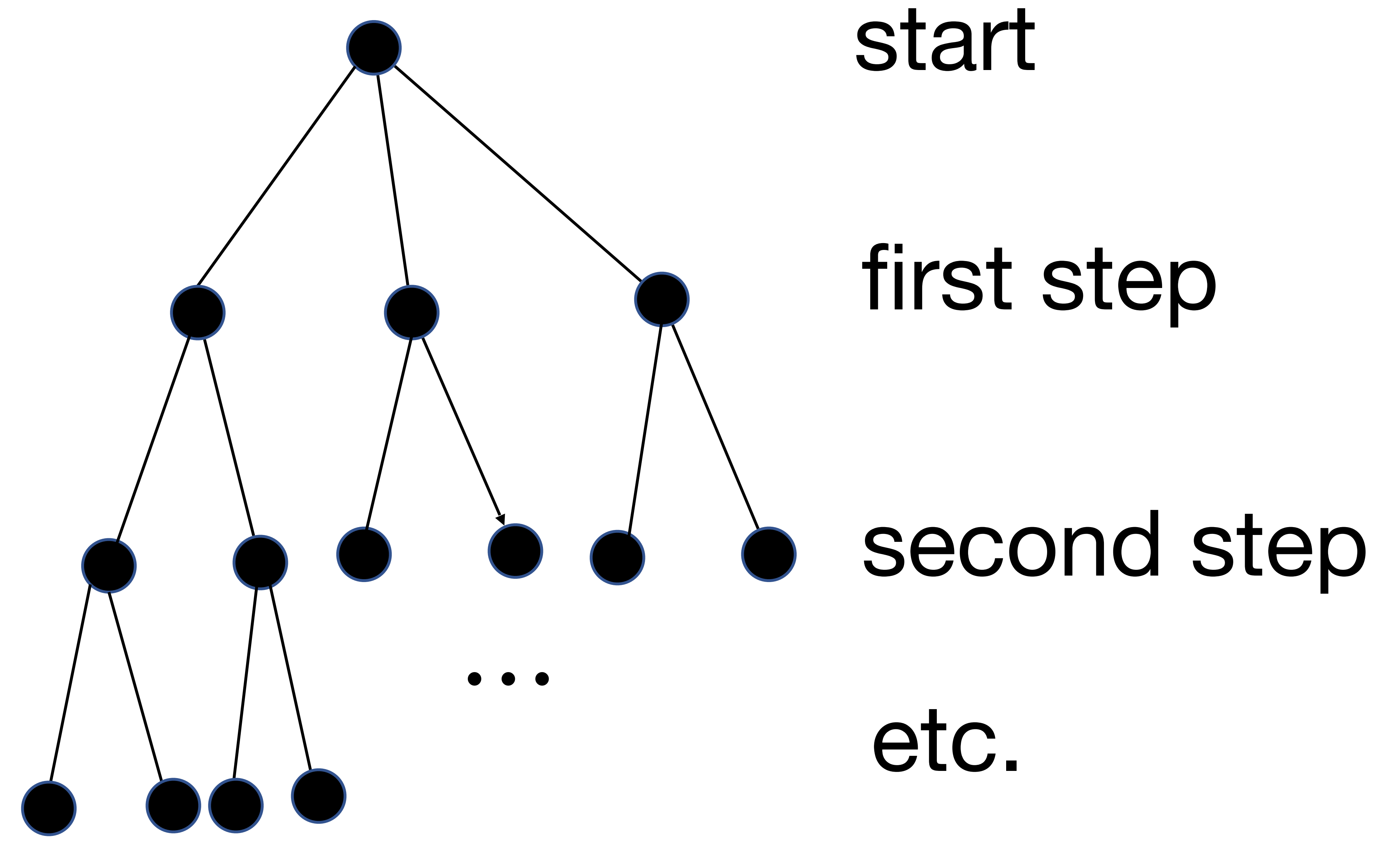
To apply backtracking, the problem needs to have solutions that can be built one step at a time

The solution space for such problems forms a tree



Strategy for solving

- Choose a step to take
- If the chosen step cannot possibly lead to a valid solution, back up and make a different choice
- Repeat this process until a complete solution is found or all possibilities have been exhausted



Examples you've seen before

In the CS 150 Anagrams lab

- Each step consisted of trying to make a word out of the remaining letters by looking through the words of a dictionary
- If all letters couldn't be used to make words, you backed up and made different choices

In CS 151, you solved maze using stacks and queues

- Each step consisted of picking a new cell of the maze to explore
- If you got stuck, you backed up

n-queens

A famous problem solvable via backtracking

- Place n chess queens on an $n \times n$ chessboard such that no two queens are in the same row, same column, or same diagonal

One step of a solution consists of picking a row for a queen in a given column

So start with the first column, pick a row

Then move to the next column and pick a row

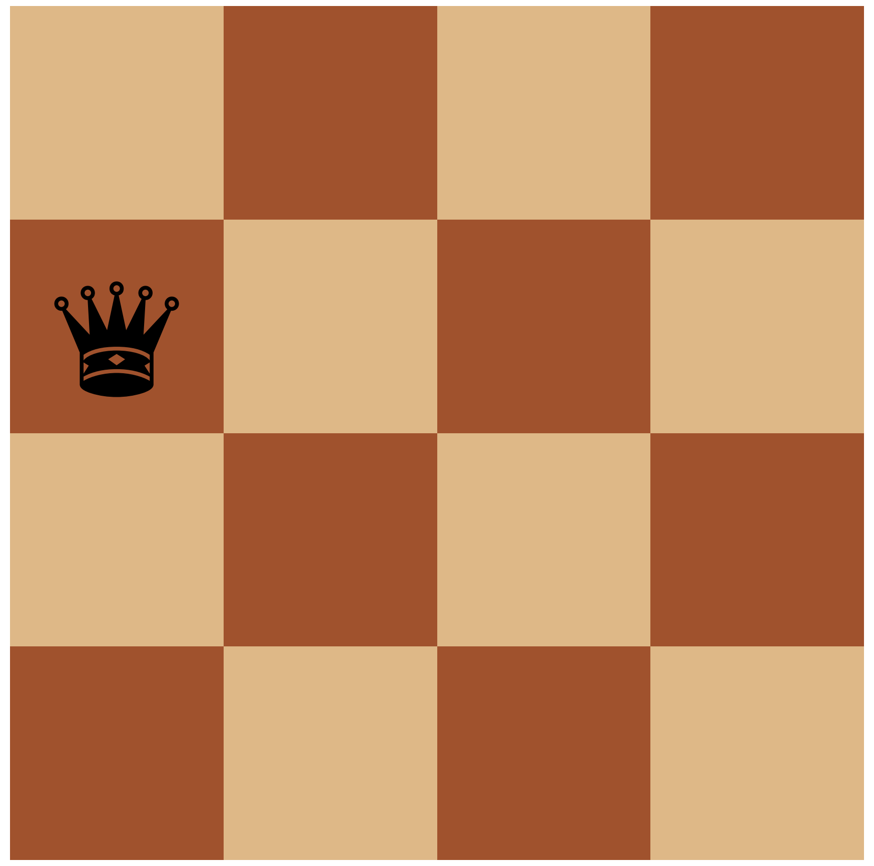
If the partial solution is not valid, backtrack

Repeat until you have a valid solution

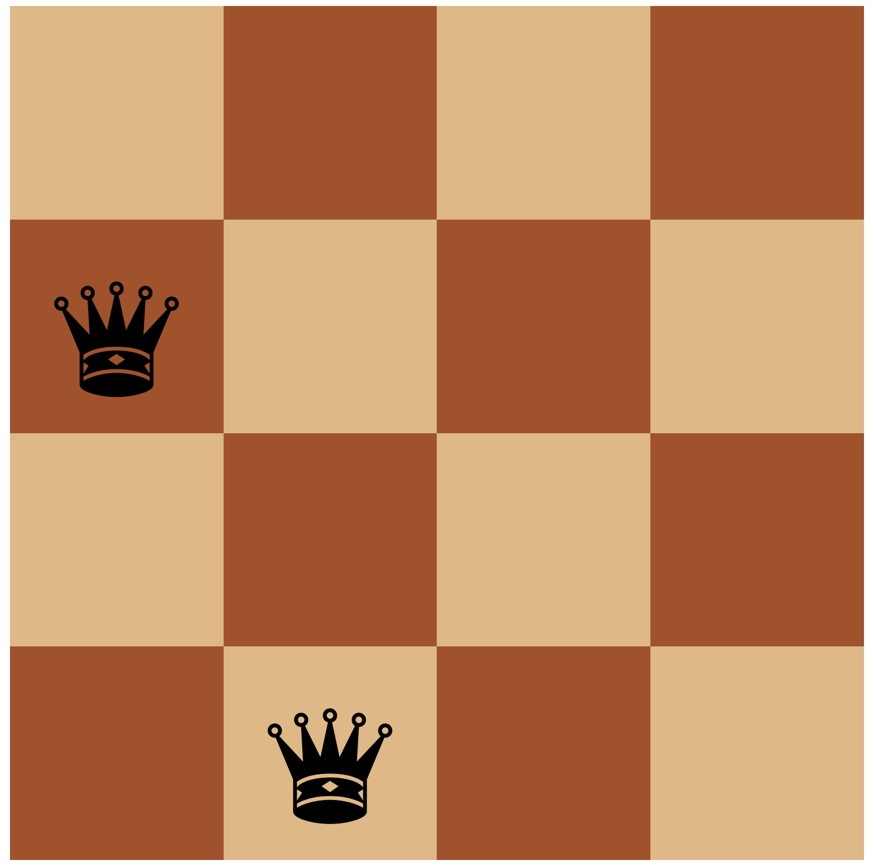
Example: $n = 4$

(Backtracking steps omitted)

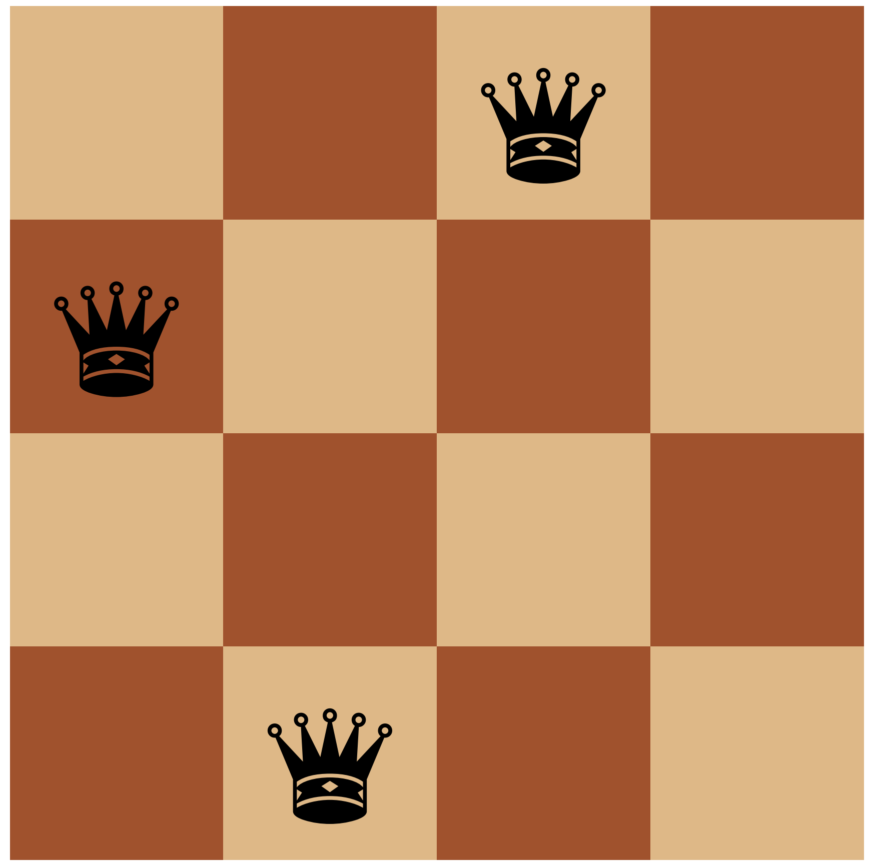
Step 1



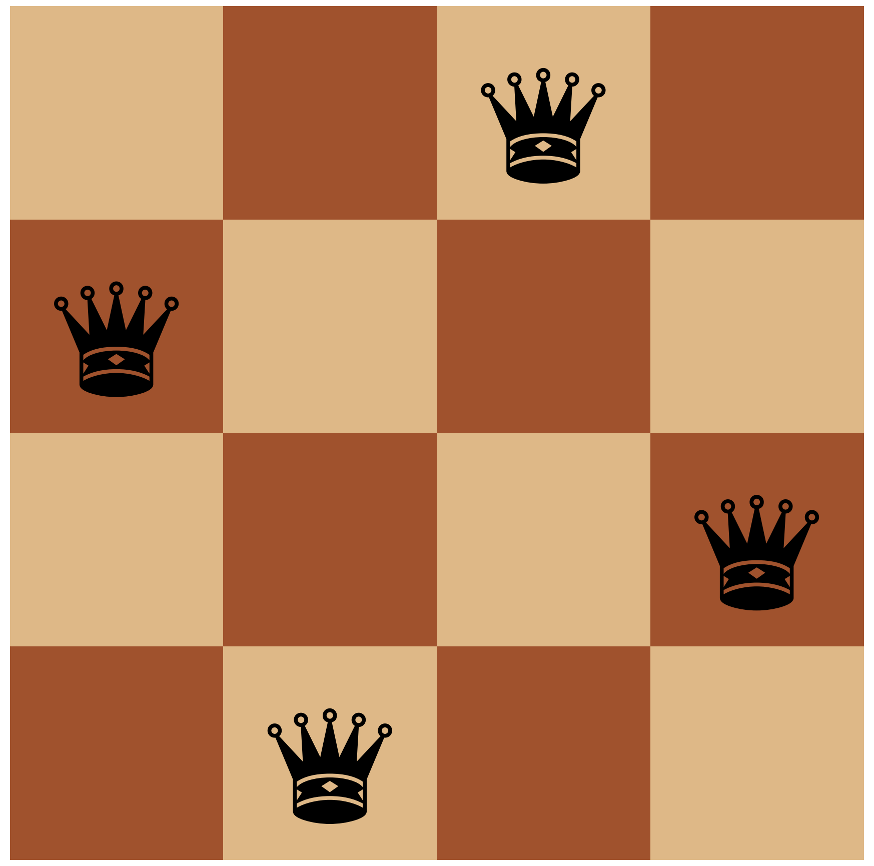
Step 2



Step 3



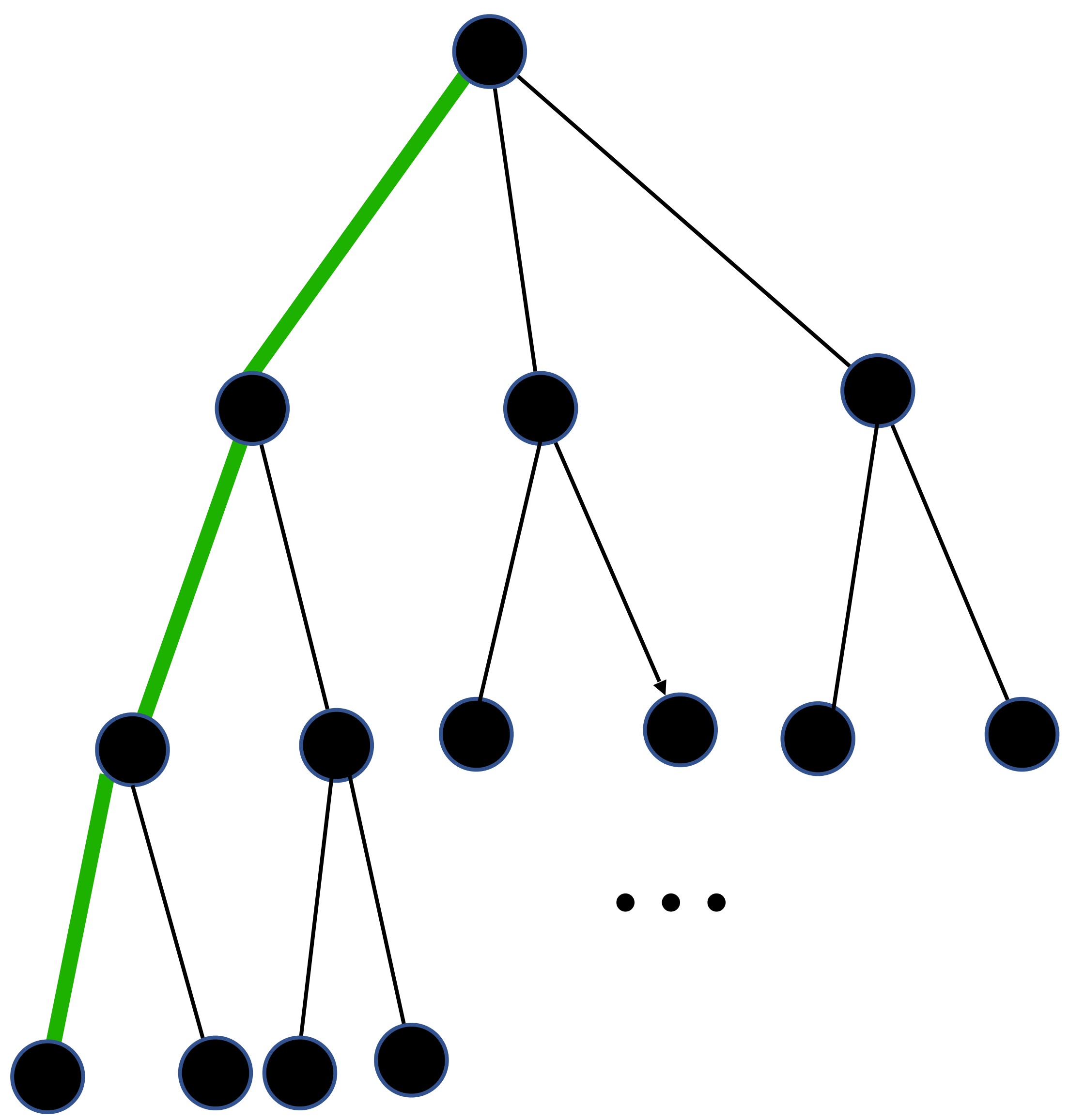
Step 4



Backtracking as search

Backtracking performs a depth-first search through the solution space

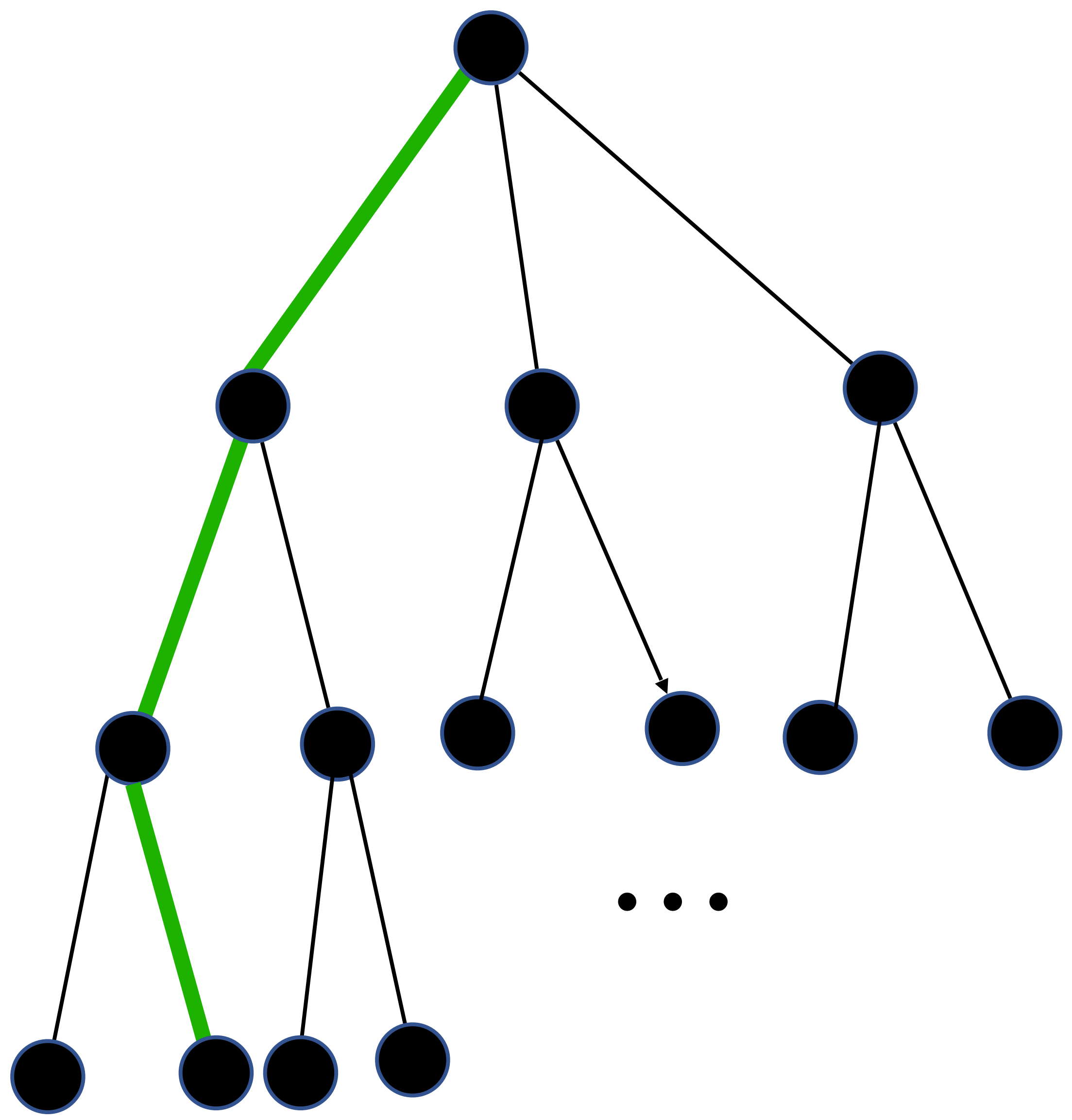
- It tries the first possible value for the first step
- Then the first value for the second step
- And so on



- If this is a valid solution, we're set!

Backtracking as search

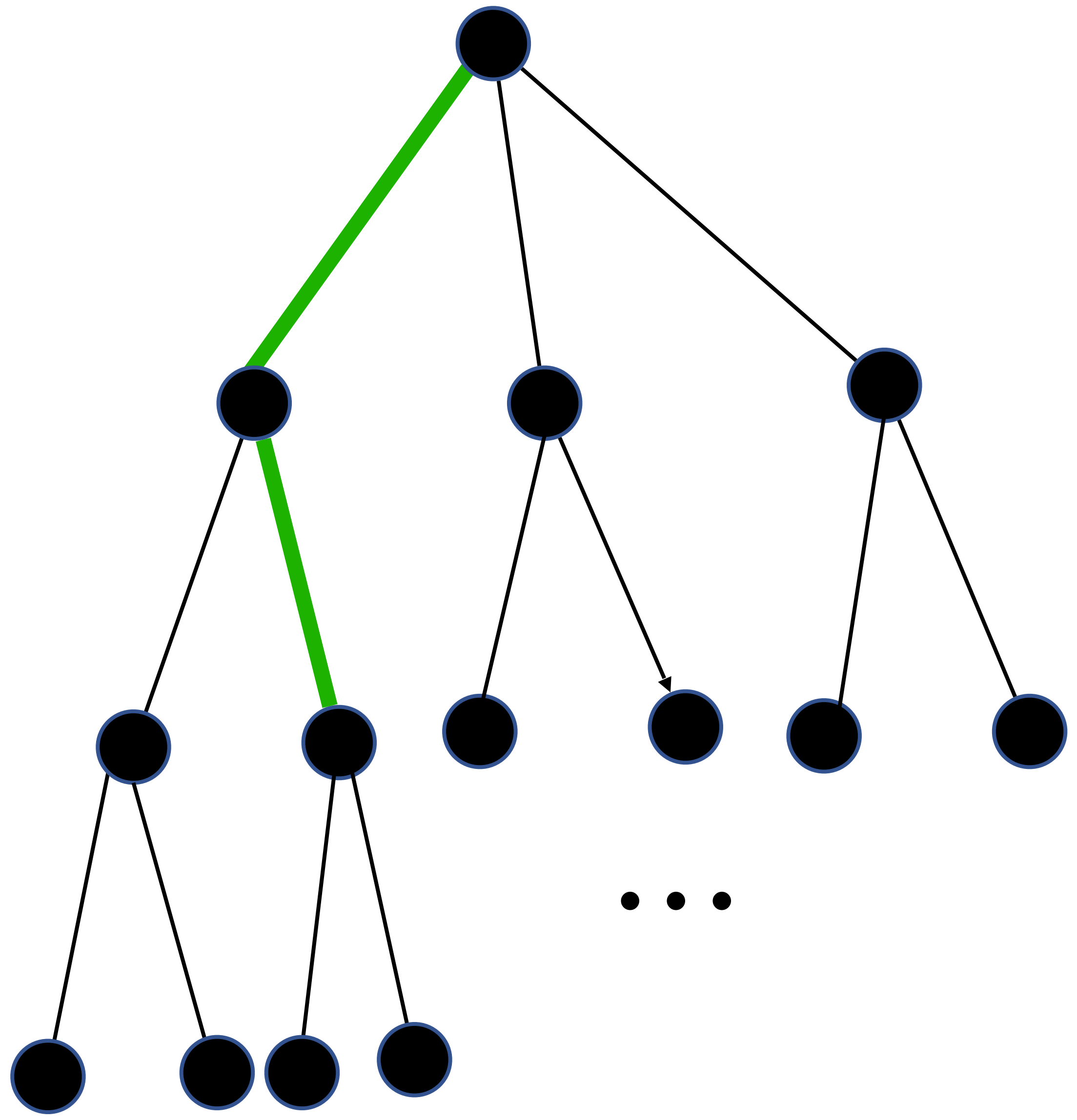
If it's not a valid solution, we back up and make a different choice



Backtracking as search

Suppose this isn't a valid solution so now we're out of options for the third step

We need to make a different second choice



Repeat this until we have a valid solution or none exist

Speeding things up

Backtracking isn't efficient but we can do better than trying every possible value

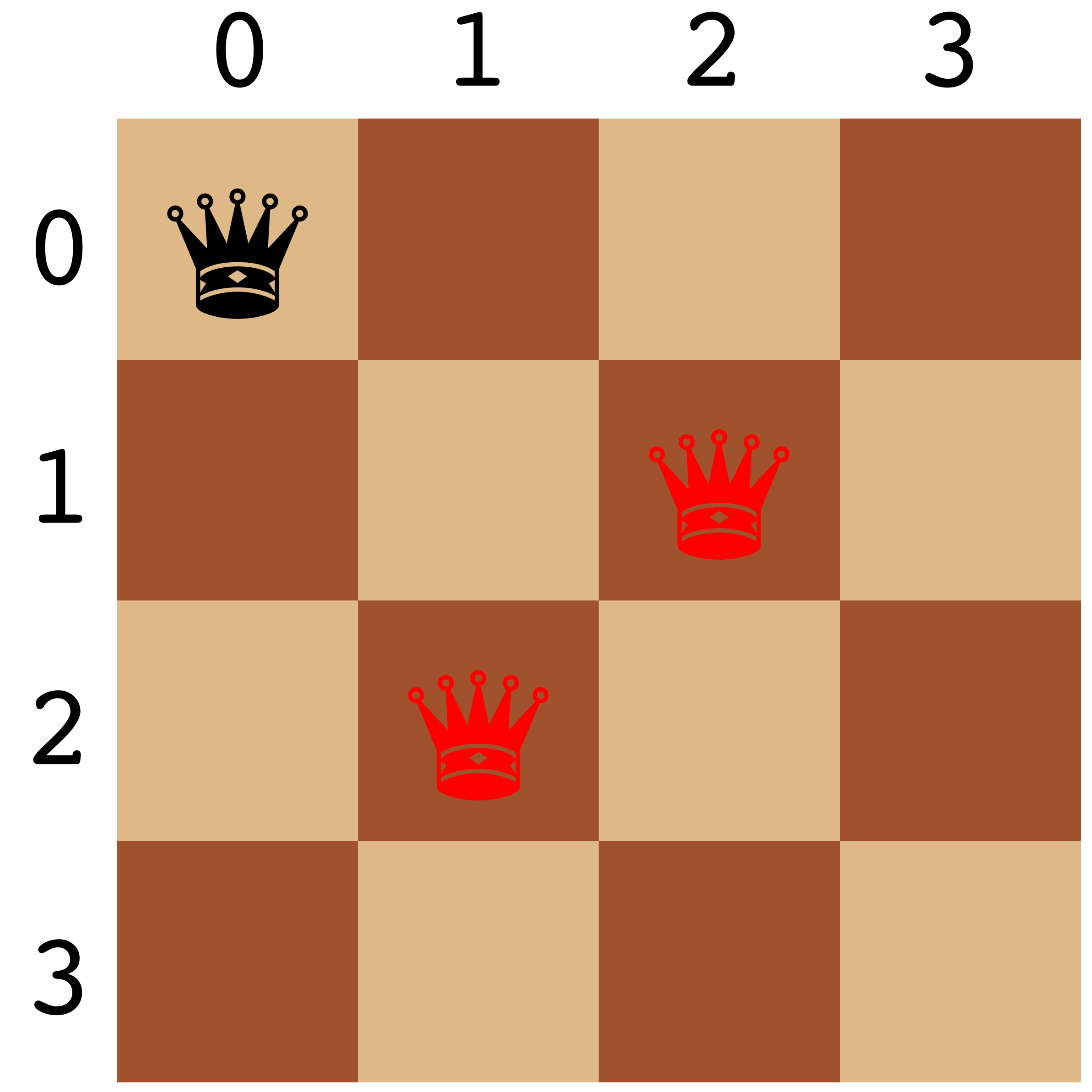
In many cases, we can test if a partial solution is *feasible*

- If so, continue as before
- If not, move on to the next (or backtrack) immediately rather than waiting until the whole subtree has been explored

We can do this with n-queens

- As soon as a partial solution contains two queens in the same row or on the same diagonal, it moves on to the next choice or backtracks

Imagine we're solving 4-queens by picking a row for each column in turn. For column 0 (from left to right), we've selected row 0 (from top to bottom), for column 1, we've selected row 2, and for column 2, we've selected row 1.



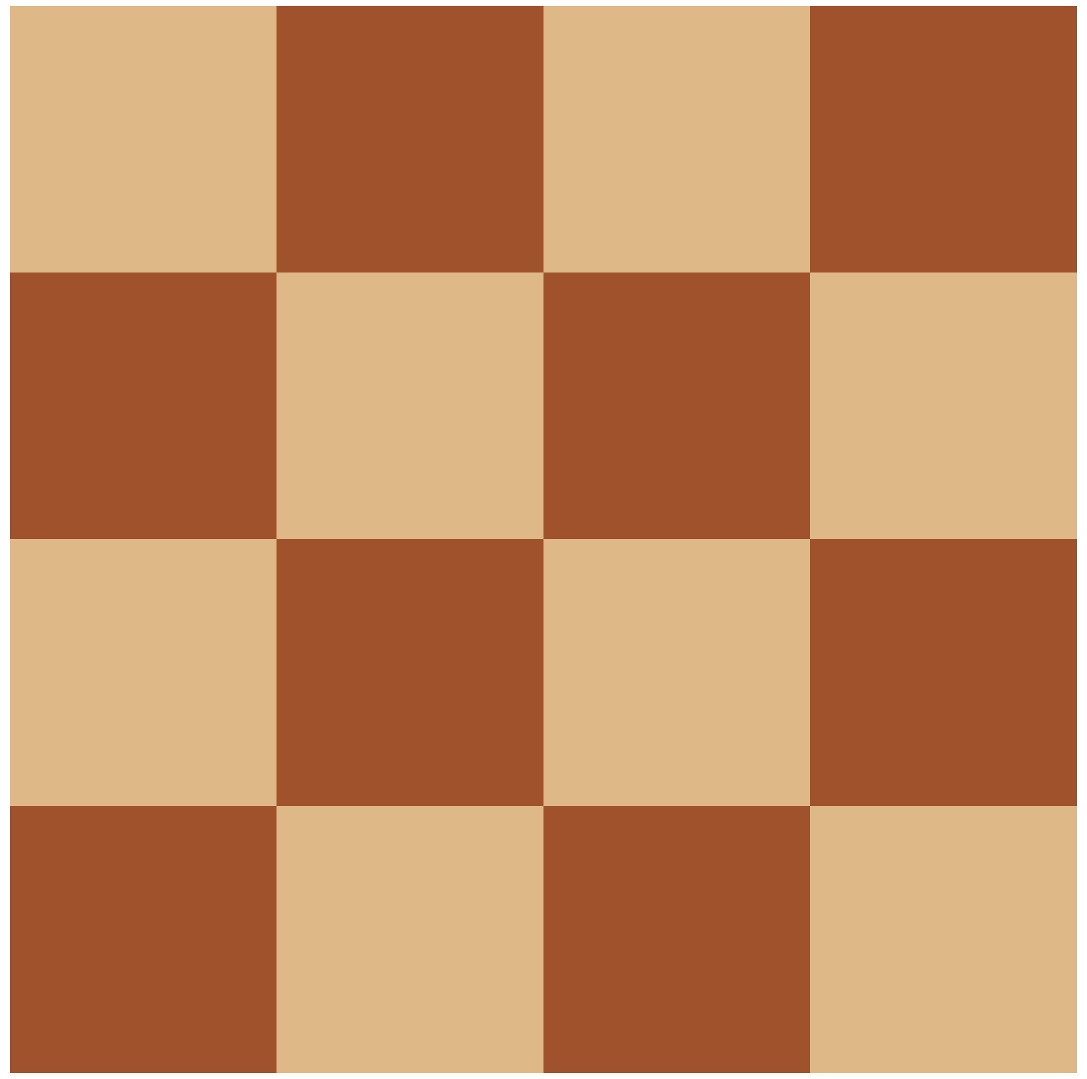
At this point, we should either pick the next row for column 2 (i.e., row 2) or backtrack rather than picking a row for column 3. Why?

A. The partial solution is not feasible: No choice for column 3 will be a valid configuration of queens

B. None of other choices for column 2 will work so we need to make a different choice

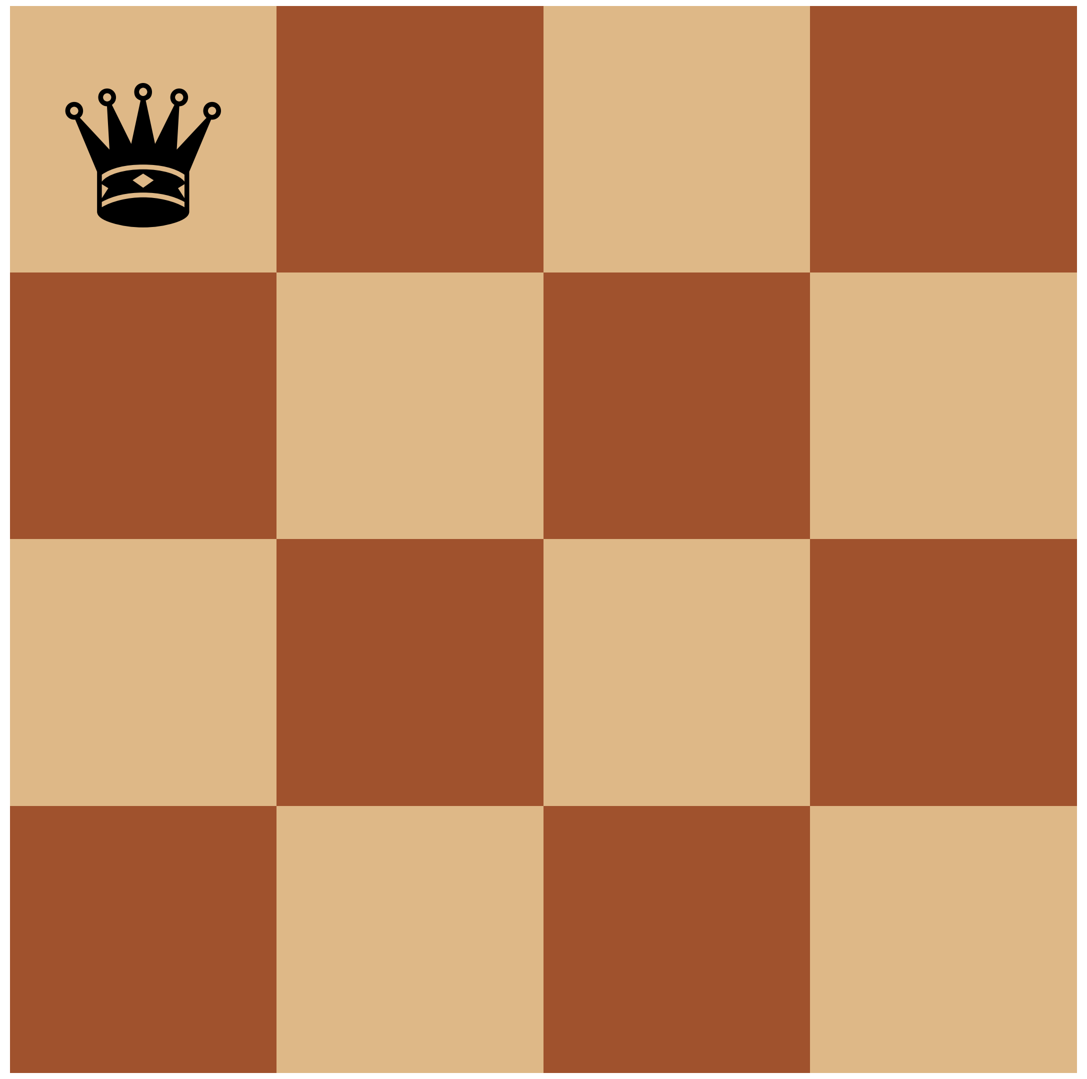
C. There's no need to pick the next row or backtrack now; it can do that after picking a row for column 3

Example: $n = 4$



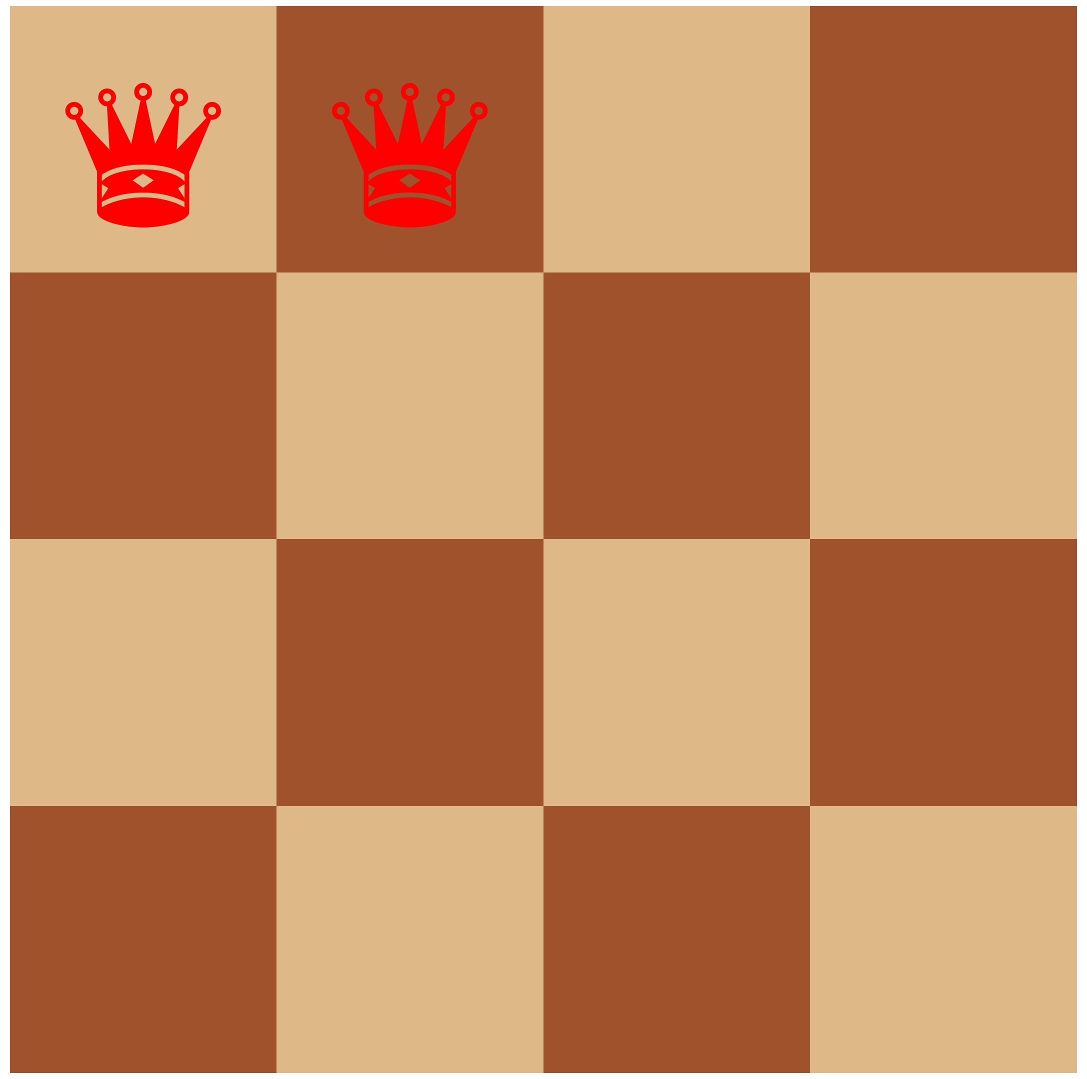
Initial state

Example: $n = 4$



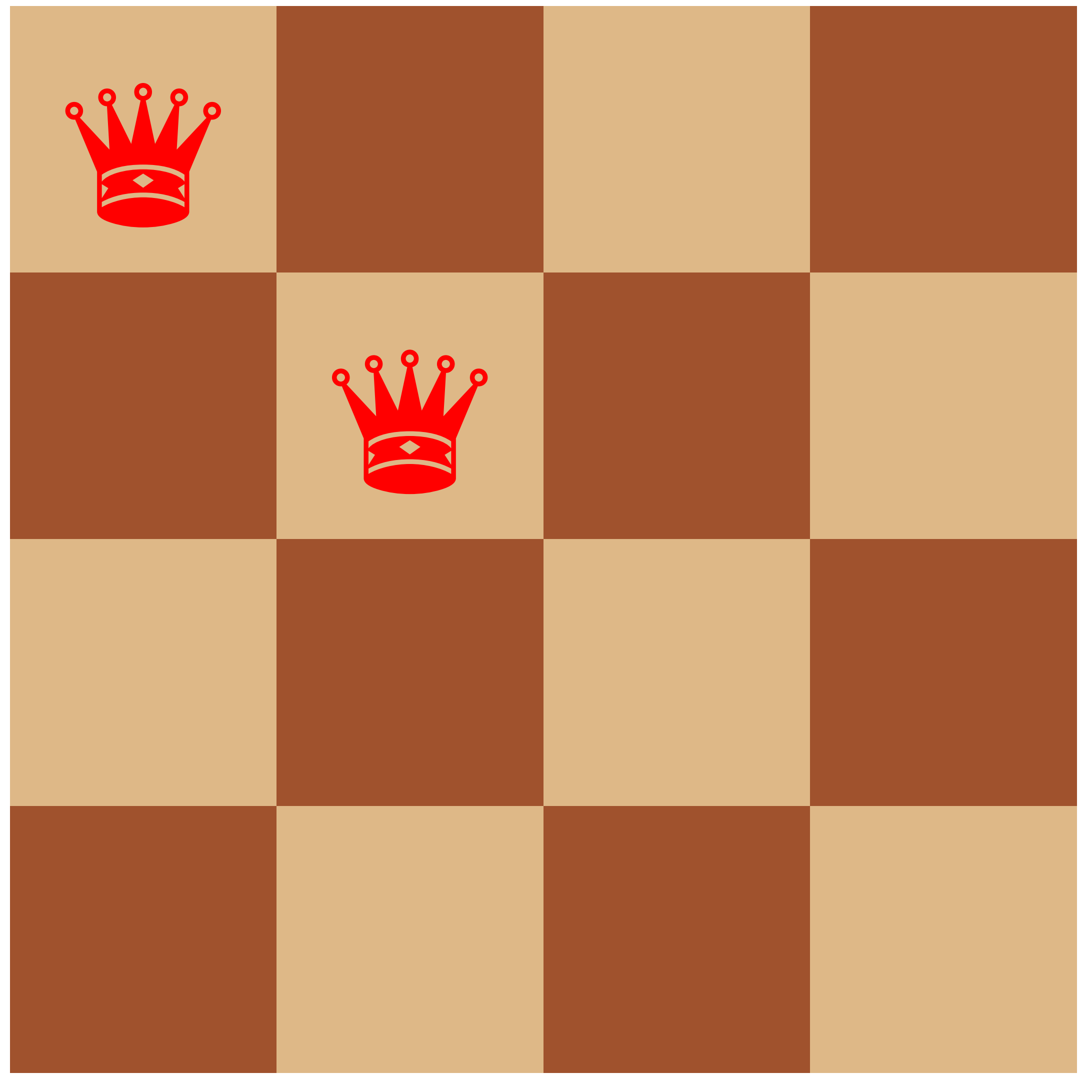
Step 1

Example: $n = 4$



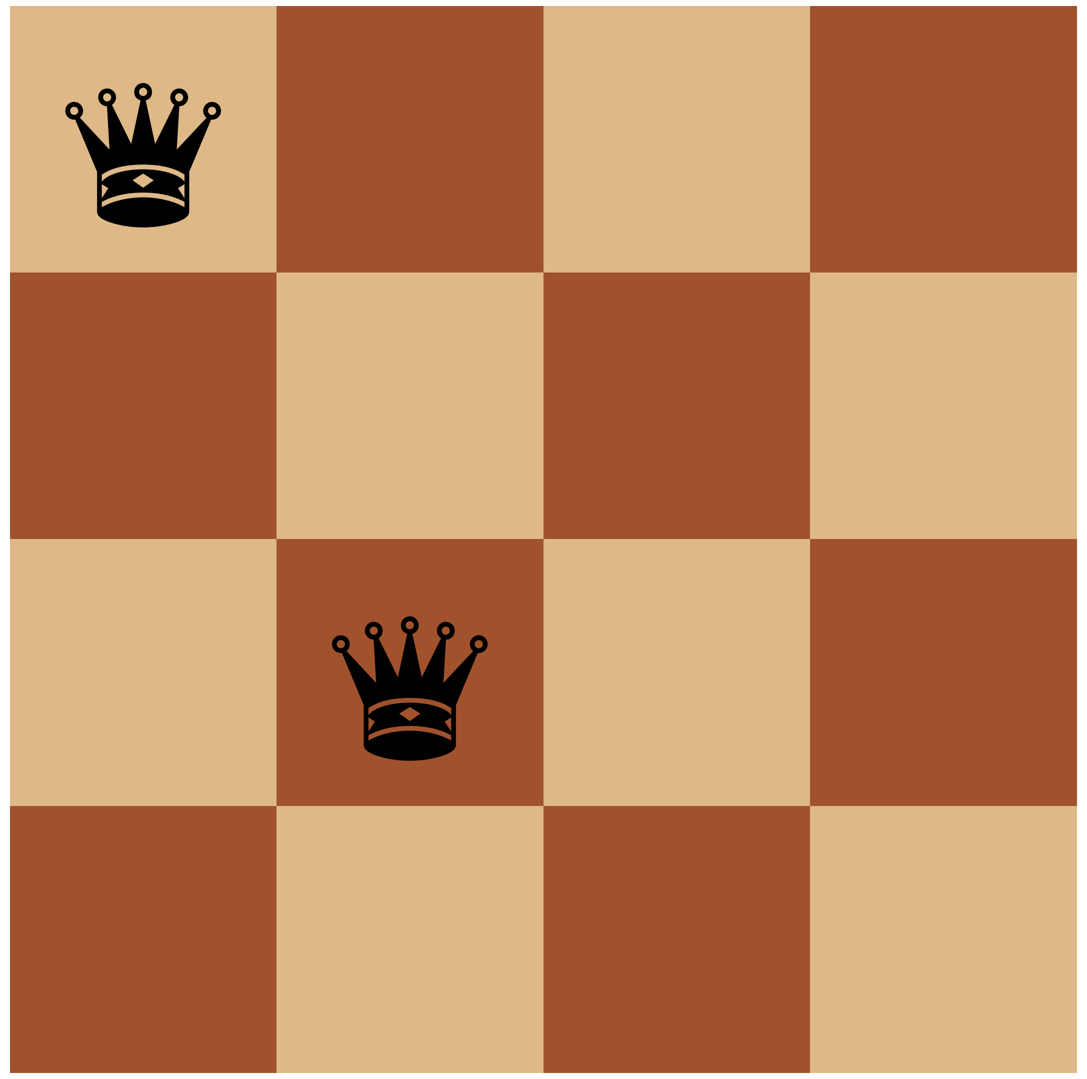
Step 2

Example: $n = 4$



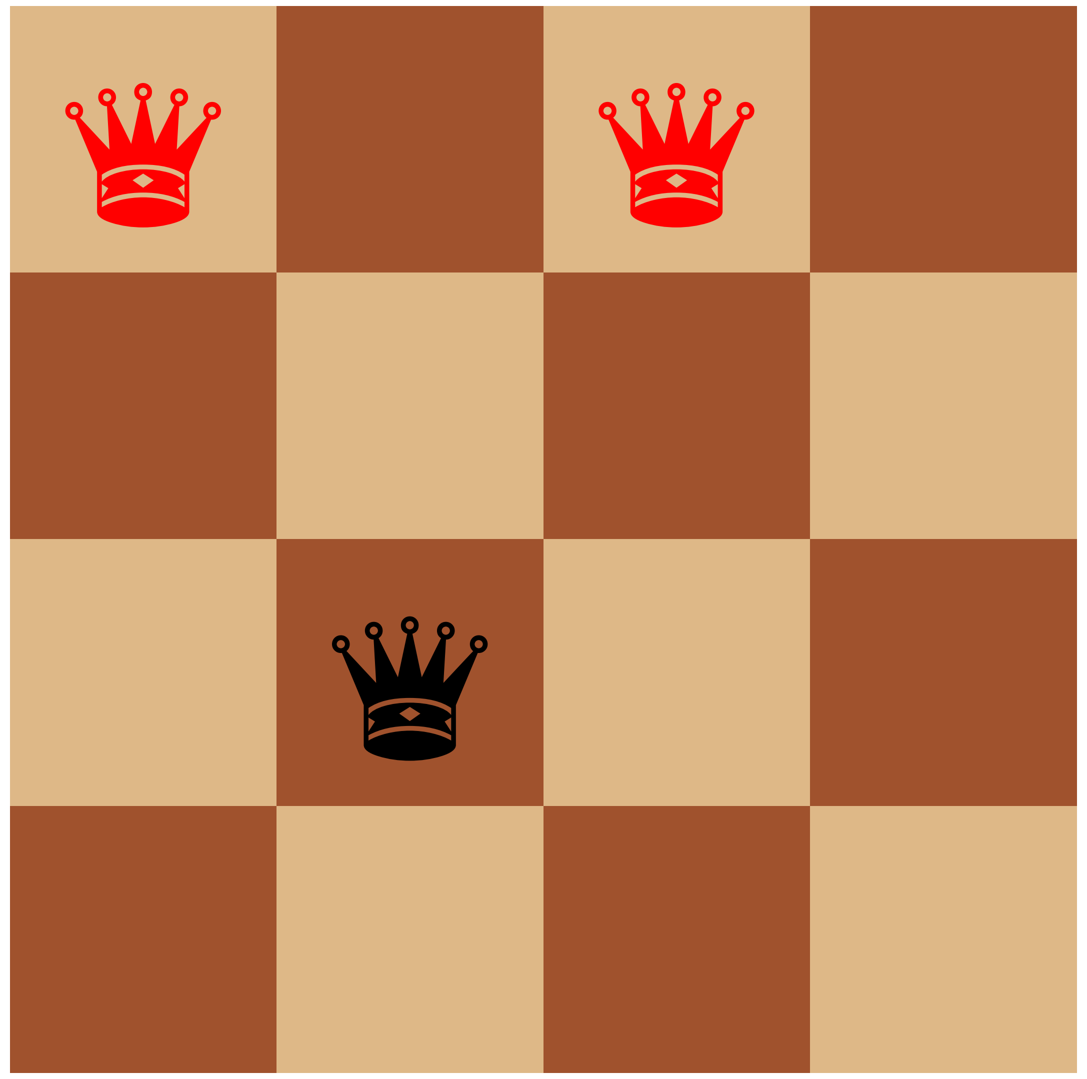
Step 3

Example: $n = 4$



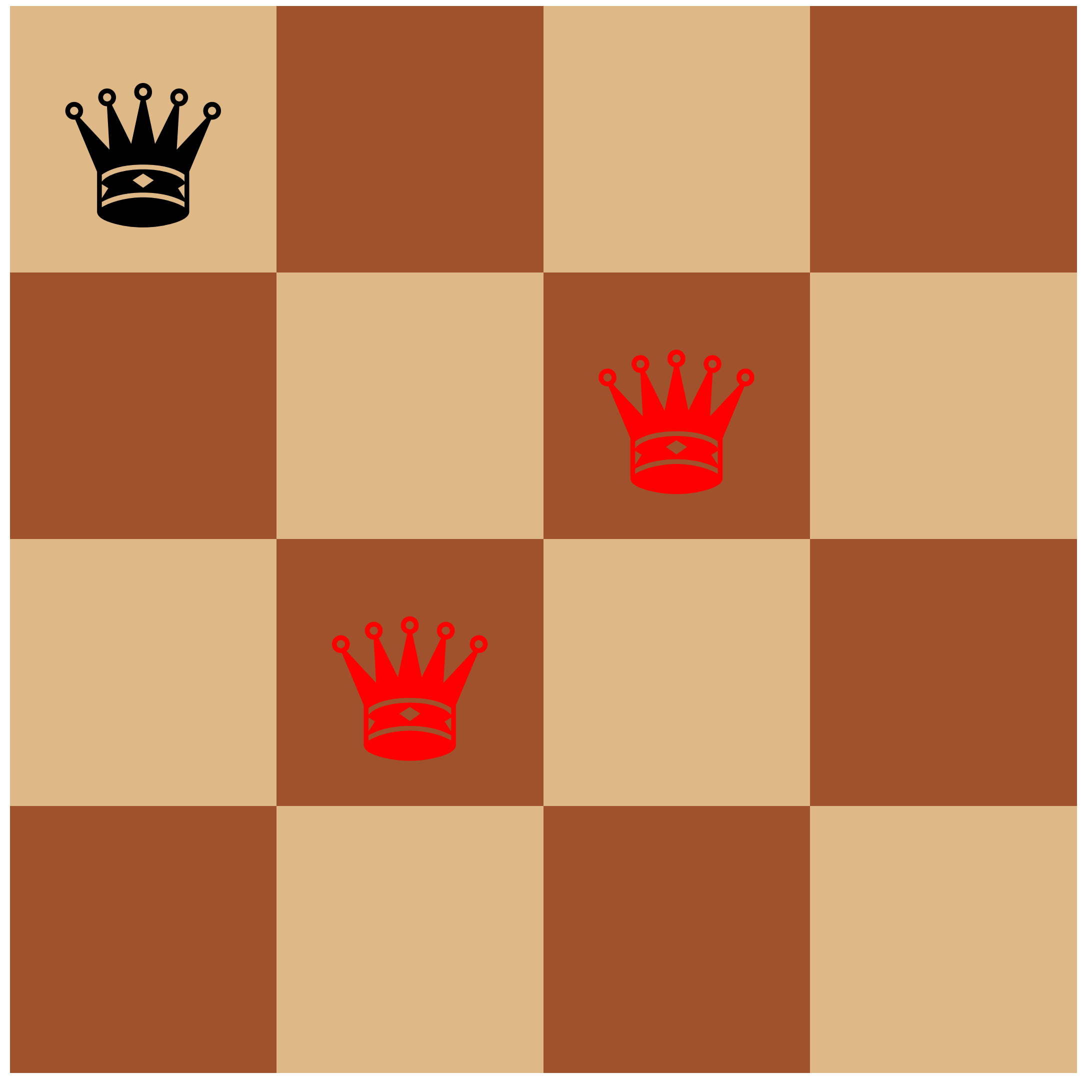
Step 4

Example: $n = 4$



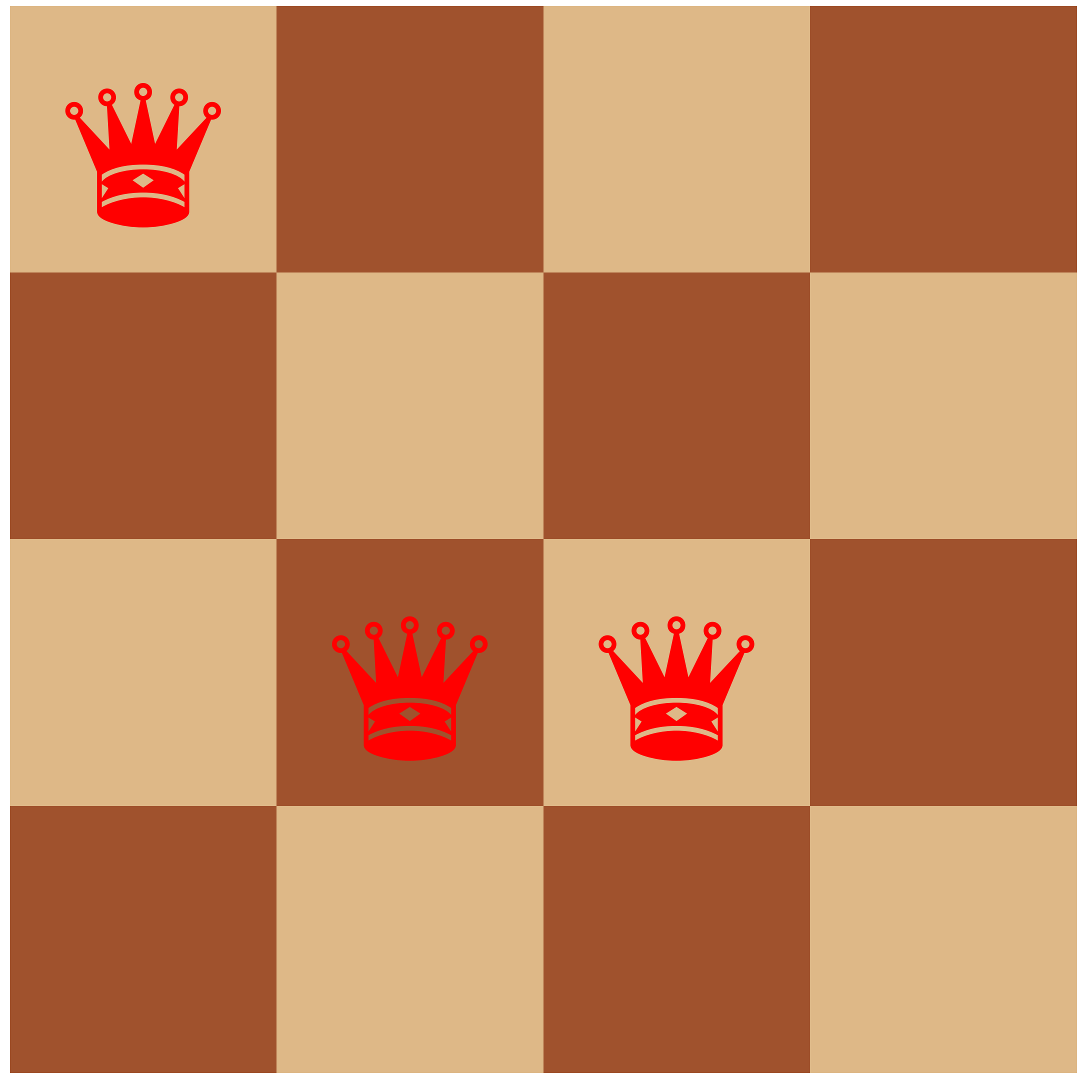
Step 5

Example: n = 4



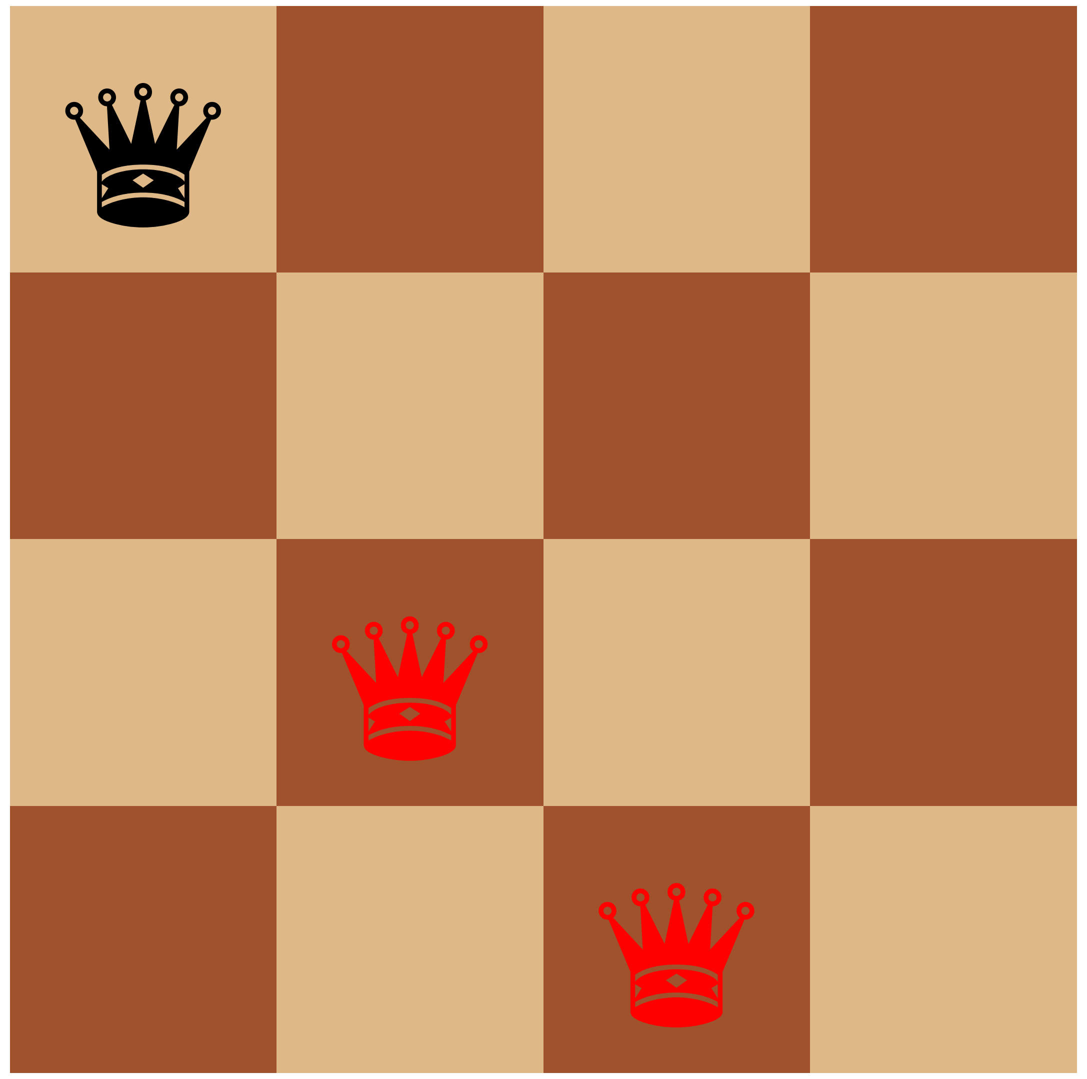
Step 6

Example: $n = 4$



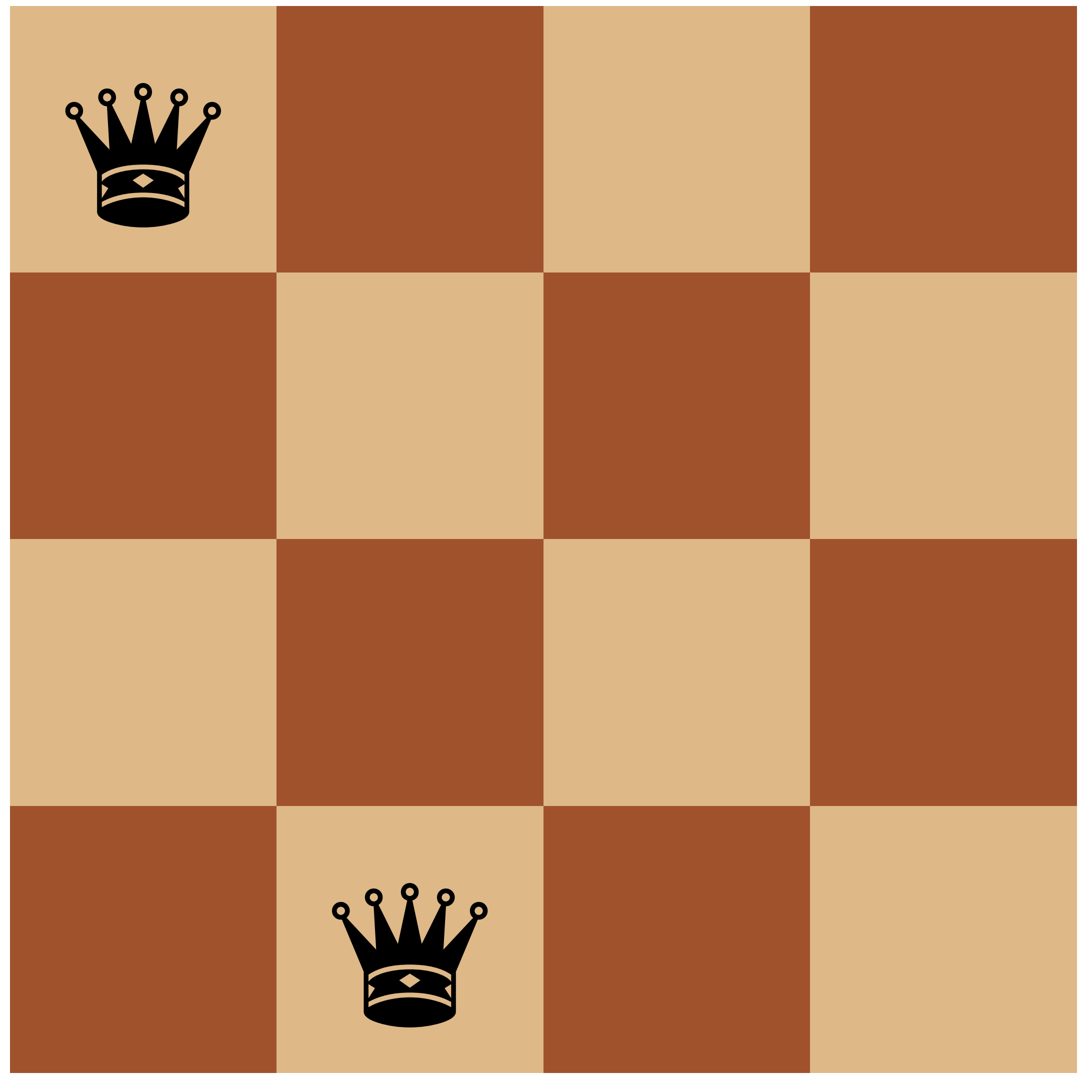
Step 7

Example: $n = 4$



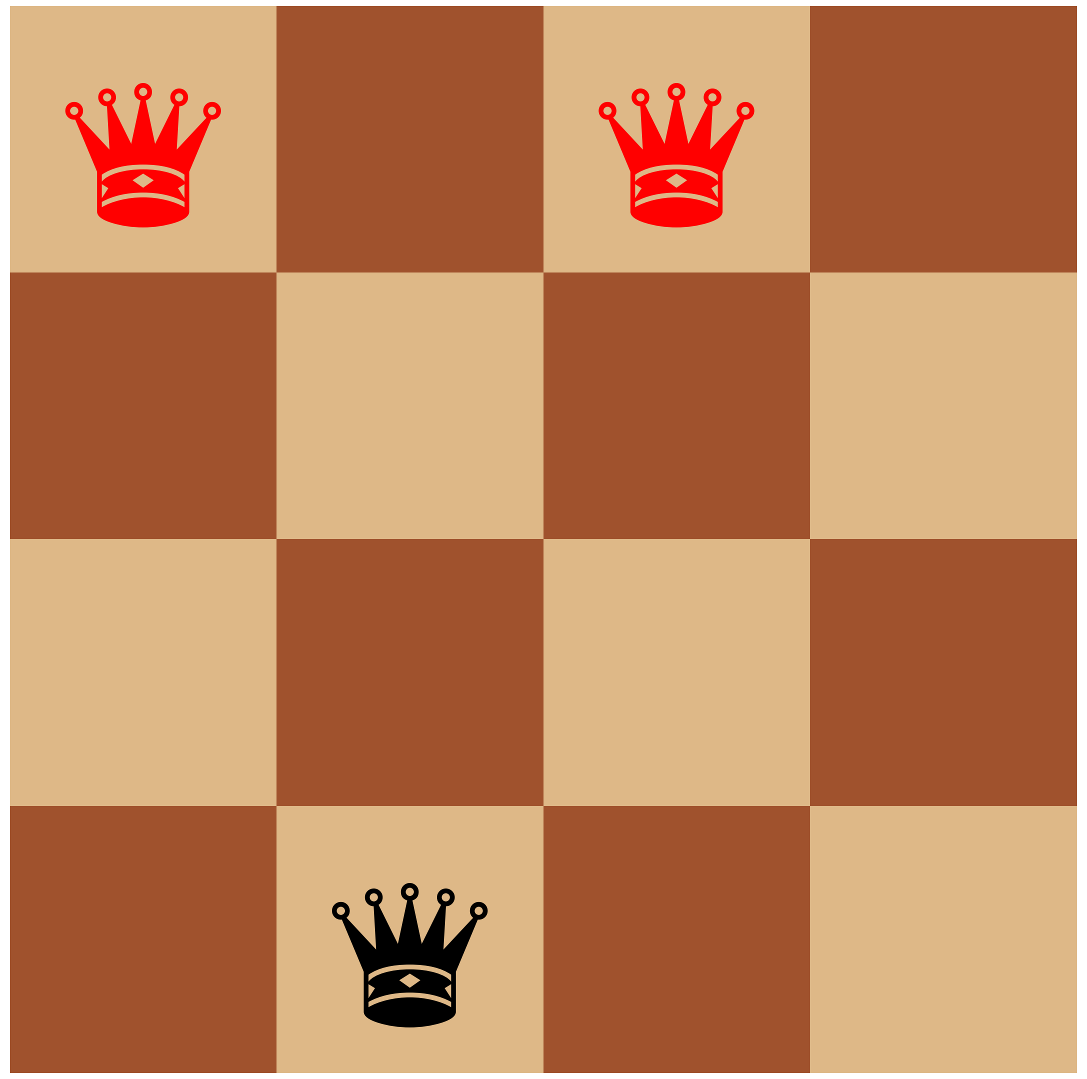
Step 8

Example: n = 4



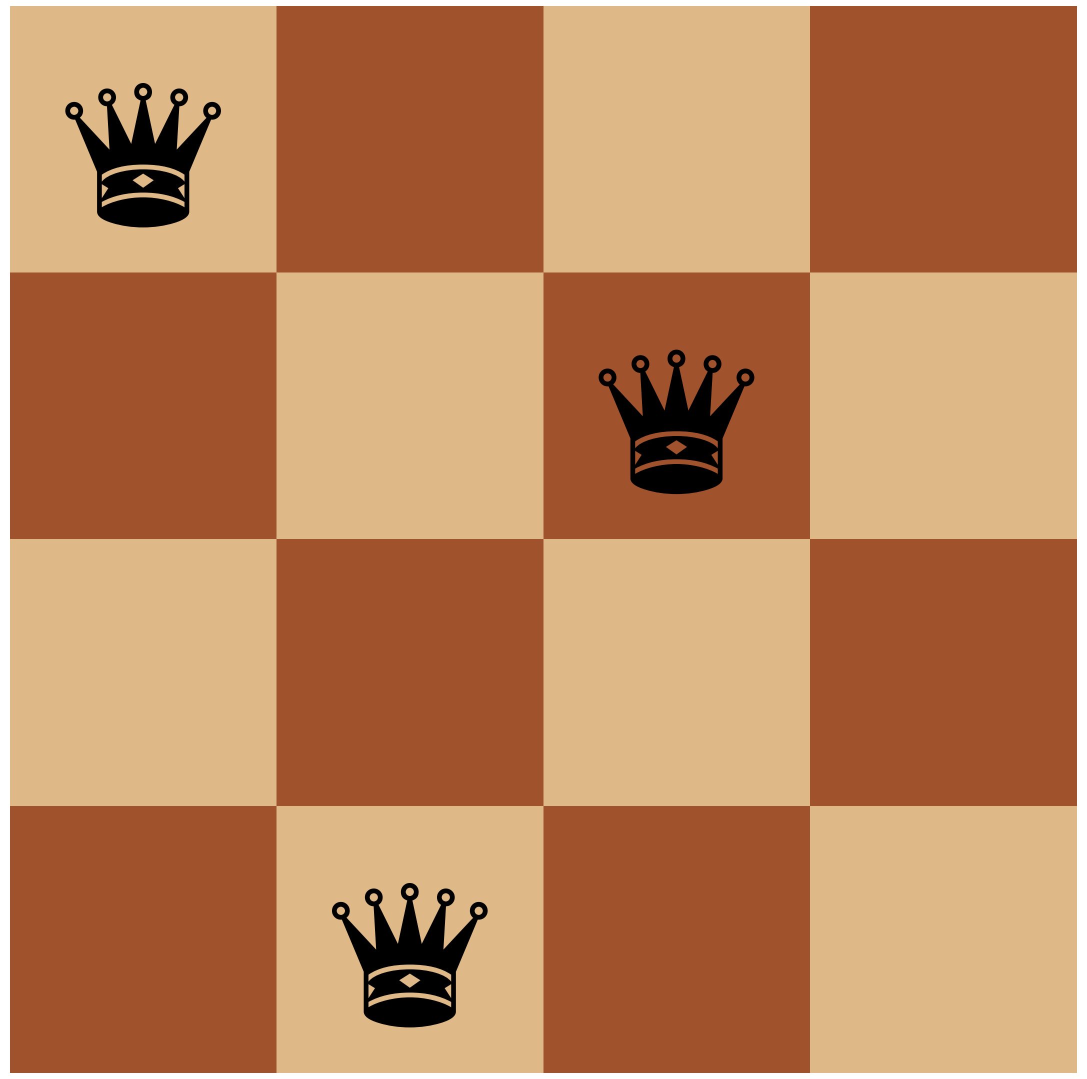
Step 9

Example: n = 4



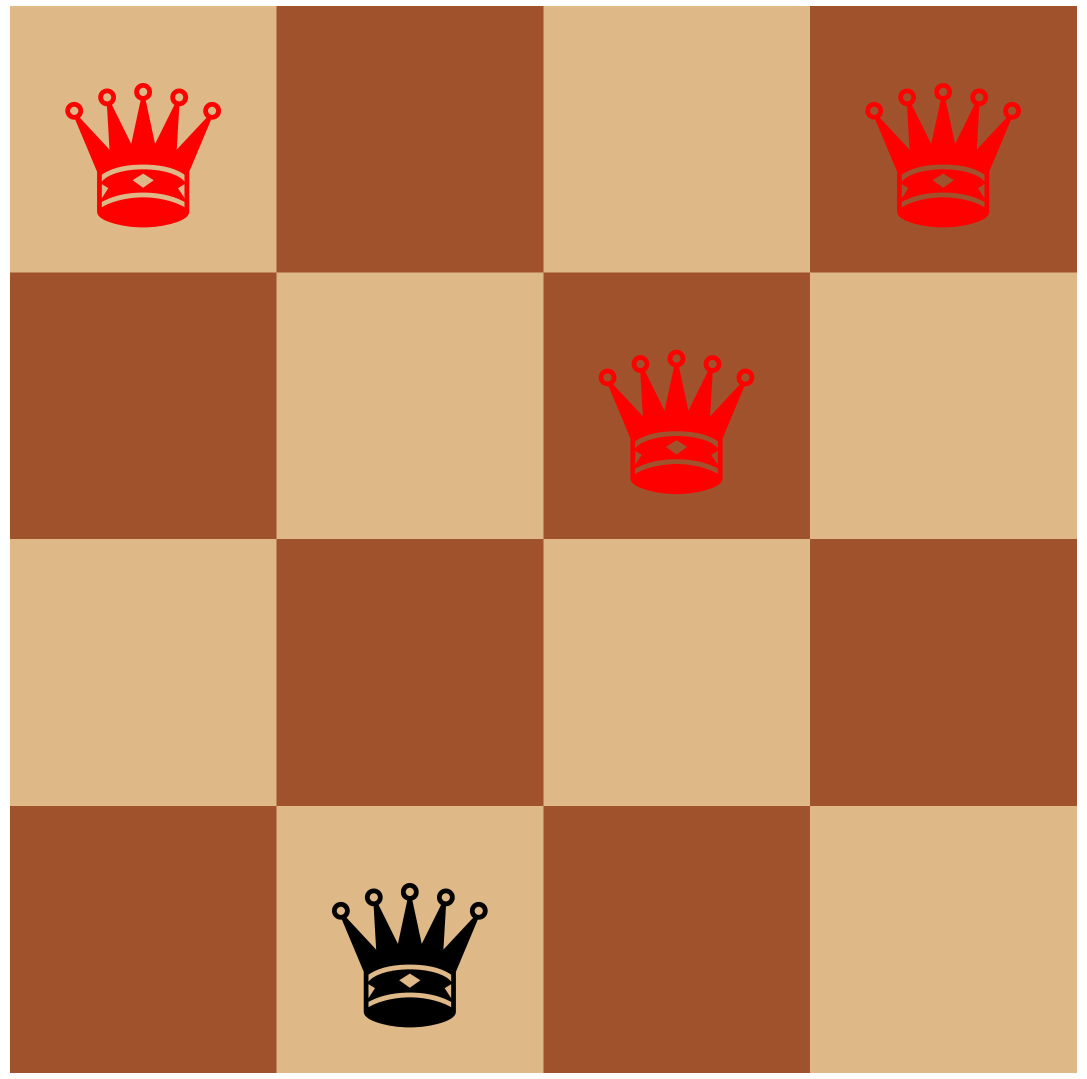
Step 10

Example: n = 4



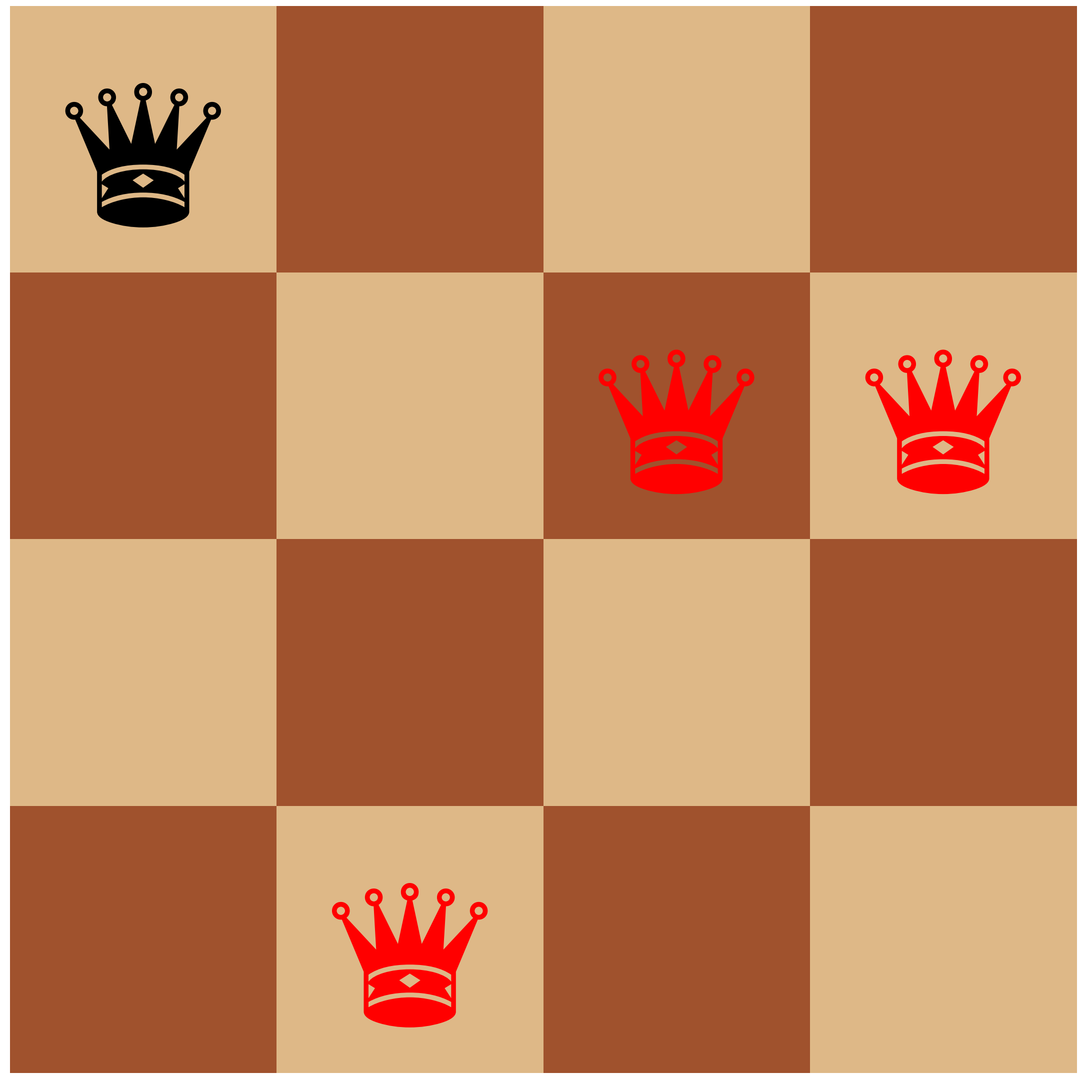
Step 11

Example: $n = 4$



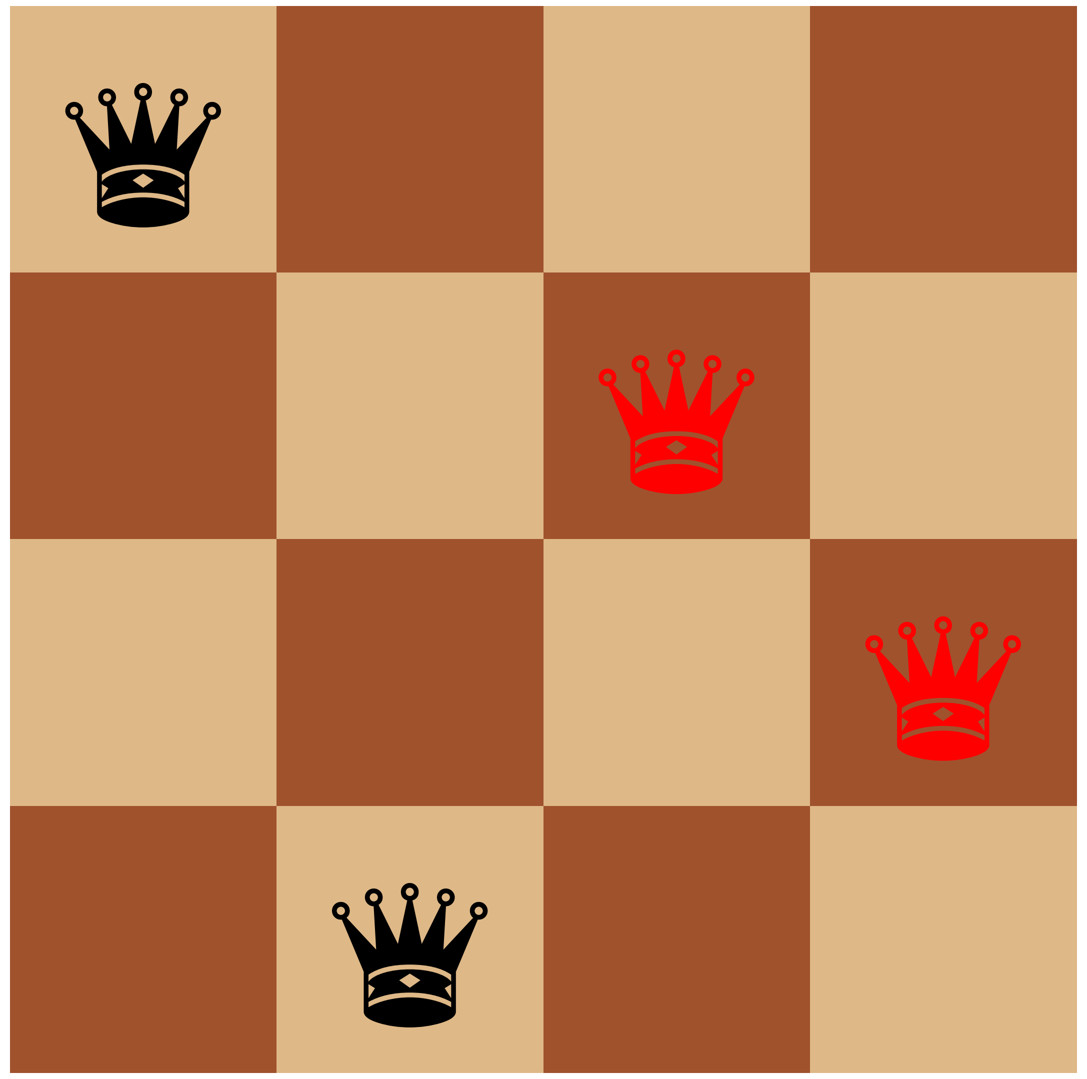
Step 12

Example: n = 4



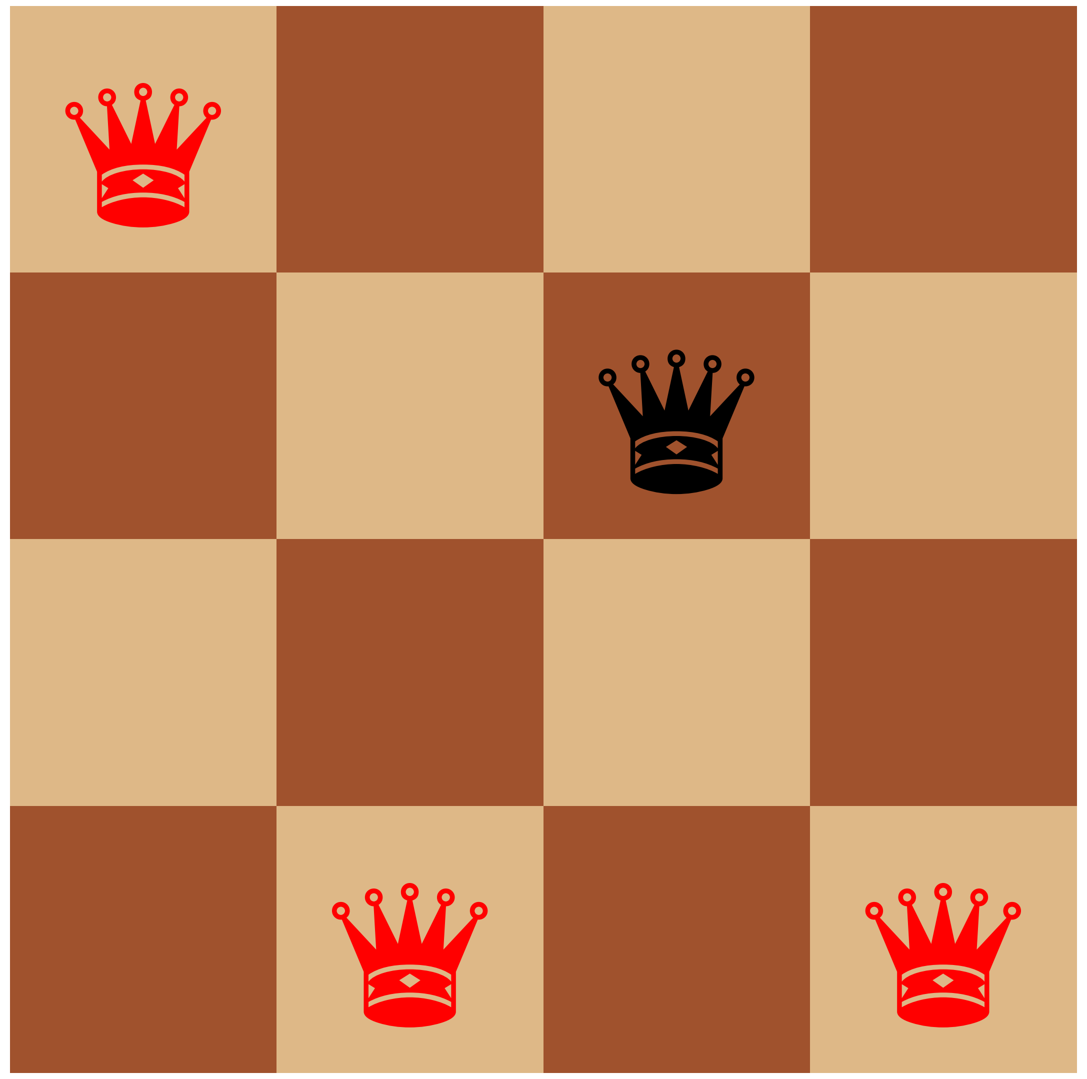
Step 13

Example: n = 4



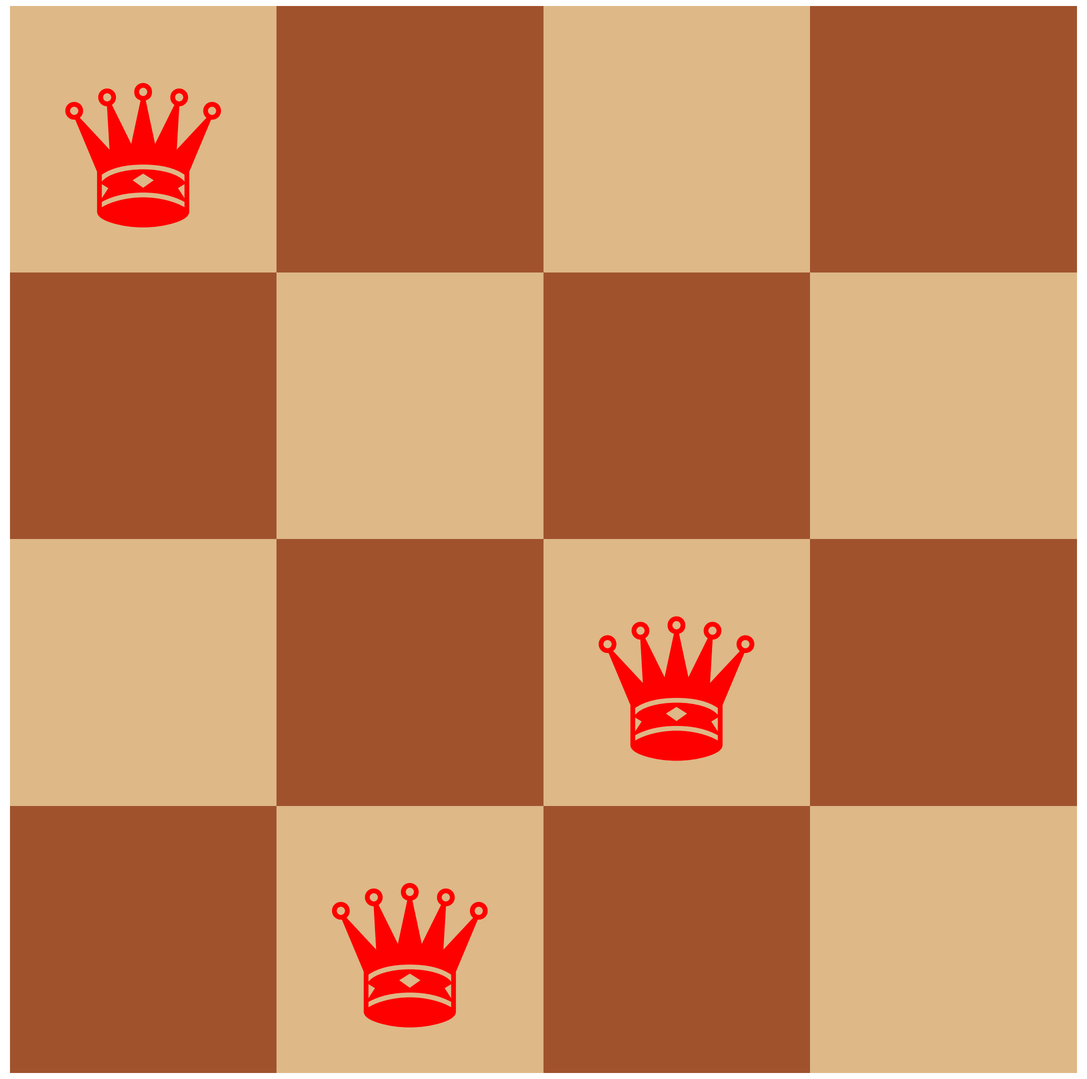
Step 14

Example: n = 4



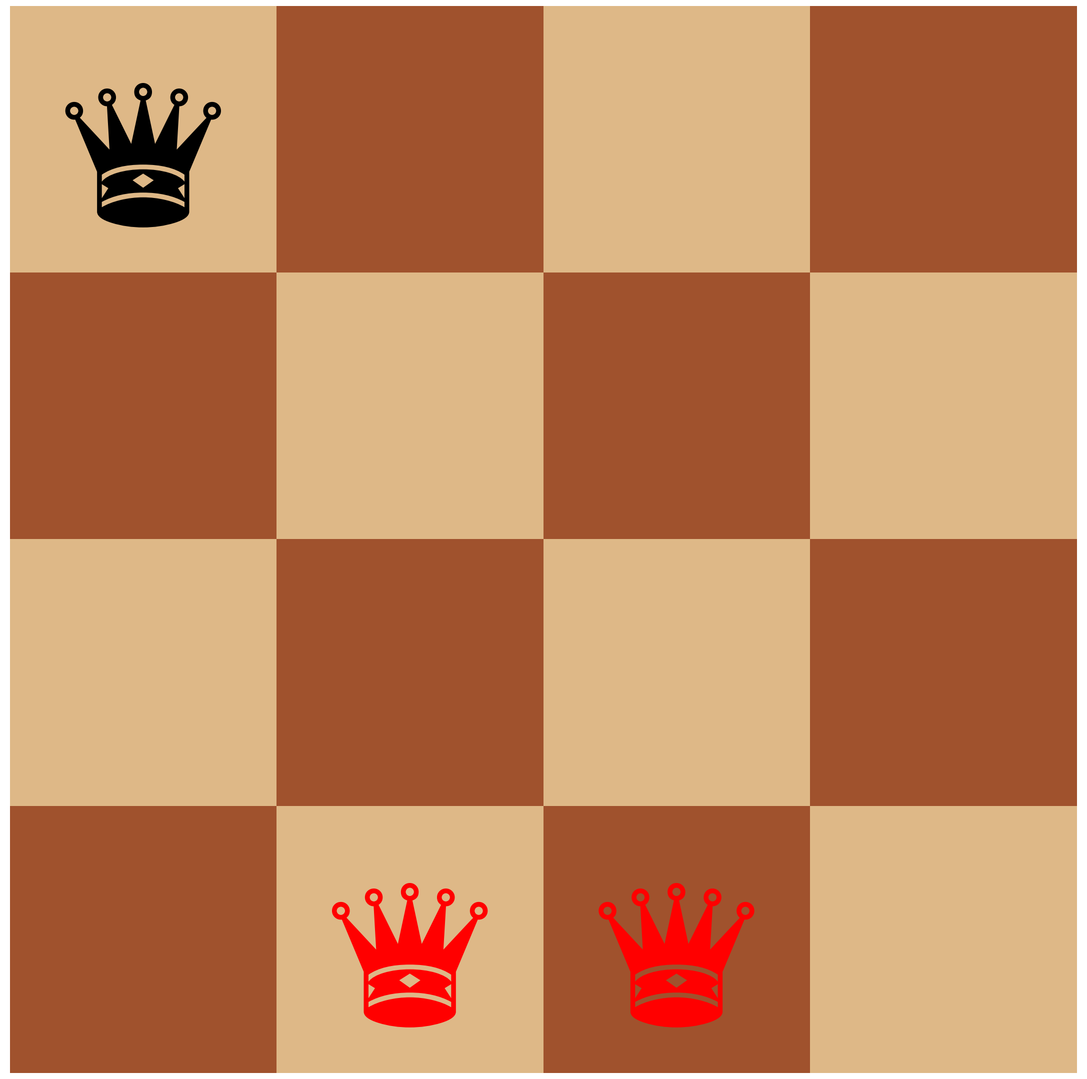
Step 15

Example: n = 4



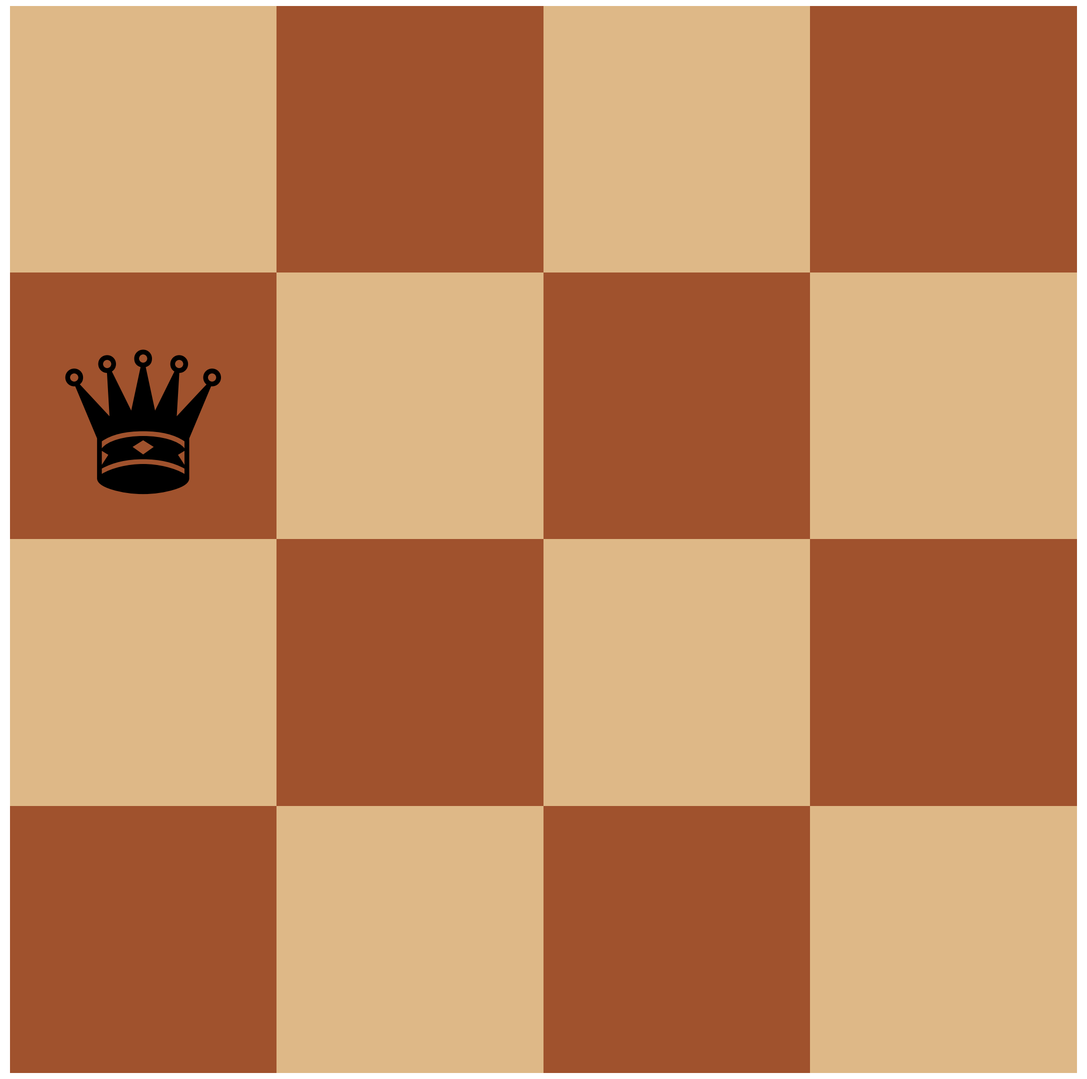
Step 16

Example: $n = 4$



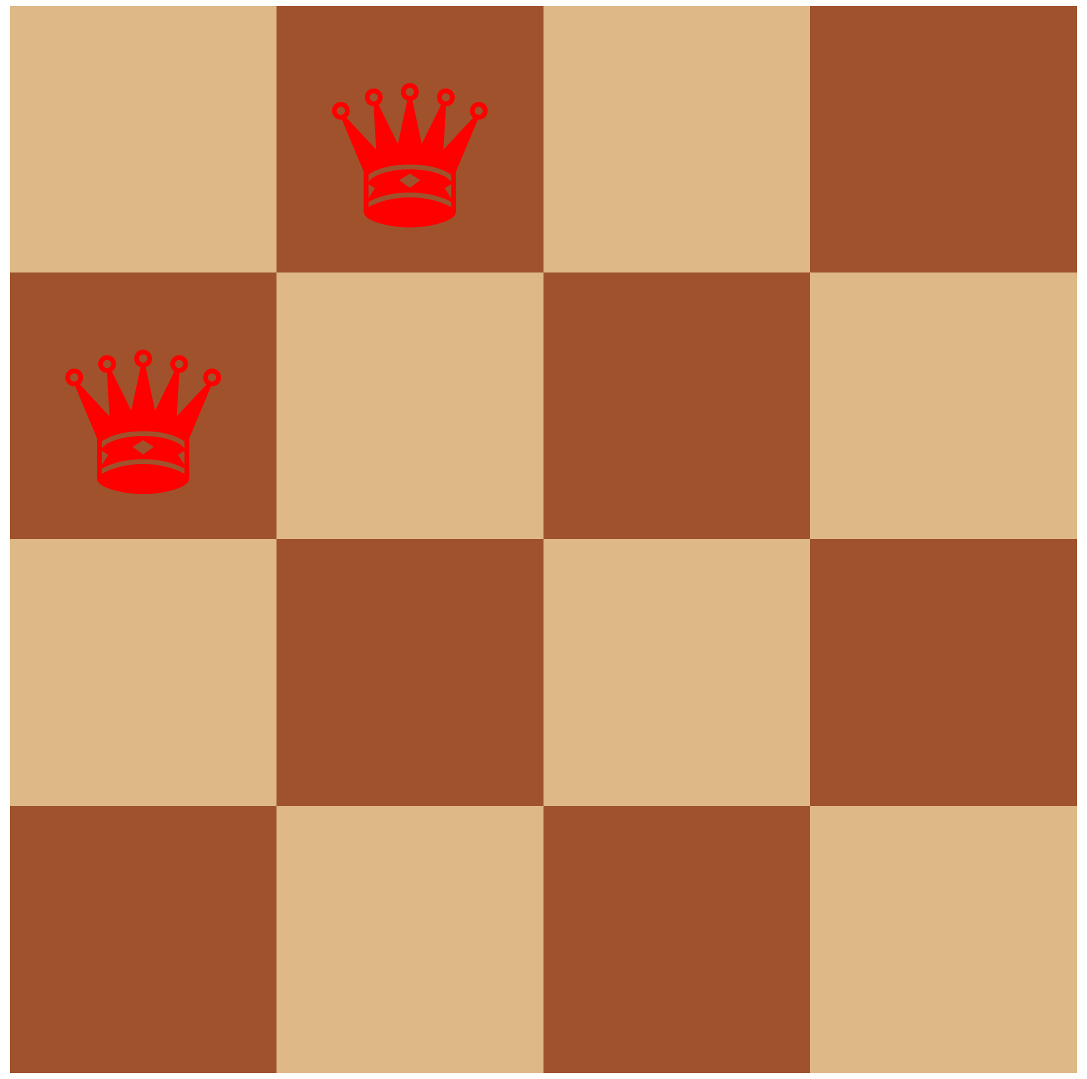
Step 17

Example: n = 4



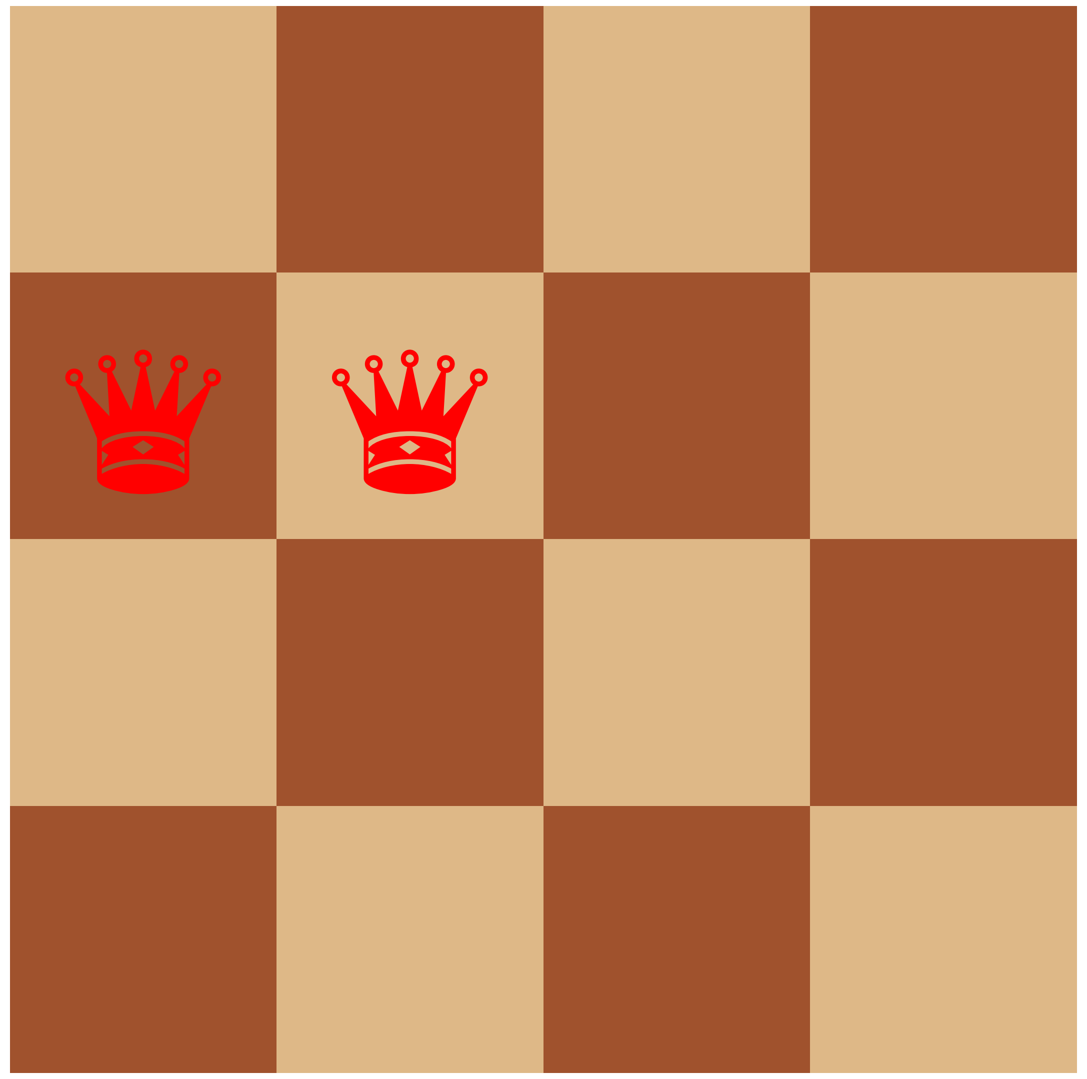
Step 18

Example: n = 4



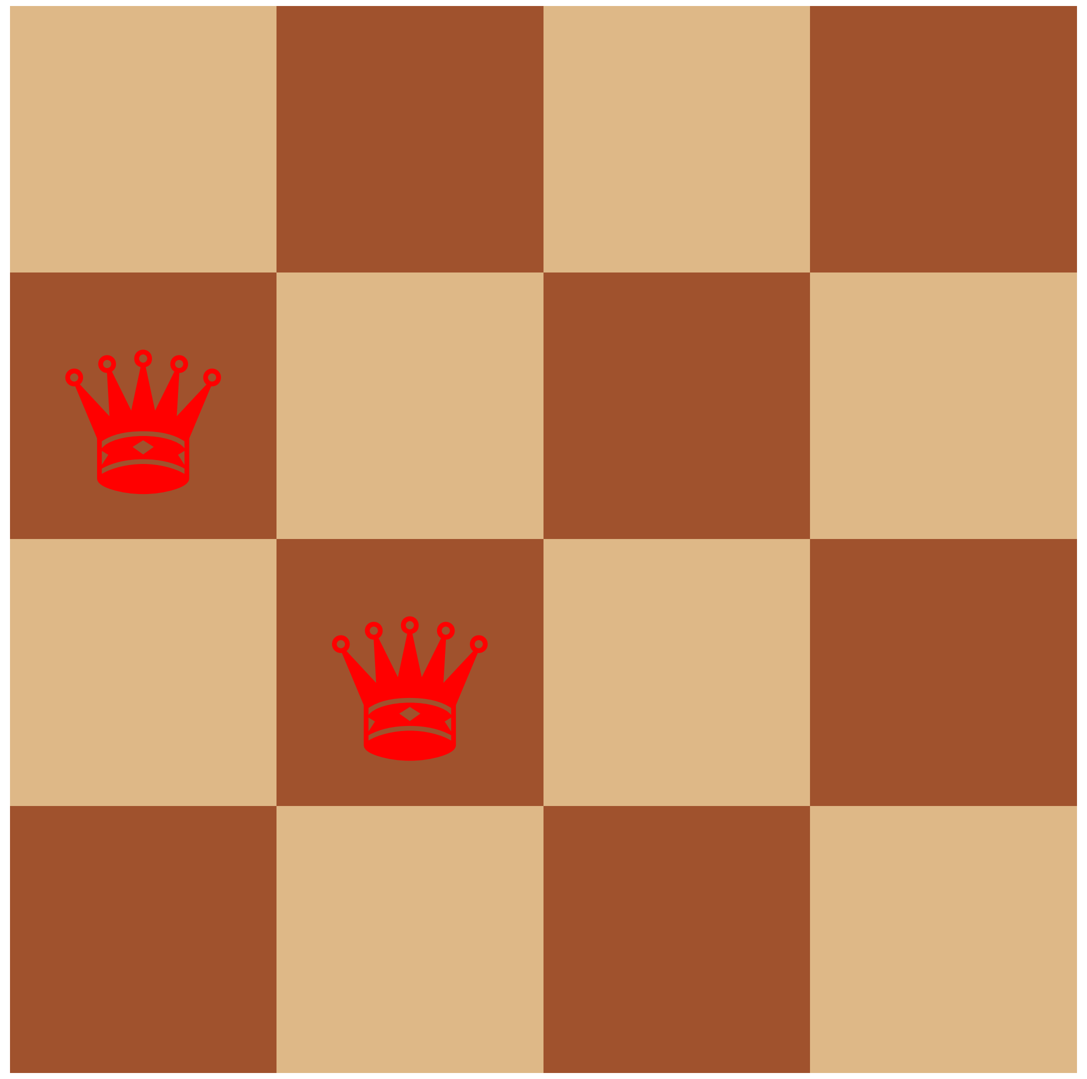
Step 19

Example: $n = 4$



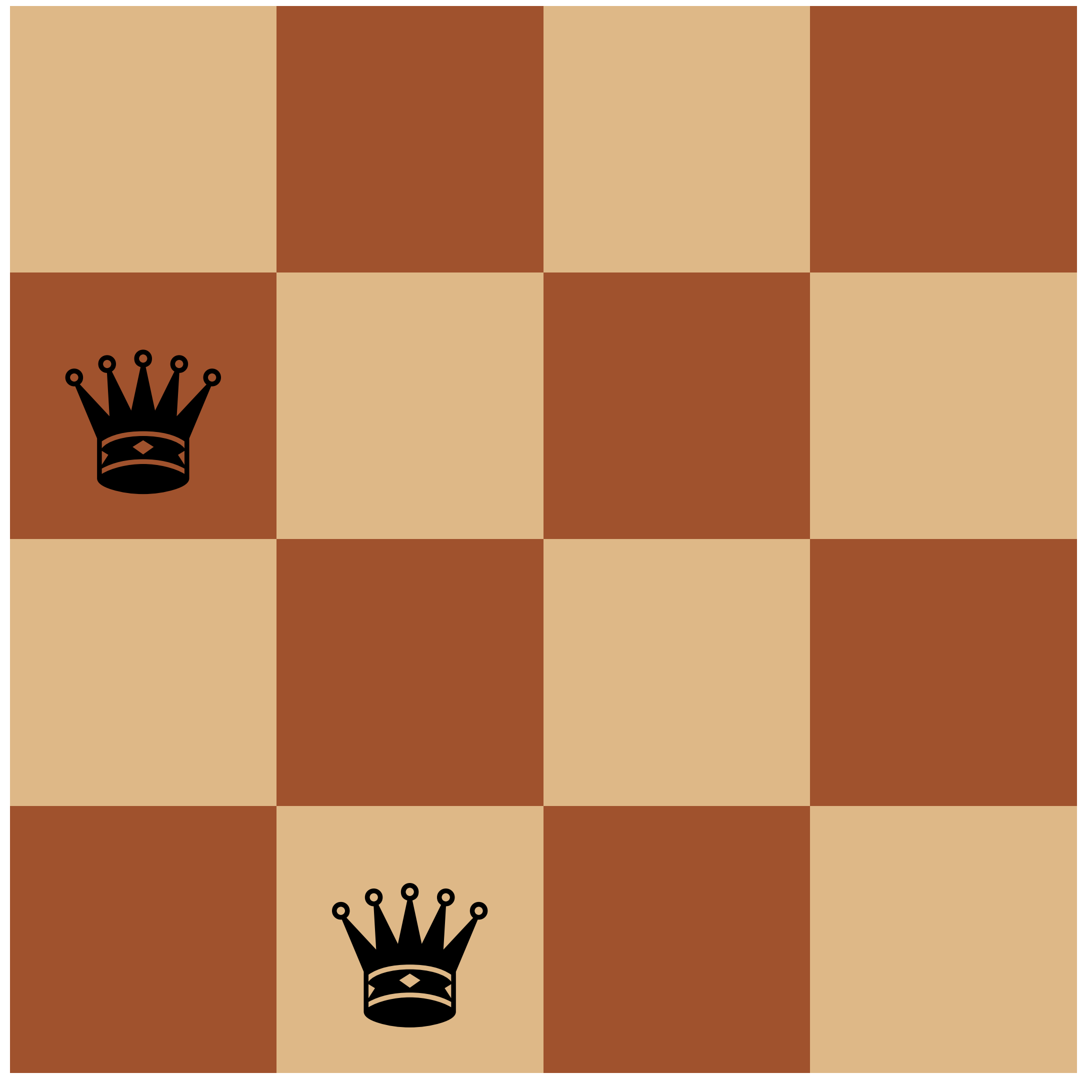
Step 20

Example: n = 4



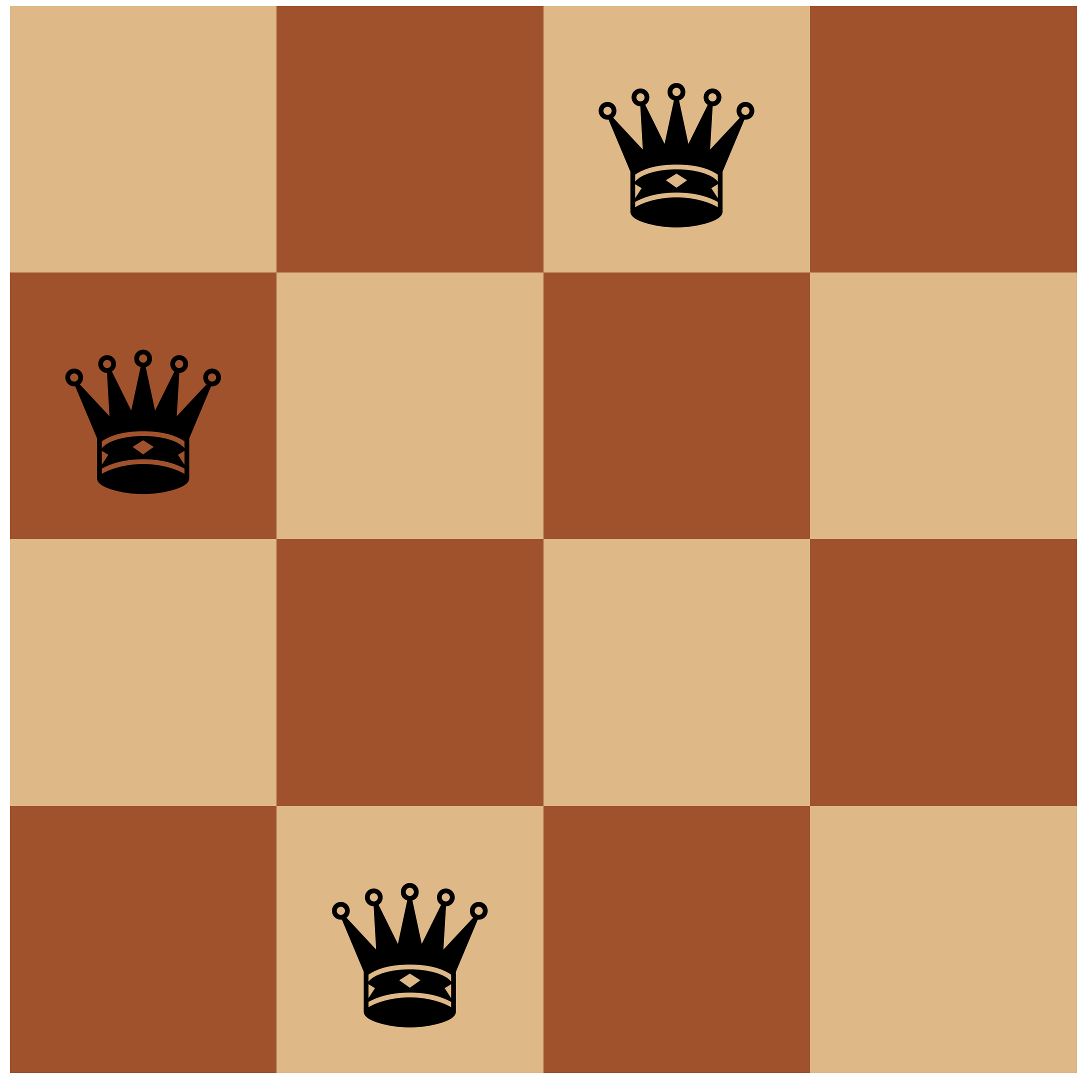
Step 21

Example: n = 4



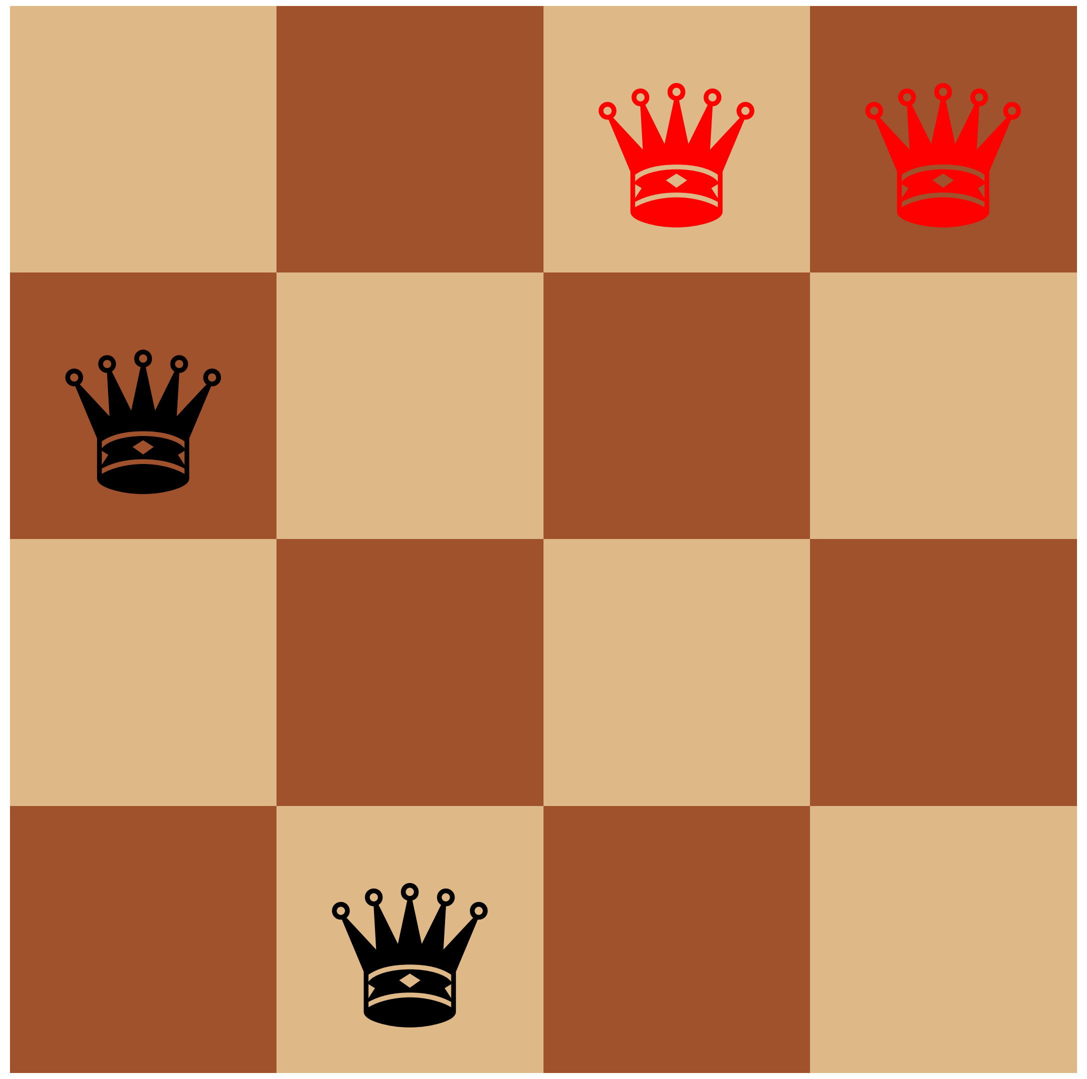
Step 22

Example: n = 4



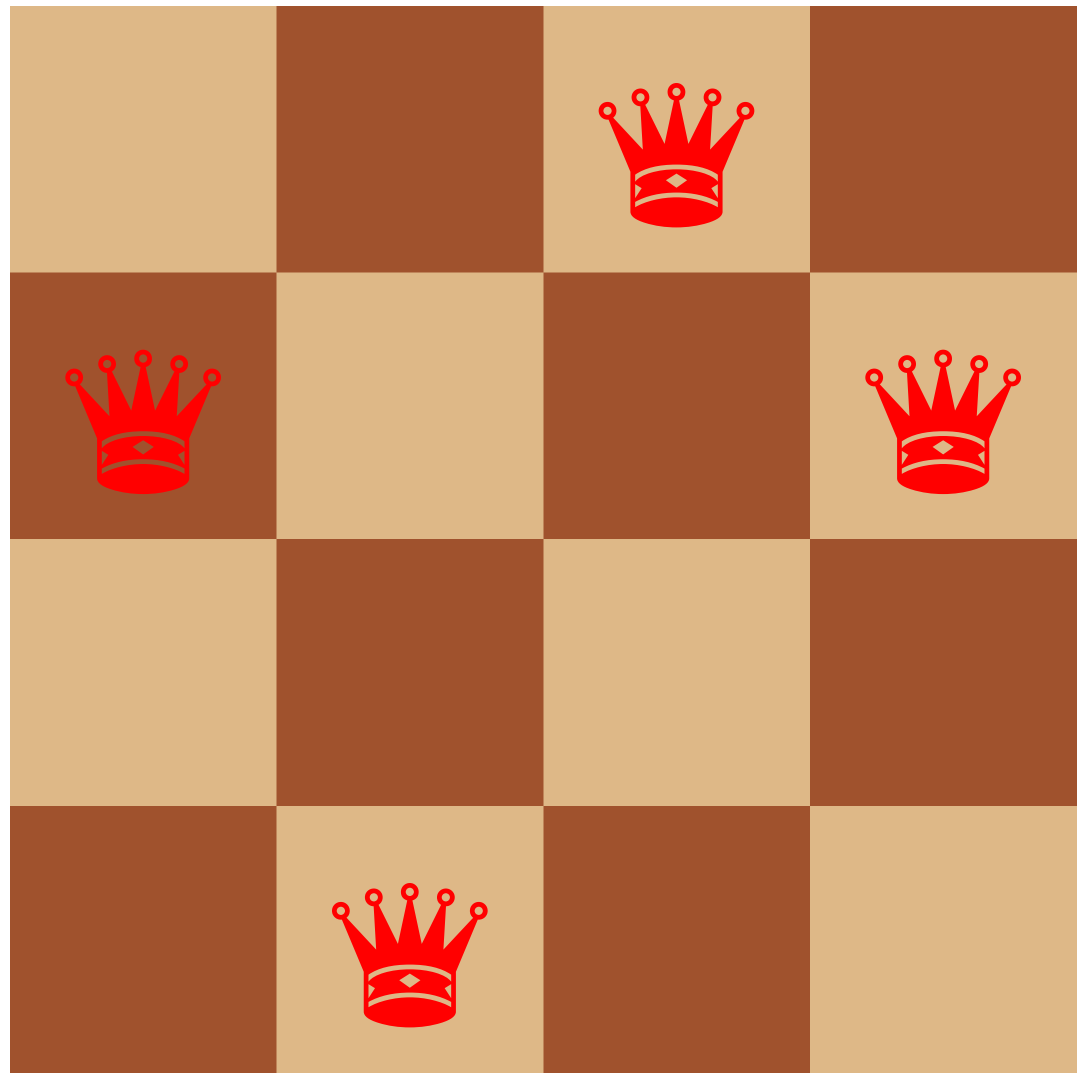
Step 23

Example: $n = 4$



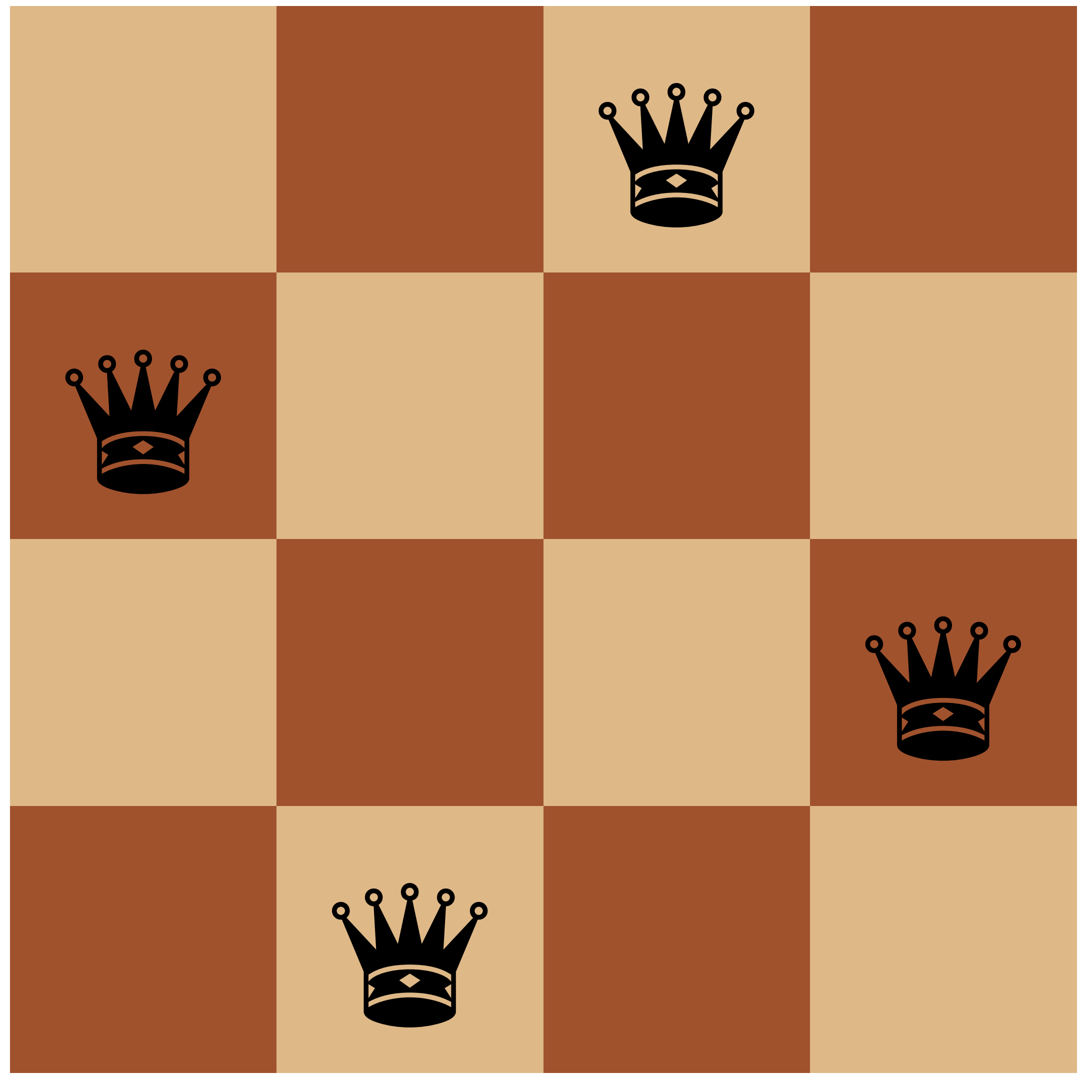
Step 24

Example: $n = 4$



Step 25

Example: $n = 4$



Step 26

Success!

Generic backtracking pseudocode

\# params are the parameters of the problem at hand
\# sofar is a list of steps that make up the current partial solution
\# this either returns a complete solution or returns a failure signal of some kind
backtrack(*params*, *sofar*)

- If *sofar* is a complete solution, return *sofar*
- For each possible value v for the next step
  - If adding *v* to *sofar* makes a feasible partial solution, then
    - *res* = backtrack(*params*, *sofar*.append(*v*))
    - If *res* is not the failure signal, then return *res*
- return failure # if we made it here, no possible value of *v* led to a solution

What should we use as a failure signal?

Some options

- `null`
- `#f`
- `'failure`

`null` actually isn't a great option because it's also the empty list `'()` and `'()` might be a valid solution

- E.g., imagine trying to find a subset of numbers in a list that sum to a given value, `(subset-sum lst n)`, if `n` is `0`, then returning `'()` is the only correct solution

The other two are reasonable choices

Backtracking in Racket

```
; sofar is the list of steps so far in reverse order
; curr is the current value to try
(define (backtrack params sofar curr)
  (cond [⟨sofar is a complete solution⟩ (reverse sofar)]
        [⟨curr is out of the range of possible values⟩ #f]
        [(feasible sofar curr)
         (let ([res (backtrack params
                               (cons curr sofar)
                               ⟨first value for next step⟩))])
           (if res
               res
               (backtrack params sofar ⟨value after curr⟩)))]
        [else (backtrack params sofar ⟨value after curr⟩)]))
```

Using backtrack

(Of course, you'll write specific backtrack and feasible functions for each problem)

```
(backtrack params empty ⟨first value for first step⟩)
```

n-queens
(single solution)

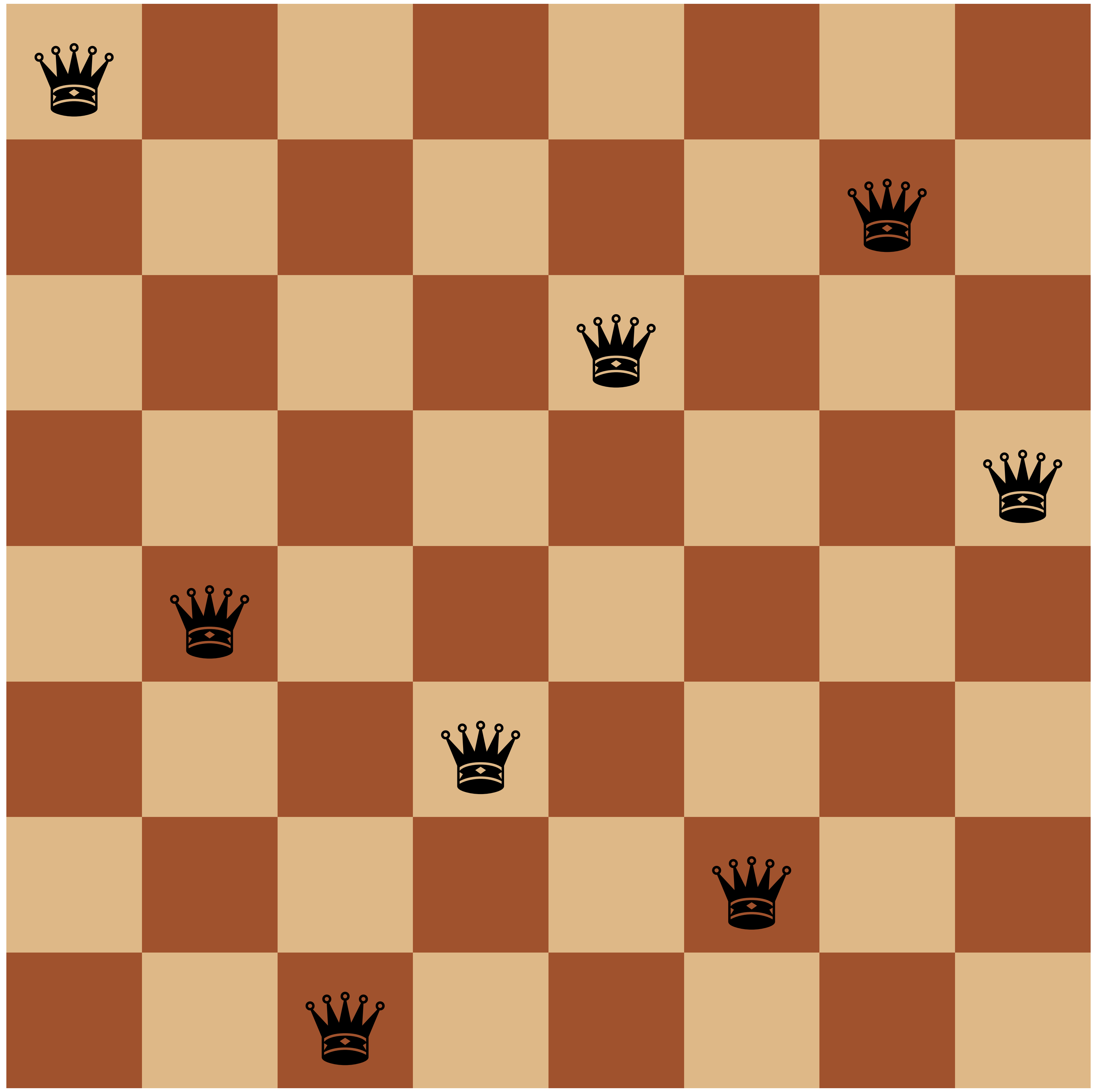
First, how should we represent a solution?

- A list of row–column pairs like
  `'((0 0) (4 1) (7 2) (5 3)`
  `  (2 4) (6 5) (1 6) (3 7))`
- A list of rows like `'(0 4 7 5 2 6 1 3)`

Either works and we can easily convert from one to the other

- `(map list list-of-rows (range n))`
- `(map first list-of-pairs)`
  The list must be sorted by column first

Let's use a list of rows



Careful!

Our normal procedure for constructing the list of steps prepends the current step to our partial solution

- `(bt (cons curr sofar) initial)`

This means our partial solution will be in reverse order which means we need to

- reverse our final result so it's in the correct order; and
- write our `(feasible? sofar curr)` procedure keeping this in mind

n-queens

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

What's our `initial` value?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. 0

B. 1

C. n

D. n-1

E. n+1

What's our `(next curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(add1 curr)`

B. `(add1 (modulo curr n))`

C. `(modulo (add1 curr) n)`

D. `(modulo (add1 curr) (add1 n))`

E. More than one of the above

What's our `(is-complete? sofar)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(feasible? sofar null)`

B. `(= (length sofar) n)`

C. `(= (length sofar) (add1 n))`

D. `(= (length sofar) (sub1 n))`

E. More than one of the above

What's our `(out-of-range? curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(< curr n)`

B. `(= curr n)`

C. `(> curr n)`

D. `(< n 0)`

E. `(not (integer? curr))`

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

Move left through reversed columns

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

Move left through reversed columns

Move up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

At various points, the backtracking algorithm needs to choose the next value to try for the current step or it needs to backtrack to a previous step.

When does it need to backtrack to a previous step?

A. It backtracks each time it encounters a partial solution that isn't feasible

B. It backtracks whenever there are no more choices for the current step

C. It backtracks when the choice it makes for the final step leads to an invalid solution

D. It backtracks after each invalid choice

E. All of the above

One common variant: all solutions

Rather than using `#f` to signal failure, we'll use `empty` to indicate the set of solutions is empty

Key differences

- Rather than stopping after a single solution is found, keep going
- Each call will return a list of solutions
- When we have a feasible solution, we need to get all the solutions both using the feasible one and not

All solutions in Racket

```
(define (all-sol params sofar curr)
  (cond [⟨sofar is a complete solution⟩ (list (reverse sofar))]
        [⟨curr is out of the range of possible values⟩ '()]
        [(feasible sofar curr)
         (let ([res1 (all-sol params
                                (cons curr sofar)
                                ⟨first value for next step⟩))]
               [res2 (all-sol params sofar ⟨value after curr⟩)])
           (append res1 res2))]
        [else (all-sol params sofar ⟨value after curr⟩)]))

(all-sol params empty ⟨first value for first step⟩)
```

Permutations of {0, 1, ..., n-1}

(Not the most efficient way)

Let's compute all permutations of {0, 1, ..., n-1} using backtracking

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list sofar)]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-perms n)
  (bt n empty initial))
```

We just need to deal with the problem-specific parts

n-queens all solutions

No harder than getting one solution, we just need to plug in the usual parts

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list (reverse sofar))]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-queens n)
  (bt n empty initial))
```