Programming Abstractions

Week 4: Modules, backtracking, and data types

Stephen Checkoway

Modules in Racket

(Lack of) modules in Scheme

Traditional

Scheme has a (load file-name) that's like C's #include

- (load "foo.scm") simply reads in the content of the file foo.scm as scheme code

Upsides

- It's simple: It simply makes all of the definitions in the loaded file available in the current file

Downsides

- Only a single namespace so different files can't have procedures with the same name
- Allows no separation between an interface and an implementation
  - E.g., "private" helper functions aren't actually private

Modules in Racket

Modern

Each file that starts with `#lang` creates a module named after the file

`#lang` also specifies the language of the file

- Racket was designed to implement programming languages
- We're only going to use the Racket language itself
- All of our files start with `#lang racket`

Exposing definitions

`(provide …)`

By default, each definition you make in a Racket file is private to the file

To expose the definition, you use `(provide …)`

To expose all definitions, you use
(provide (all-defined-out))

E.g.,

```
#lang racket
(provide (all-defined-out))

(define (mul2 x)
  (* x 2))
```

Exposing only some definitions

`(provide sym1 sym2…)`

You can specify exactly which definitions are exposed by specifying them via one or more `provides`

```
#lang racket
(provide foo-a foo-b)
(provide bar-a bar-b)

(define helper …)    ; Not exposed

(define foo-a …)
(define foo-b …)

(define bar-a …)
(define bar-b …)
```

Importing definitions from modules

`(require …)`

To get access to a module's definitions we need to `require` the module

E.g., the `test.rkt` files in the assignments require the homework file

- `(require "hw1.rkt")` imports the definitions from the file `hw1.rkt`

Plus a whole lot more

Racket supports a dizzying array of module options

`#lang racket` is a shorthand for `(module module-id racket …)`

Submodules can be created using a variety of module forms: `module`, `module*`, `module+`

Requiring modules can

- Import specific symbols
- Exclude specific symbols
- Rename symbols (e.g., two modules can be imported with conflicting names)

We won't need any of this extra functionality in the course, because our programs are so short

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

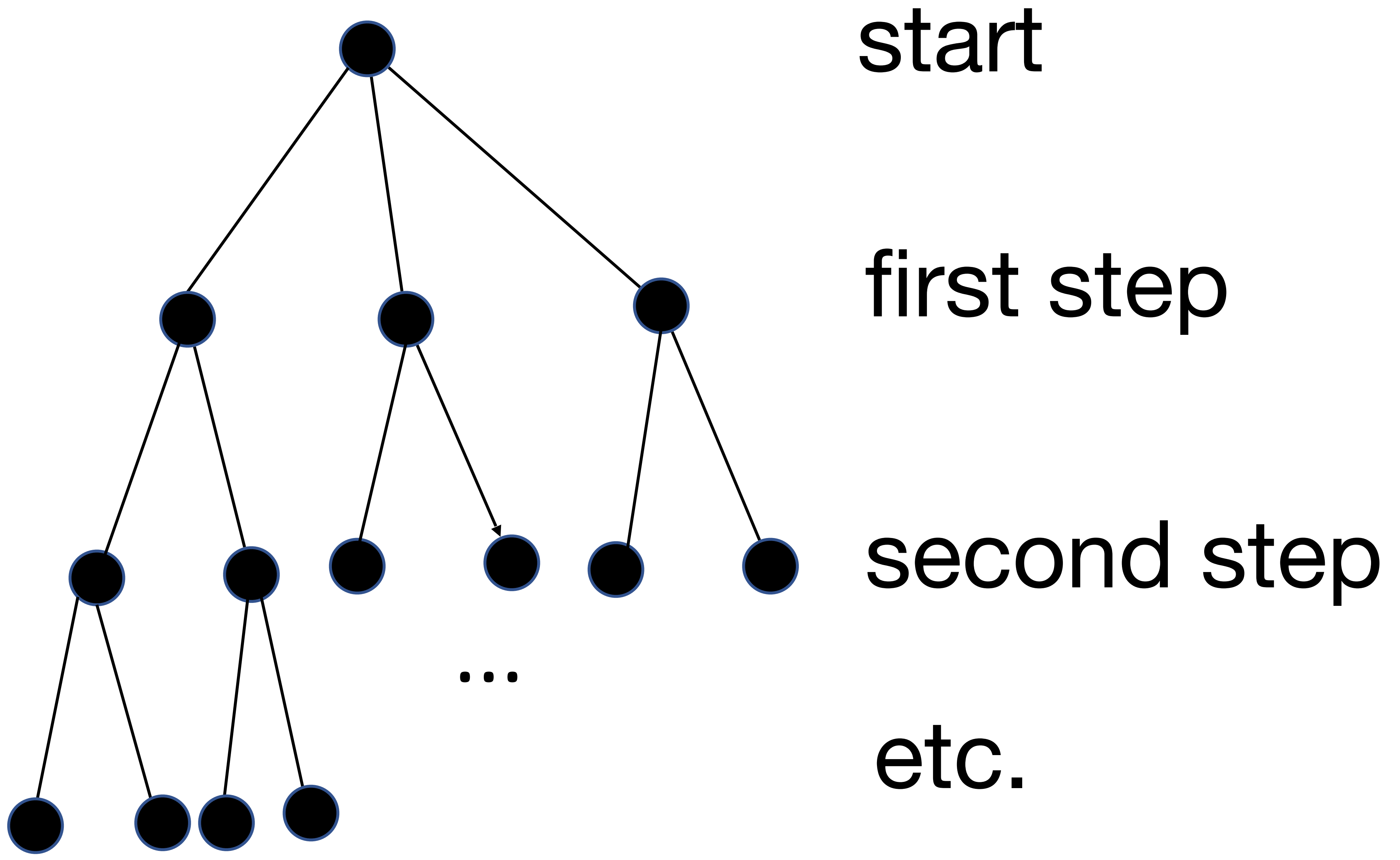
- Not efficient: often exponential time
  - Thus it only works on small problems

+ Fairly easy to implement for a wide class of problems

Types of problems

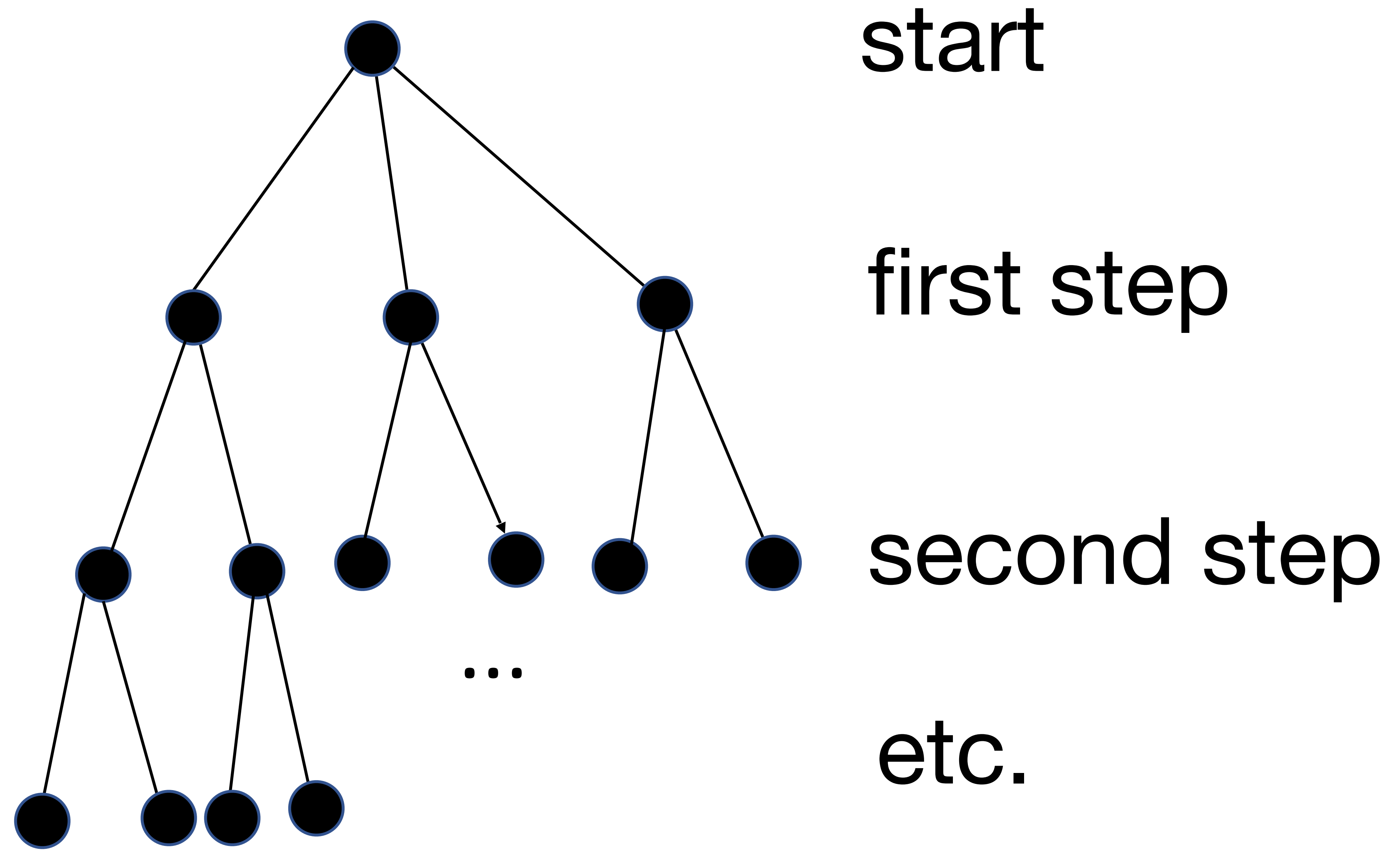
To apply backtracking, the problem needs to have solutions that can be built one step at a time

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

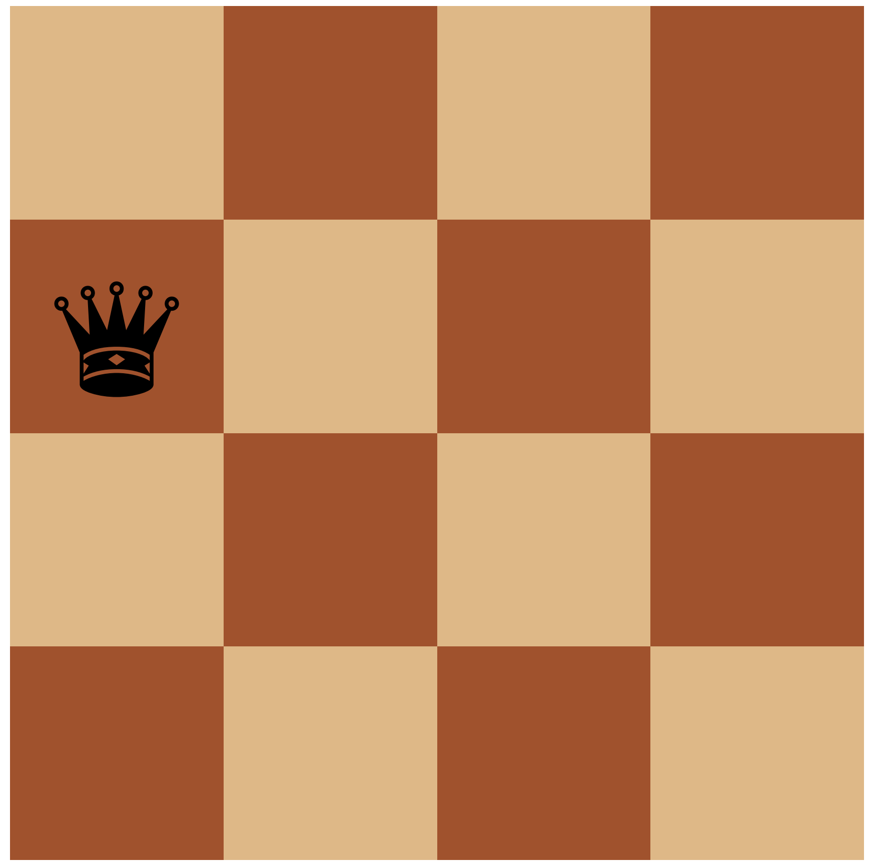
If the partial solution is not valid, backtrack

Repeat until you have a valid solution

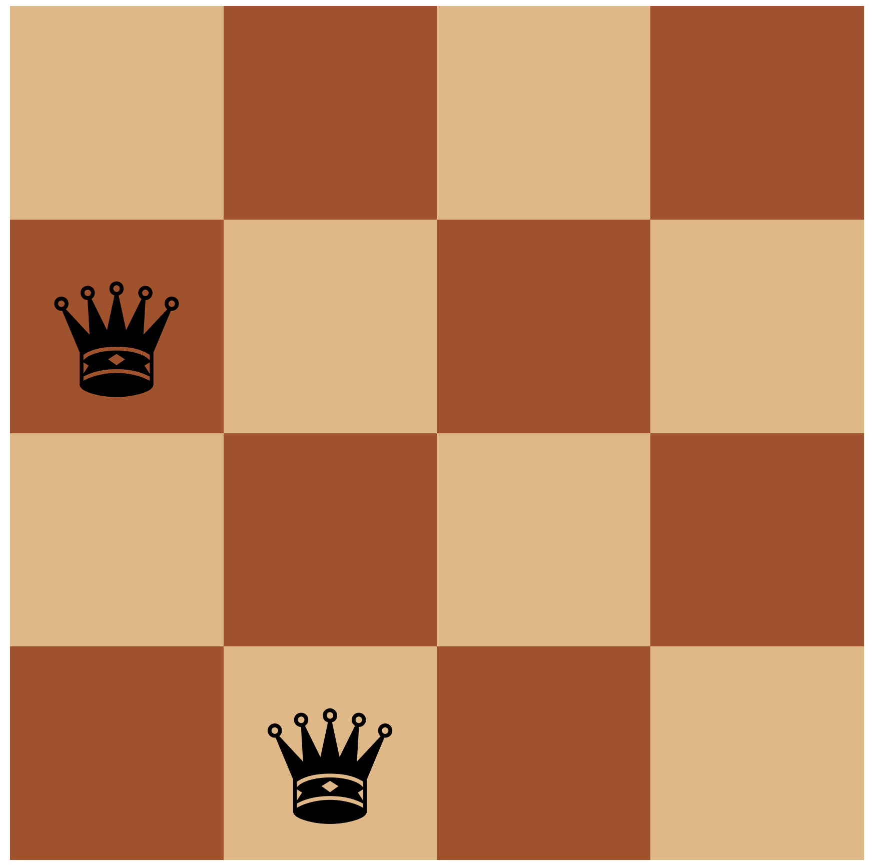
Example: *n* = 4

(Backtracking steps omitted)

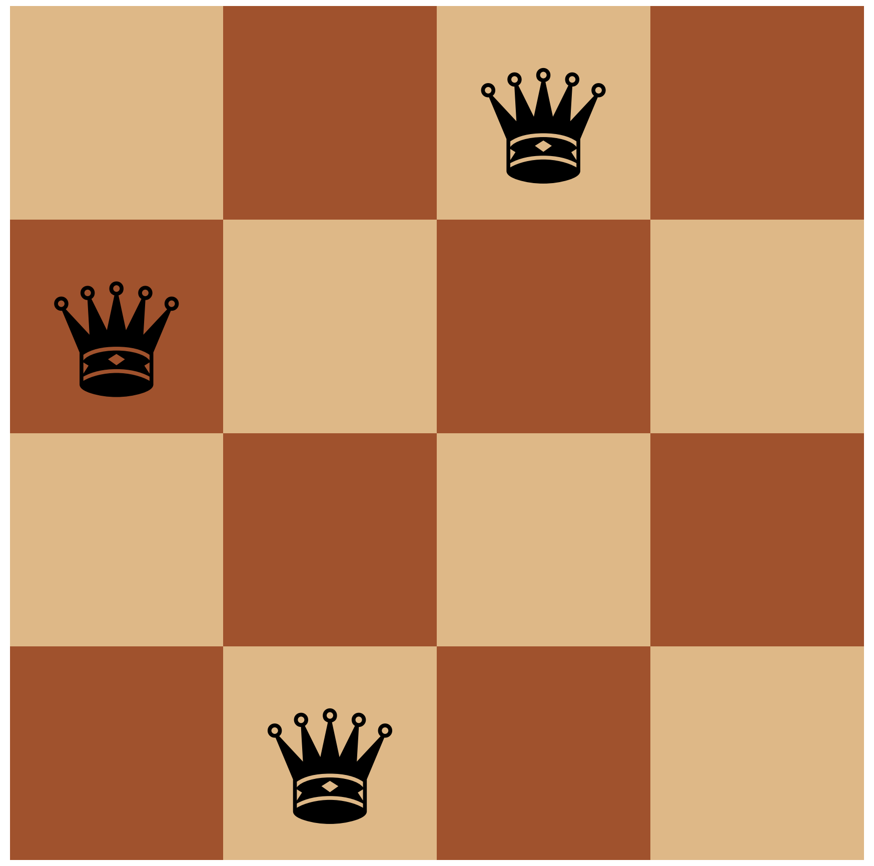
Step 1



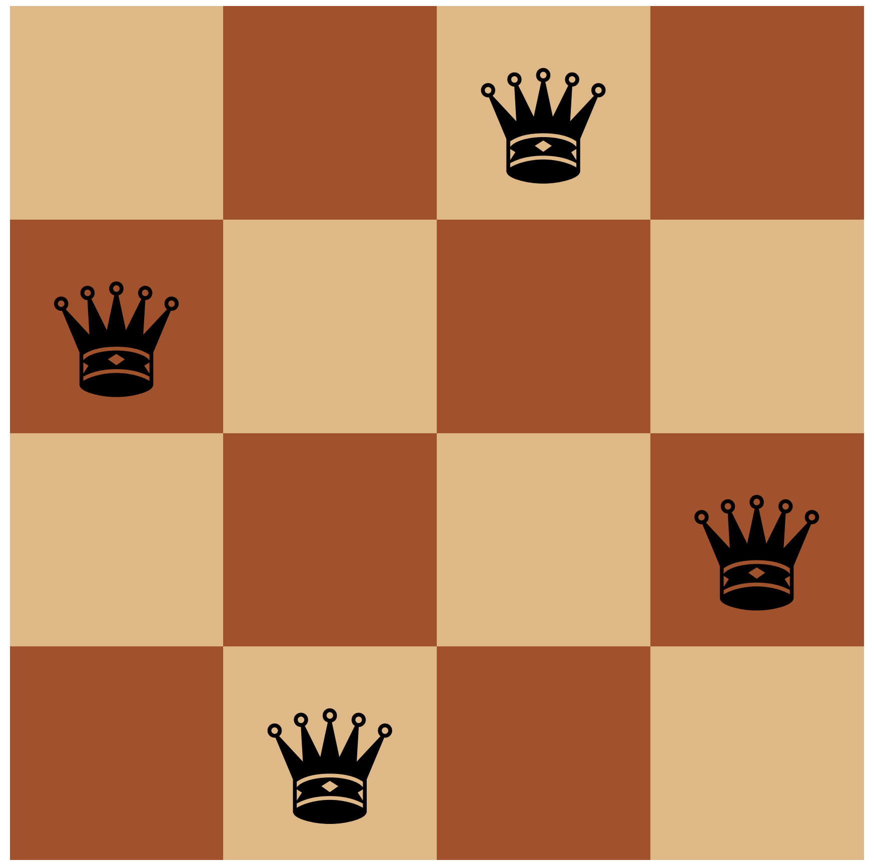
Step 2



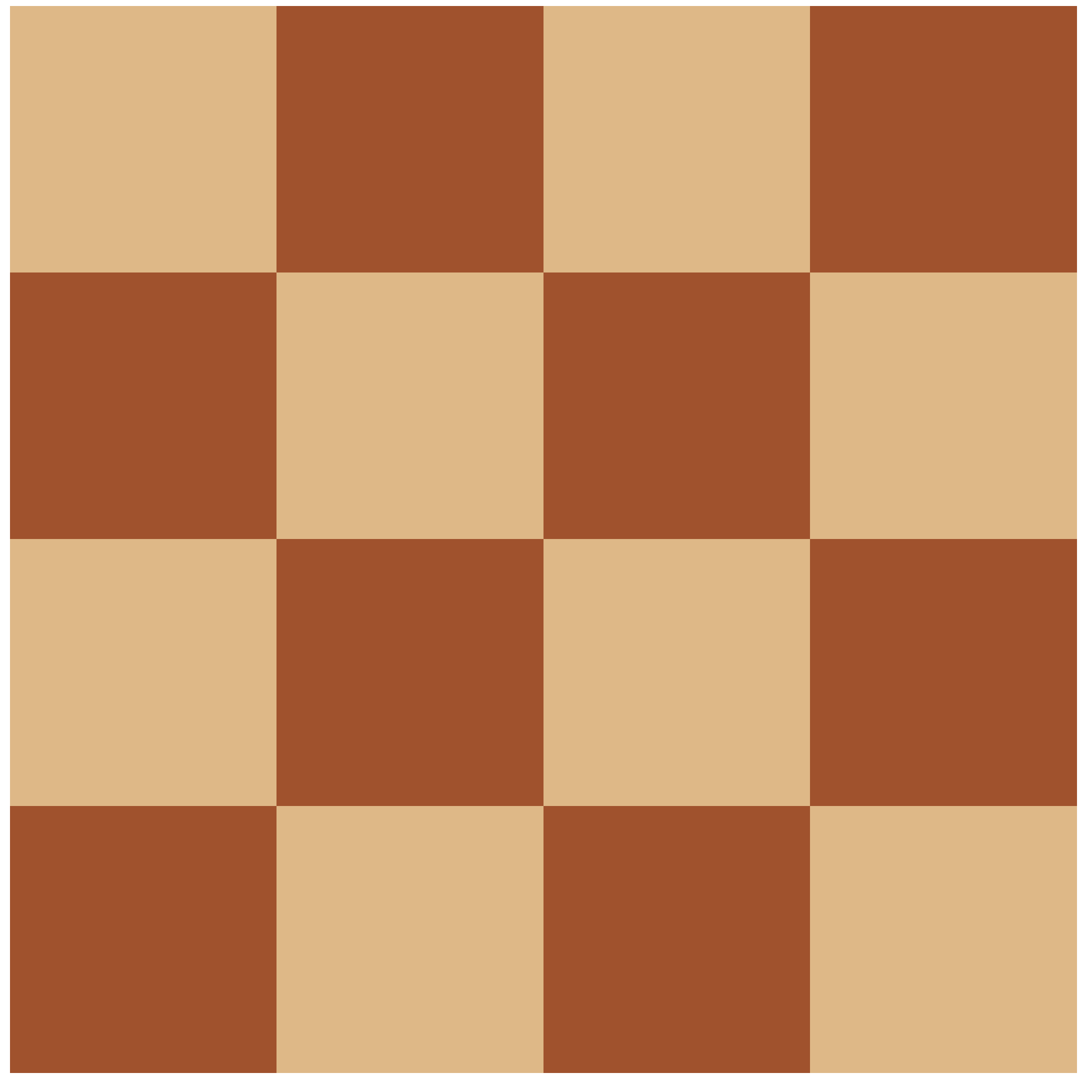
Step 3



Step 4

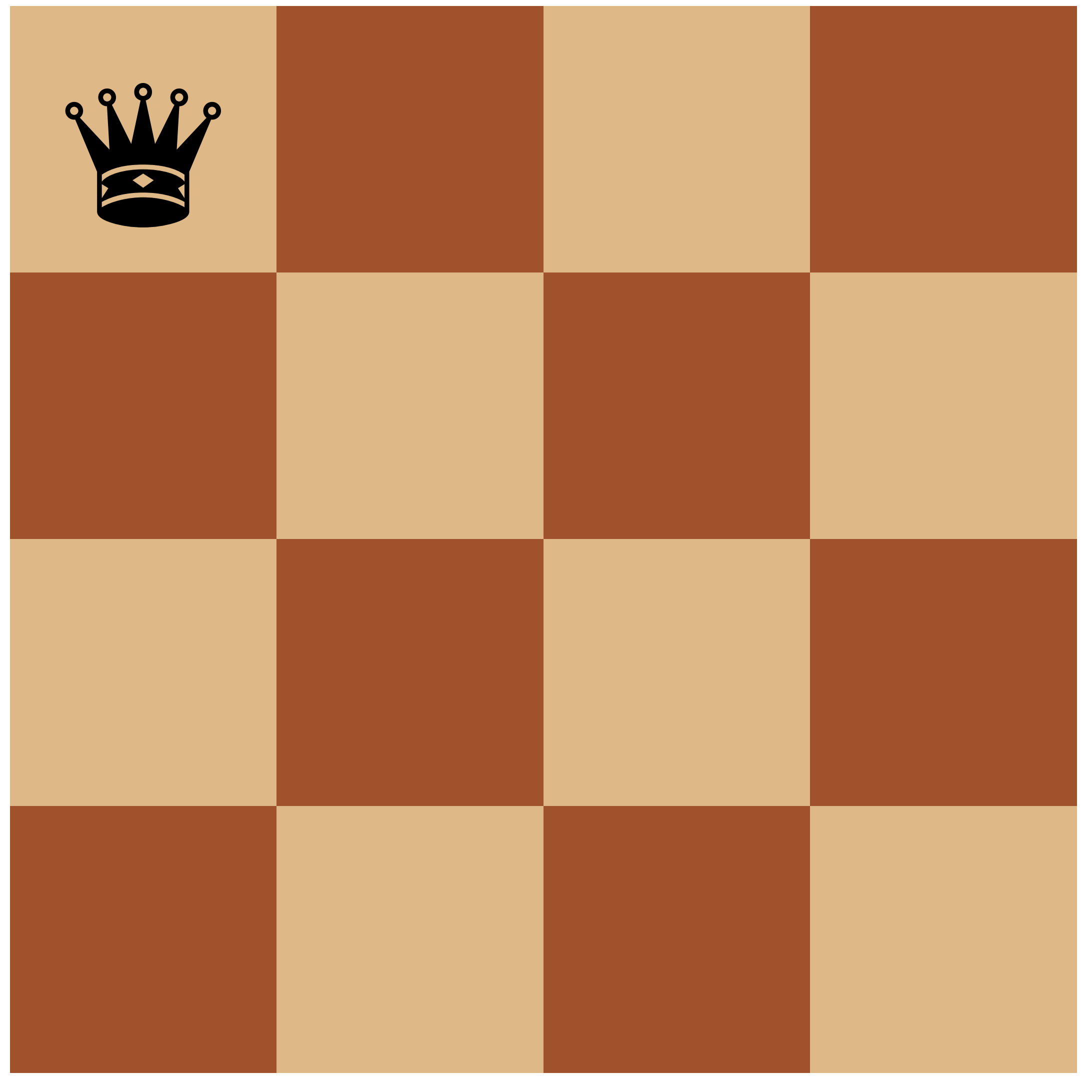


Example: $n = 4$



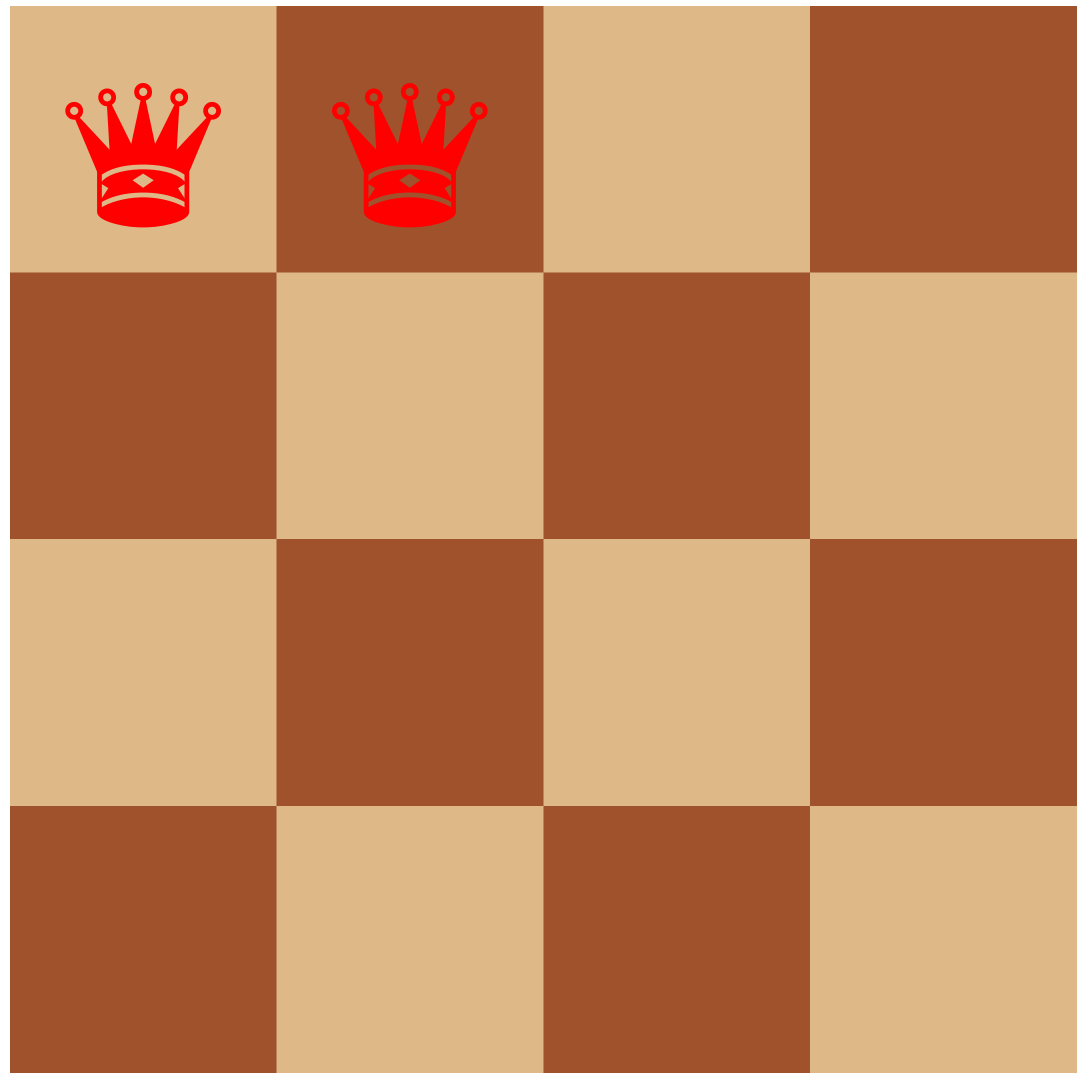
Initial state

Example: n = 4



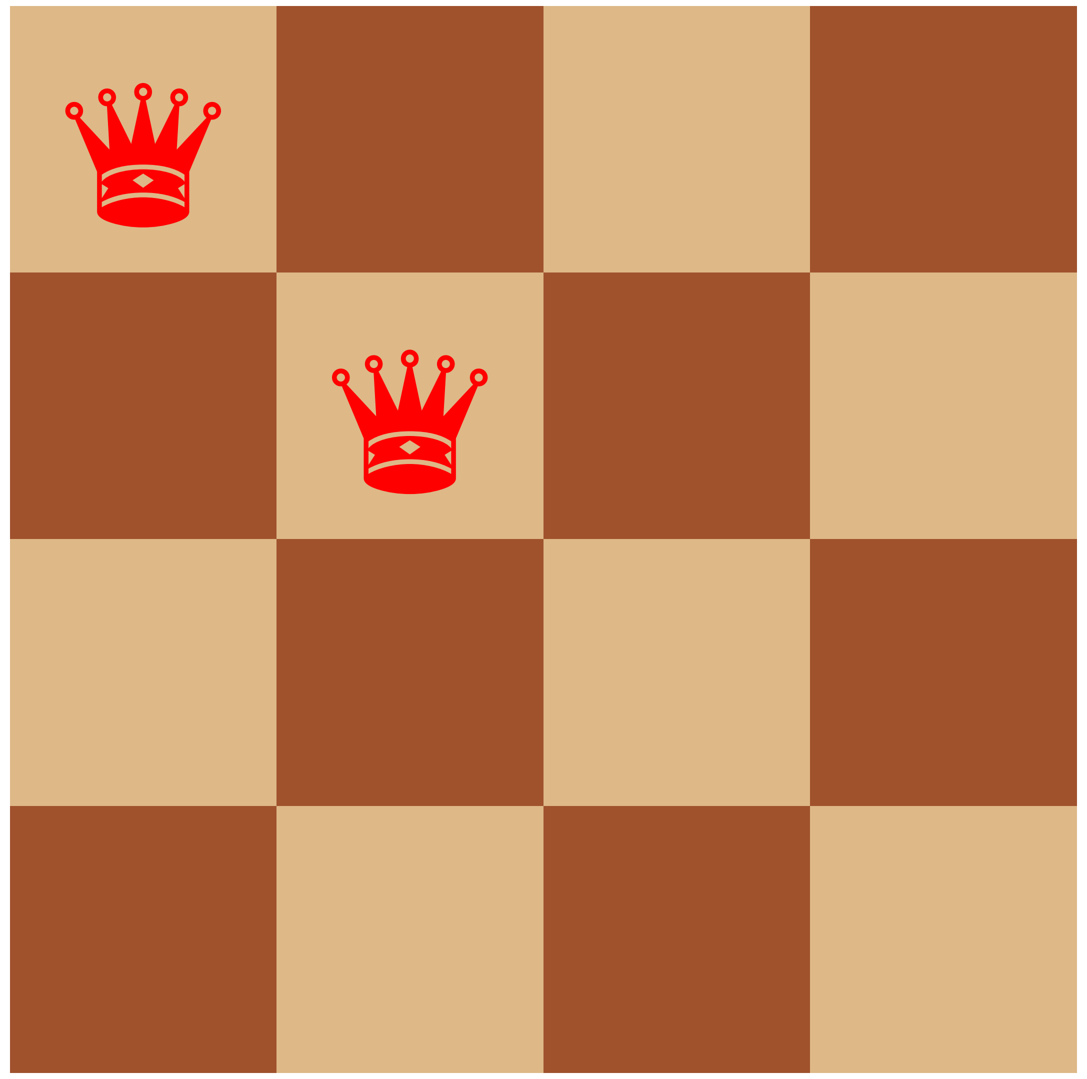
Step 1

Example: n = 4



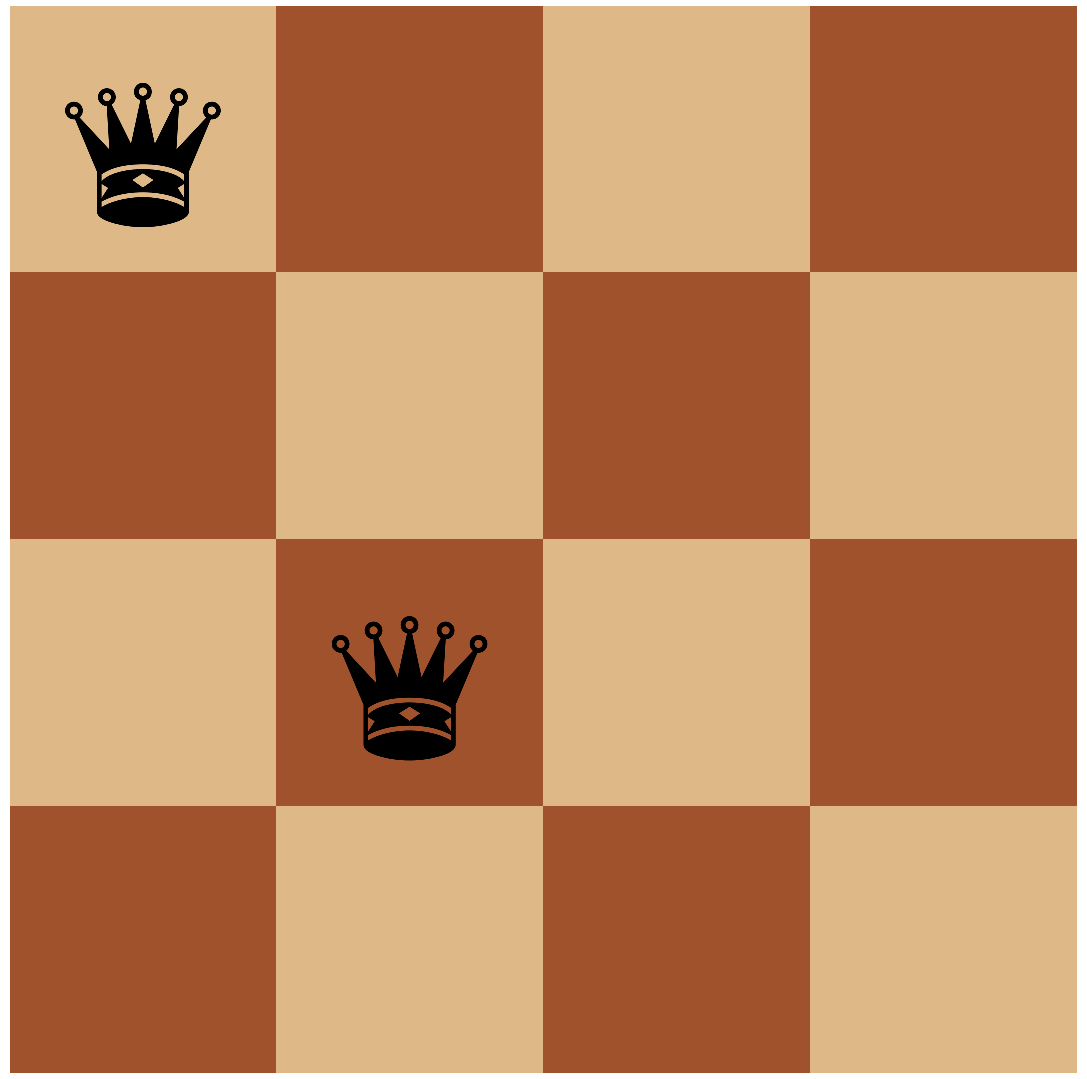
Step 2

Example: n = 4



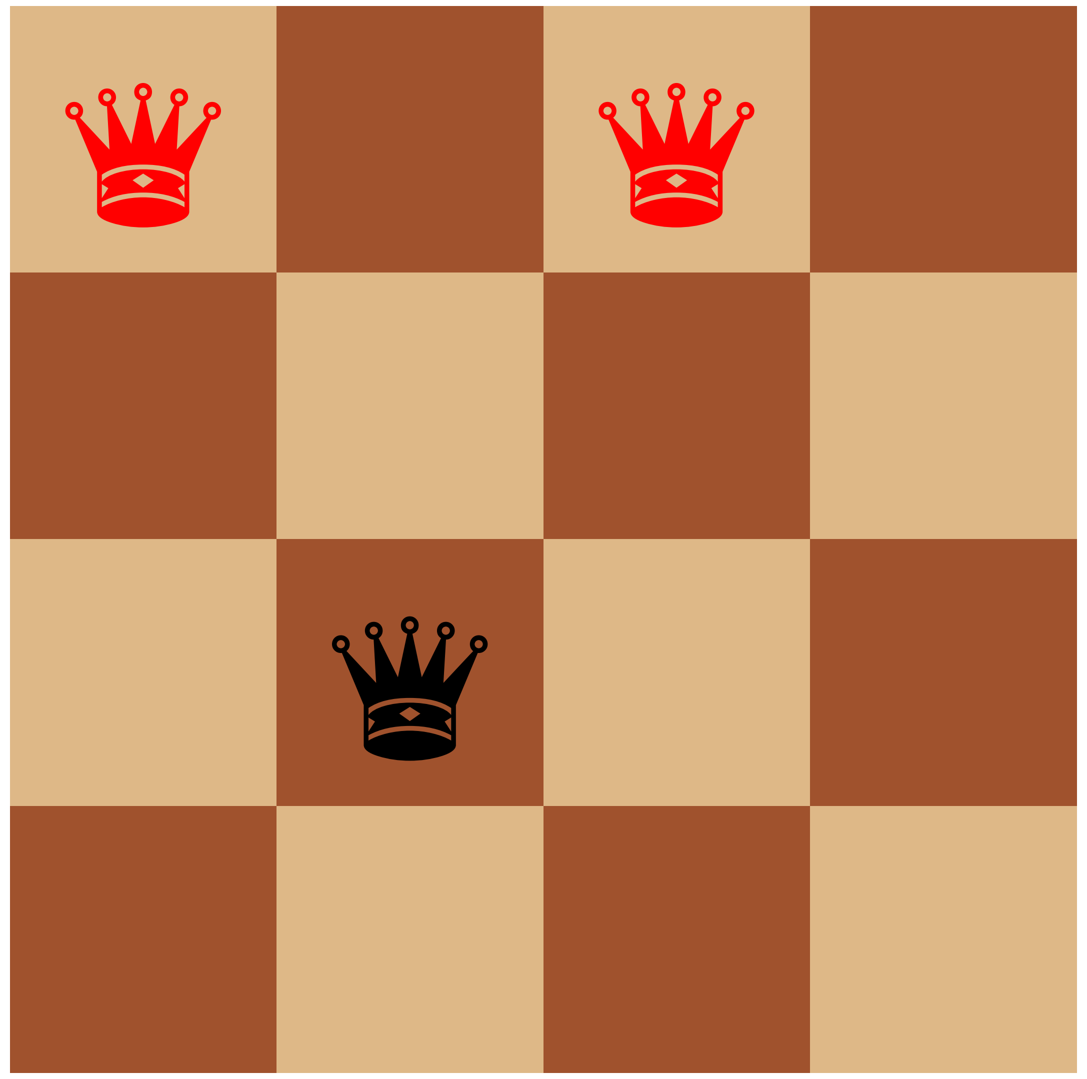
Step 3

Example: $n = 4$



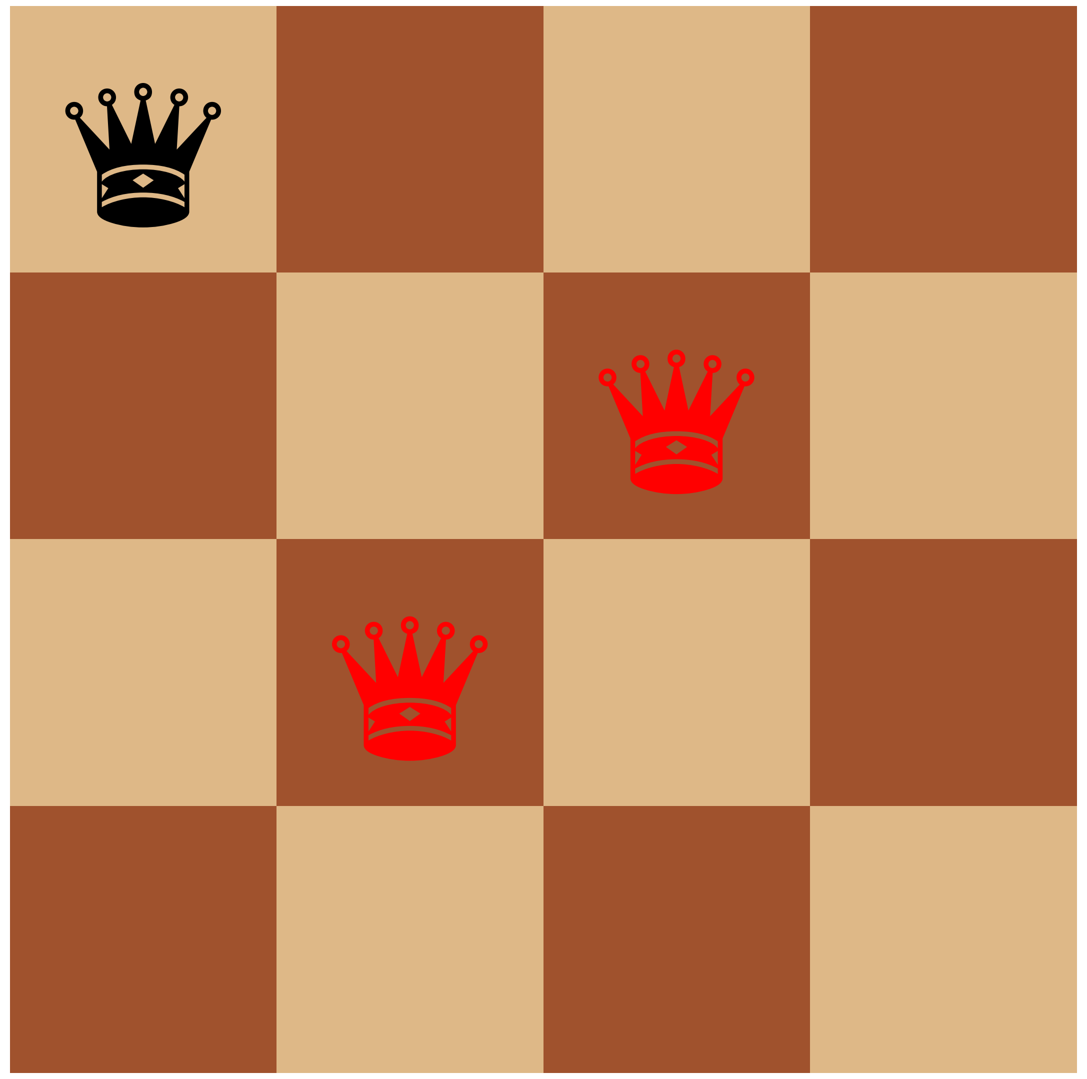
Step 4

Example: $n = 4$



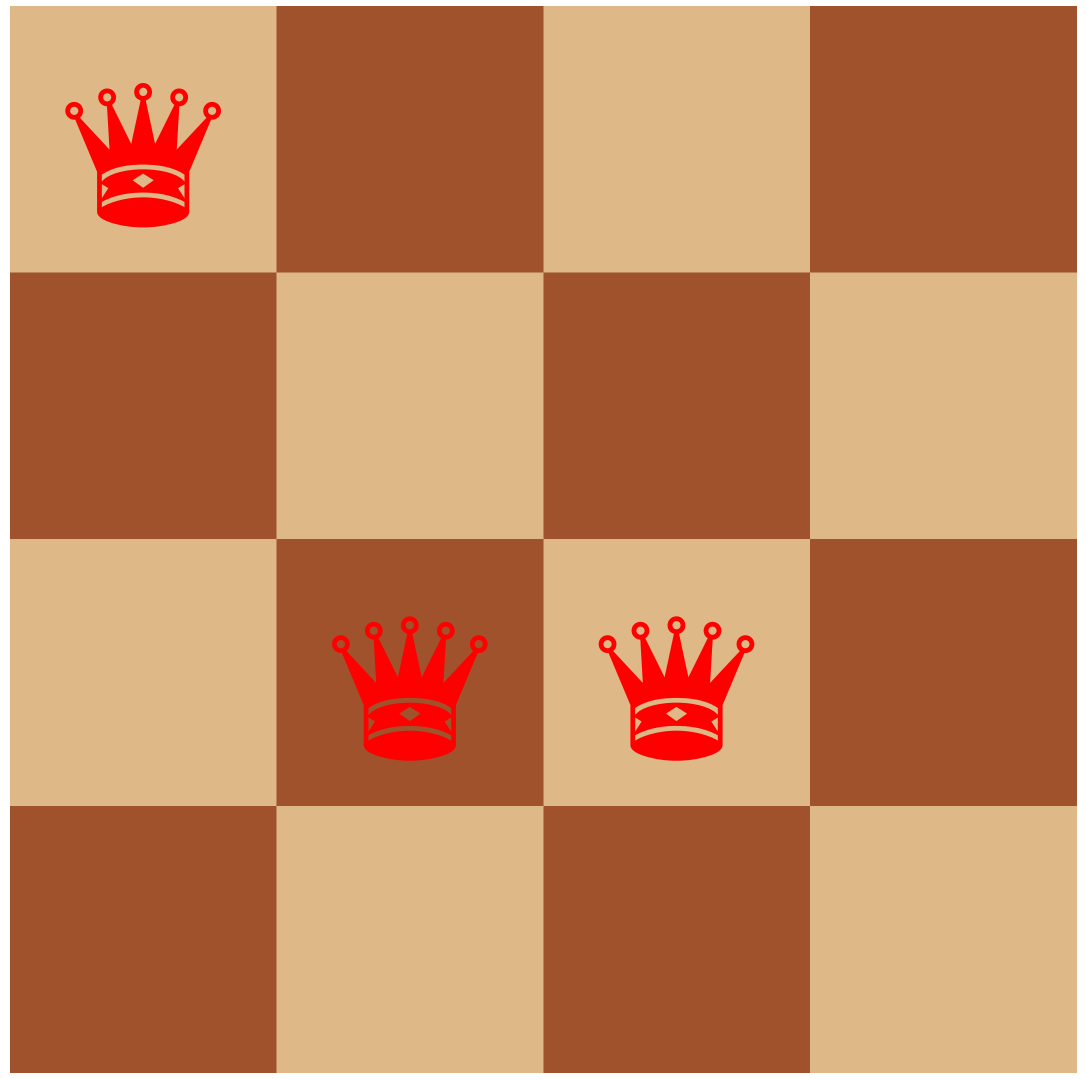
Step 5

Example: $n = 4$



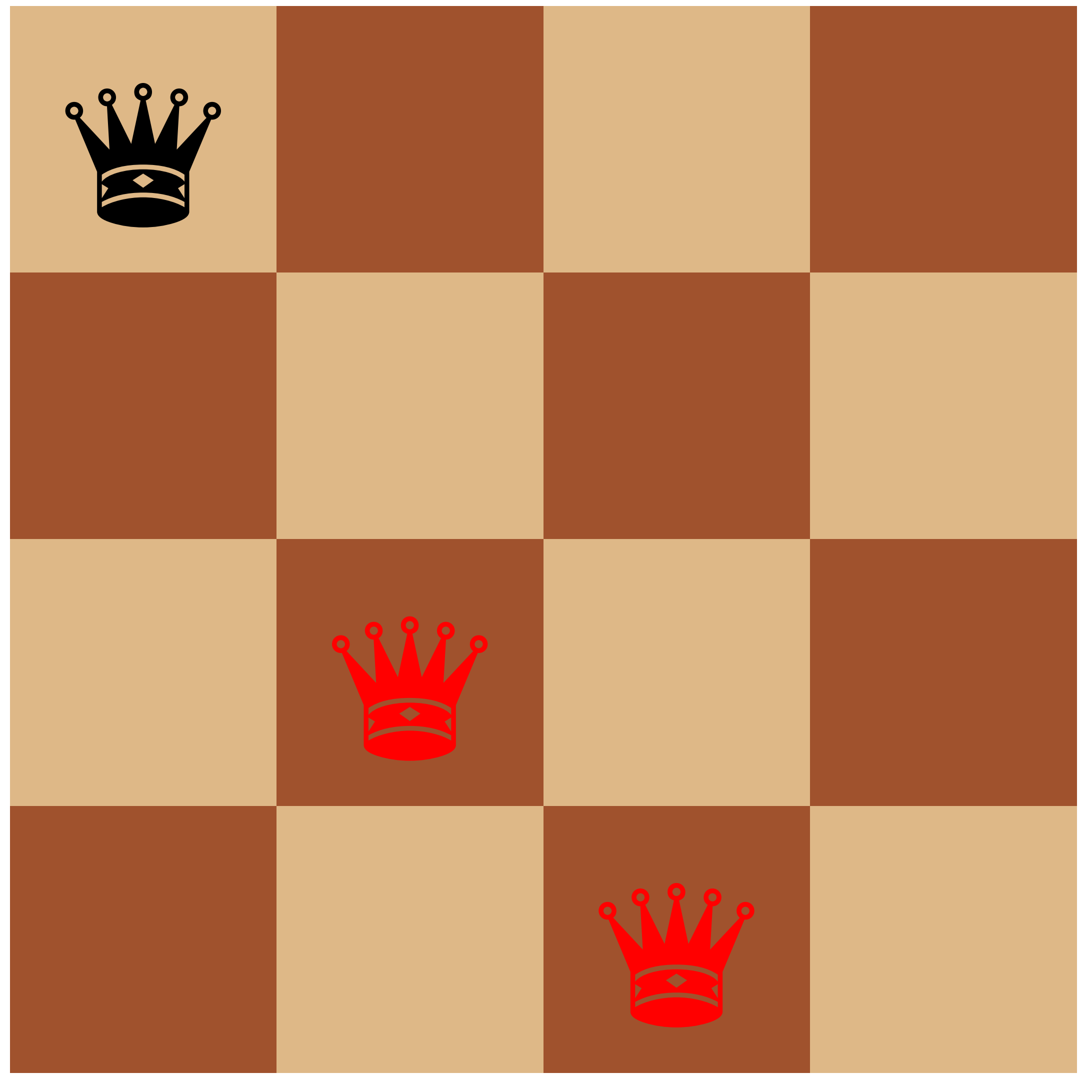
Step 6

Example: $n = 4$



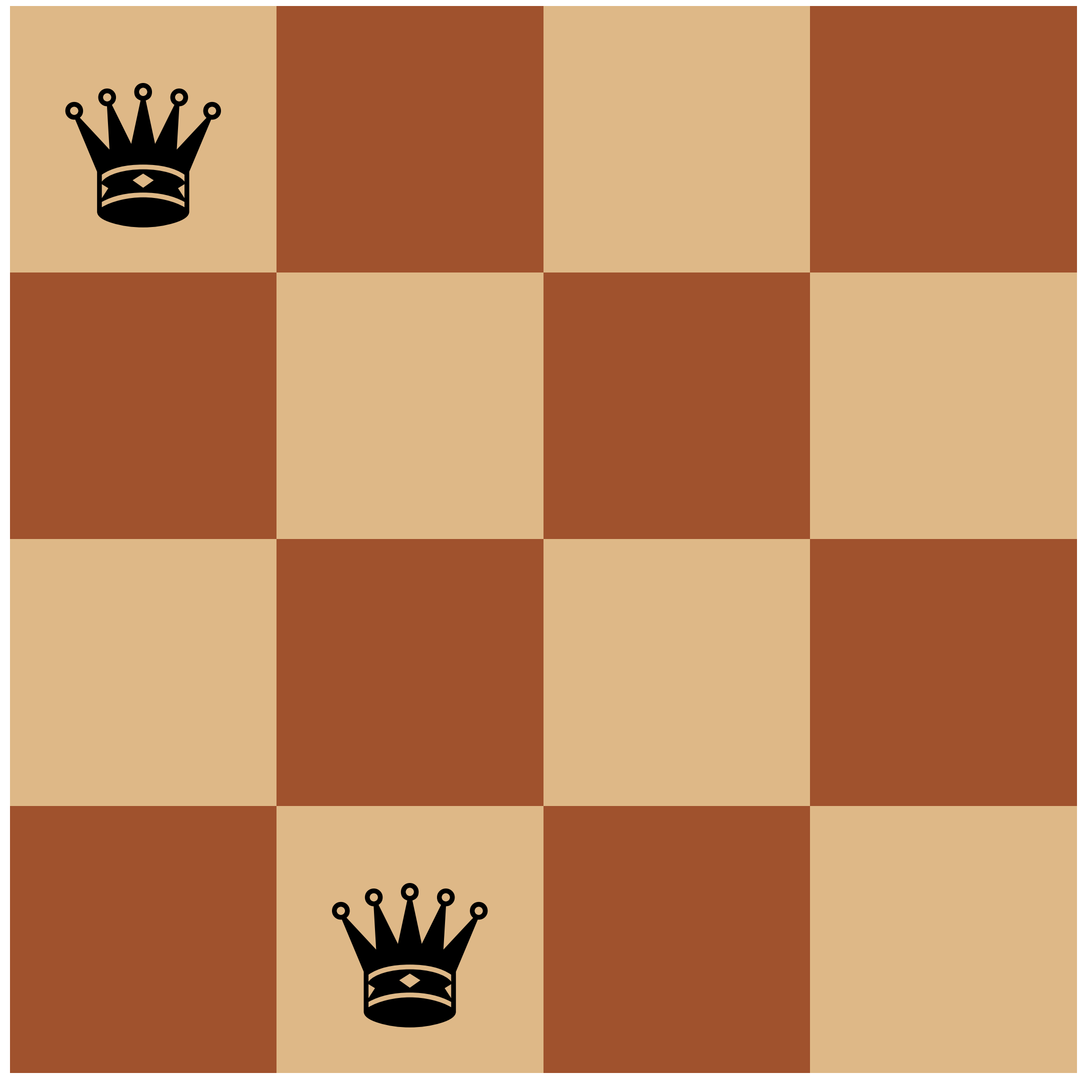
Step 7

Example: n = 4



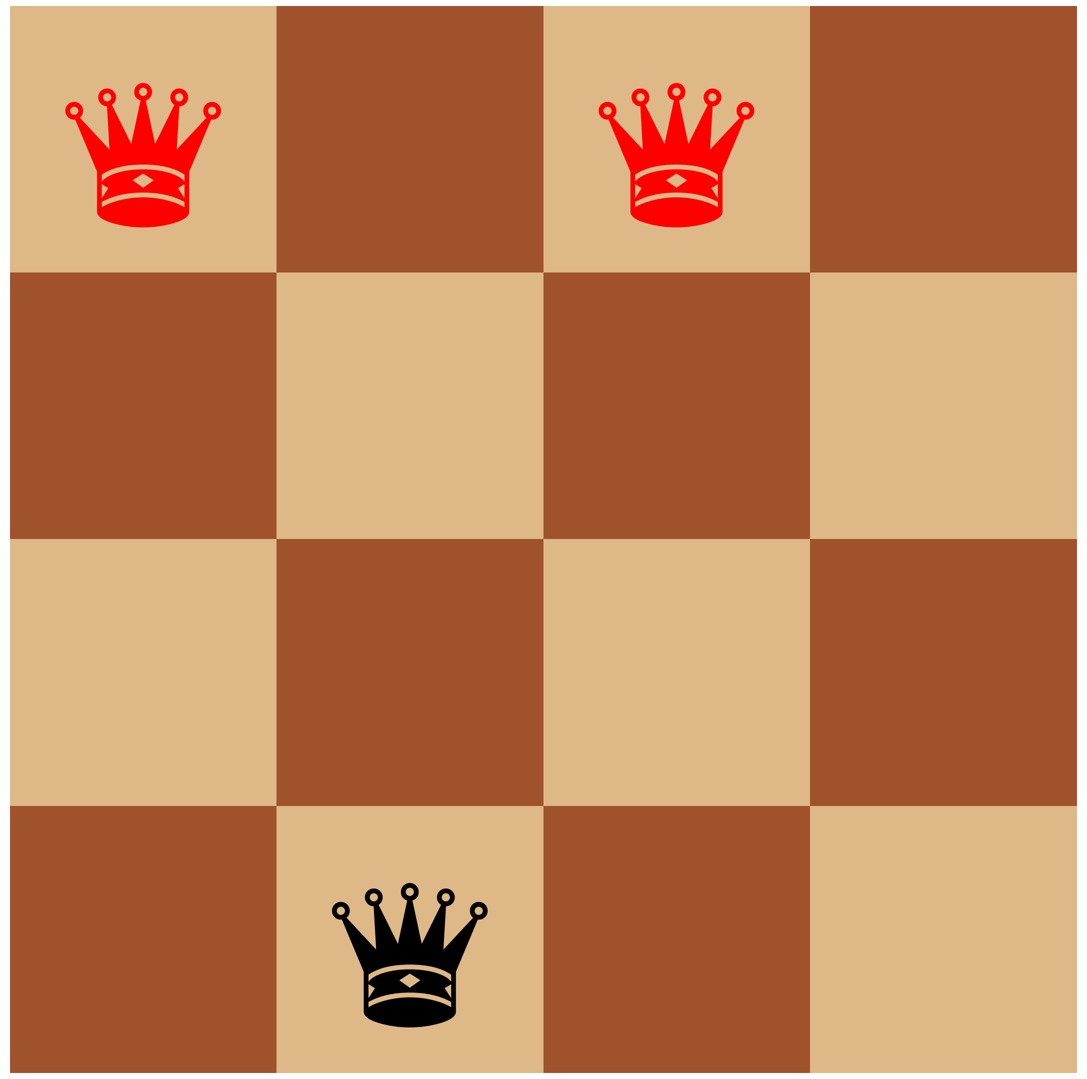
Step 8

Example: n = 4



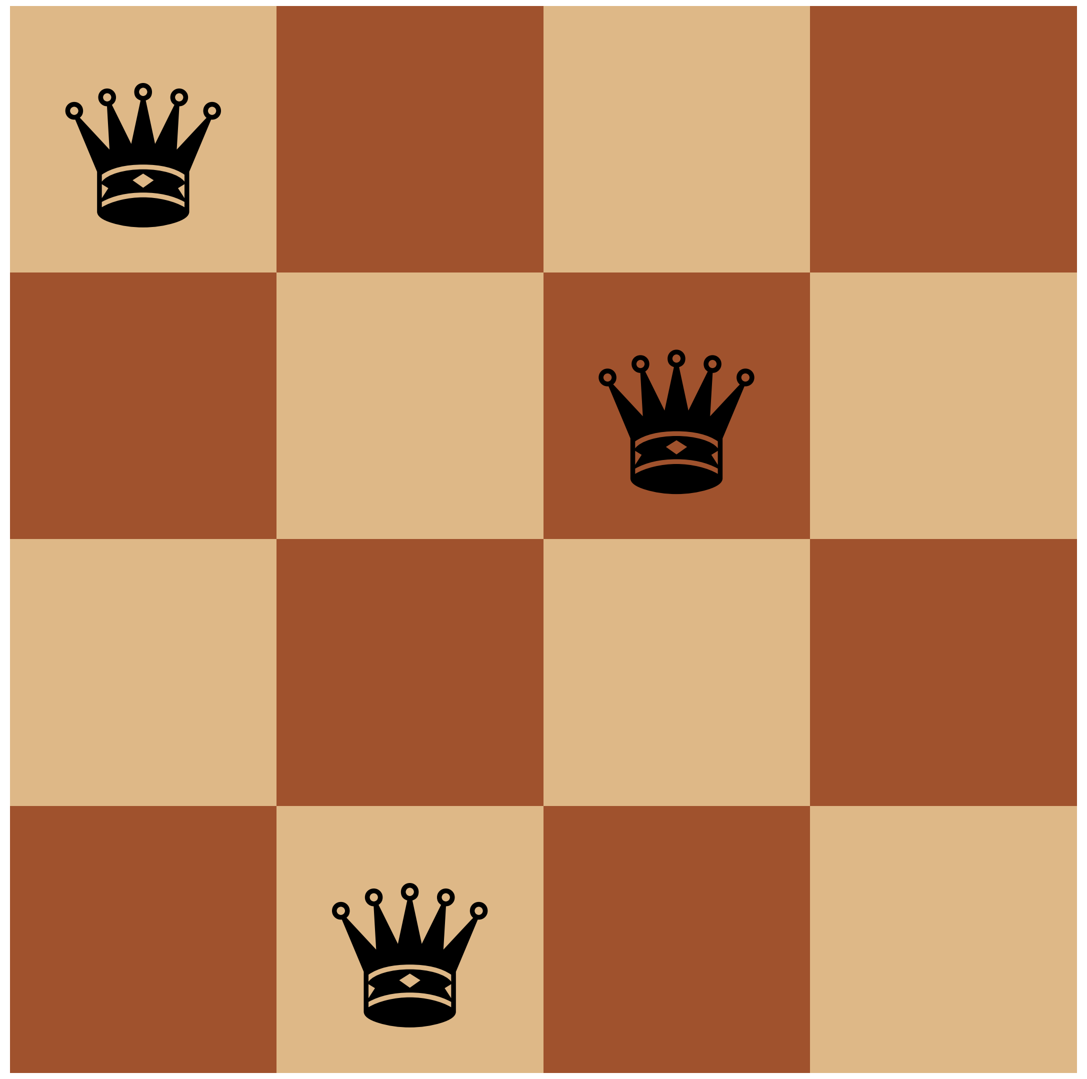
Step 9

Example: n = 4



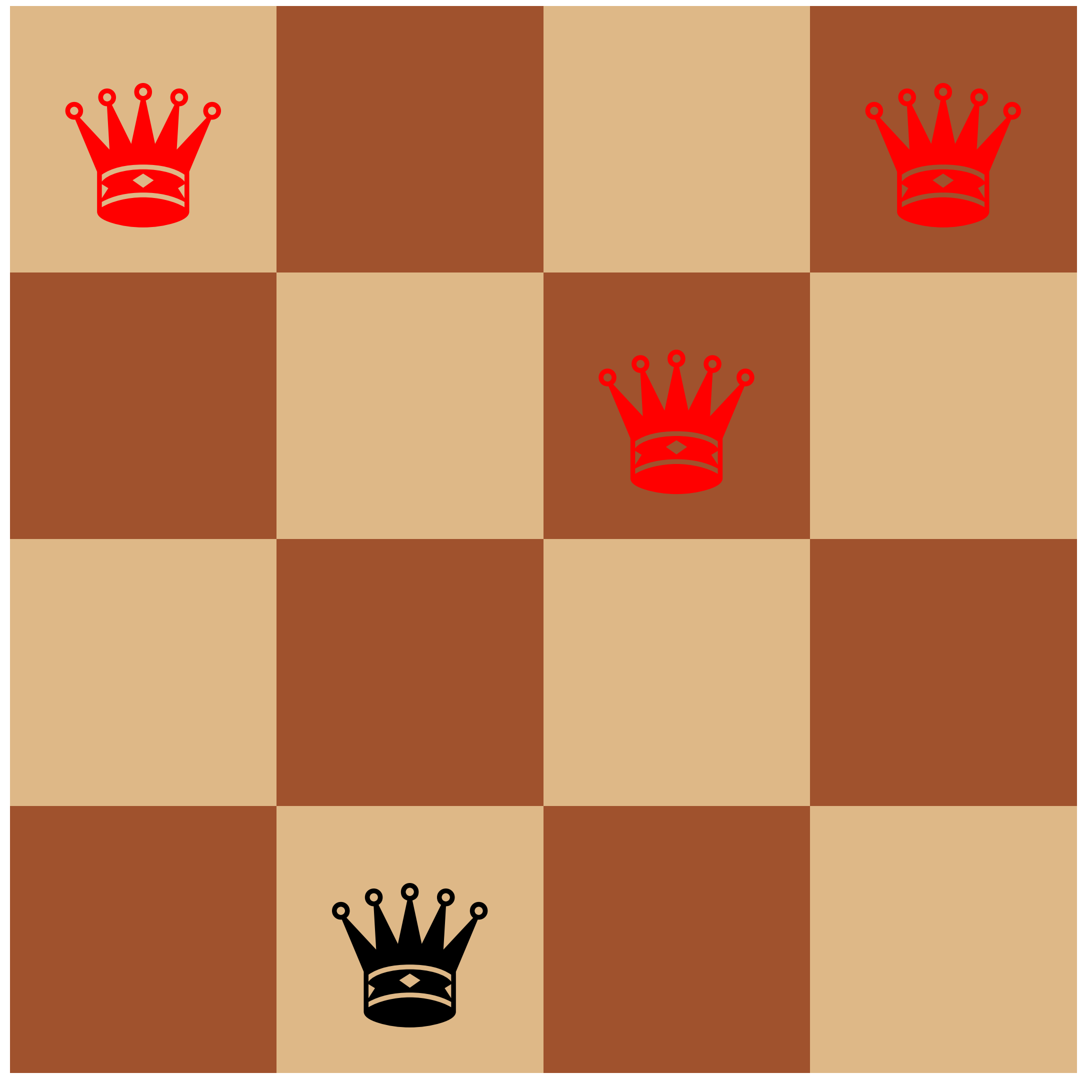
Step 10

Example: $n = 4$



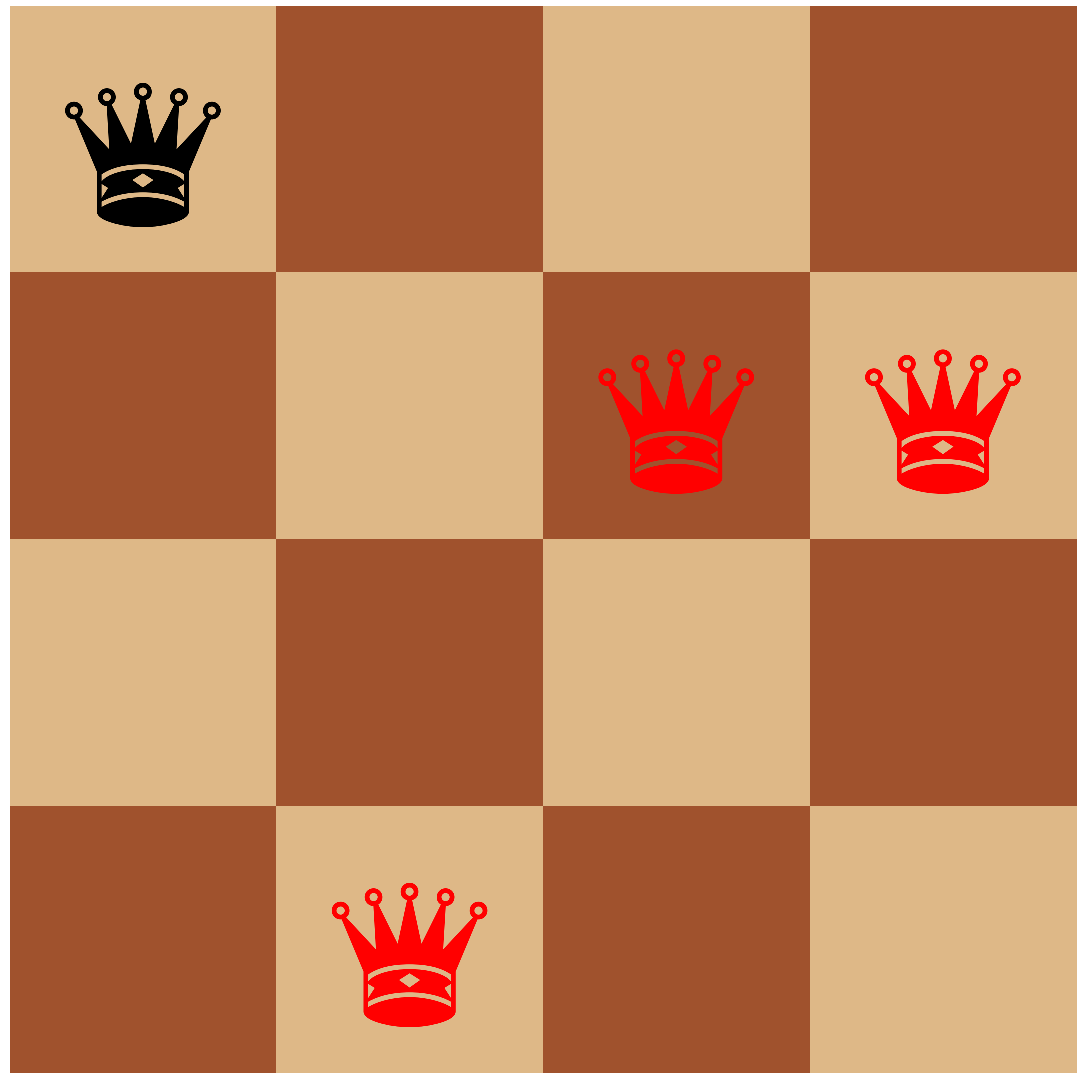
Step 11

Example: $n = 4$



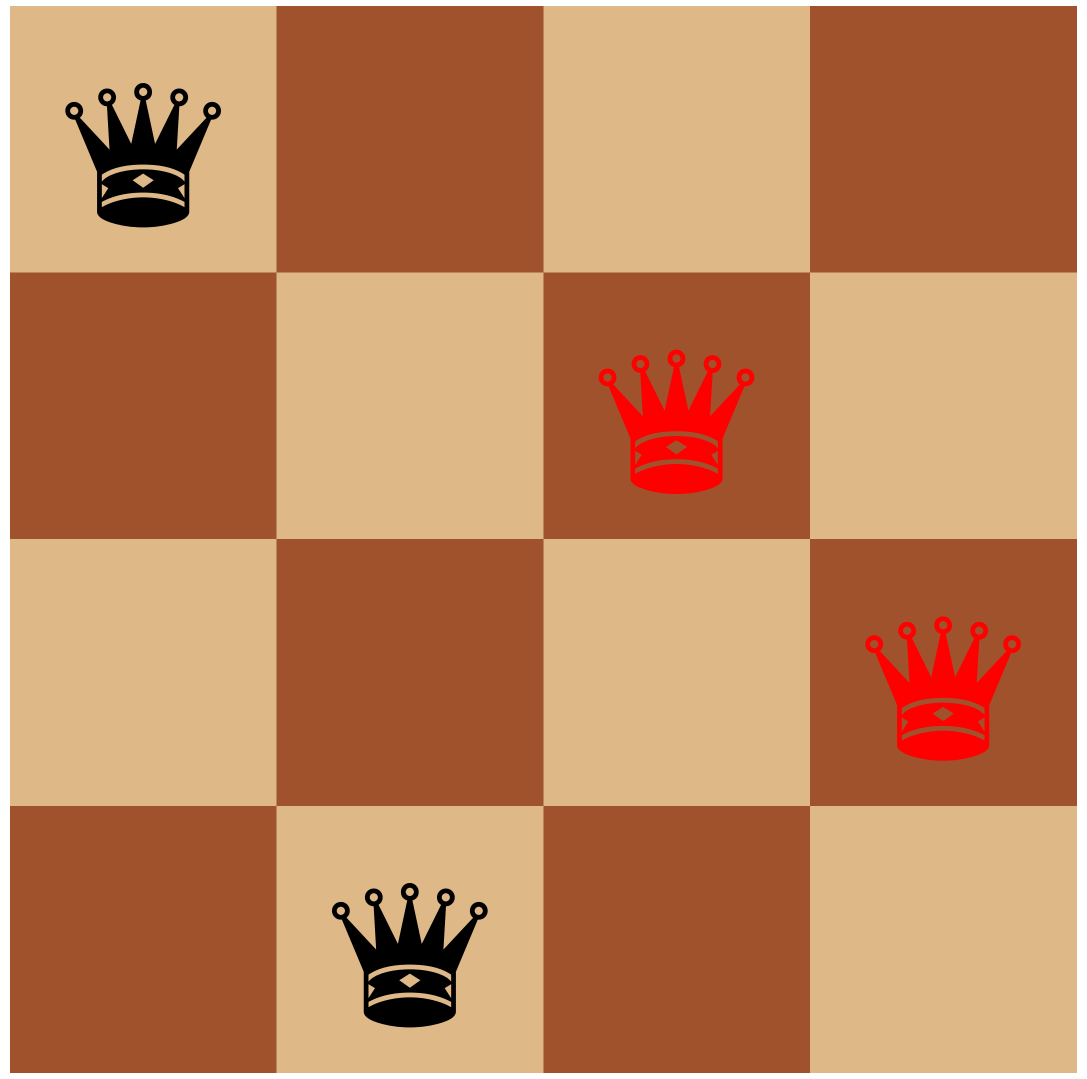
Step 12

Example: $n = 4$



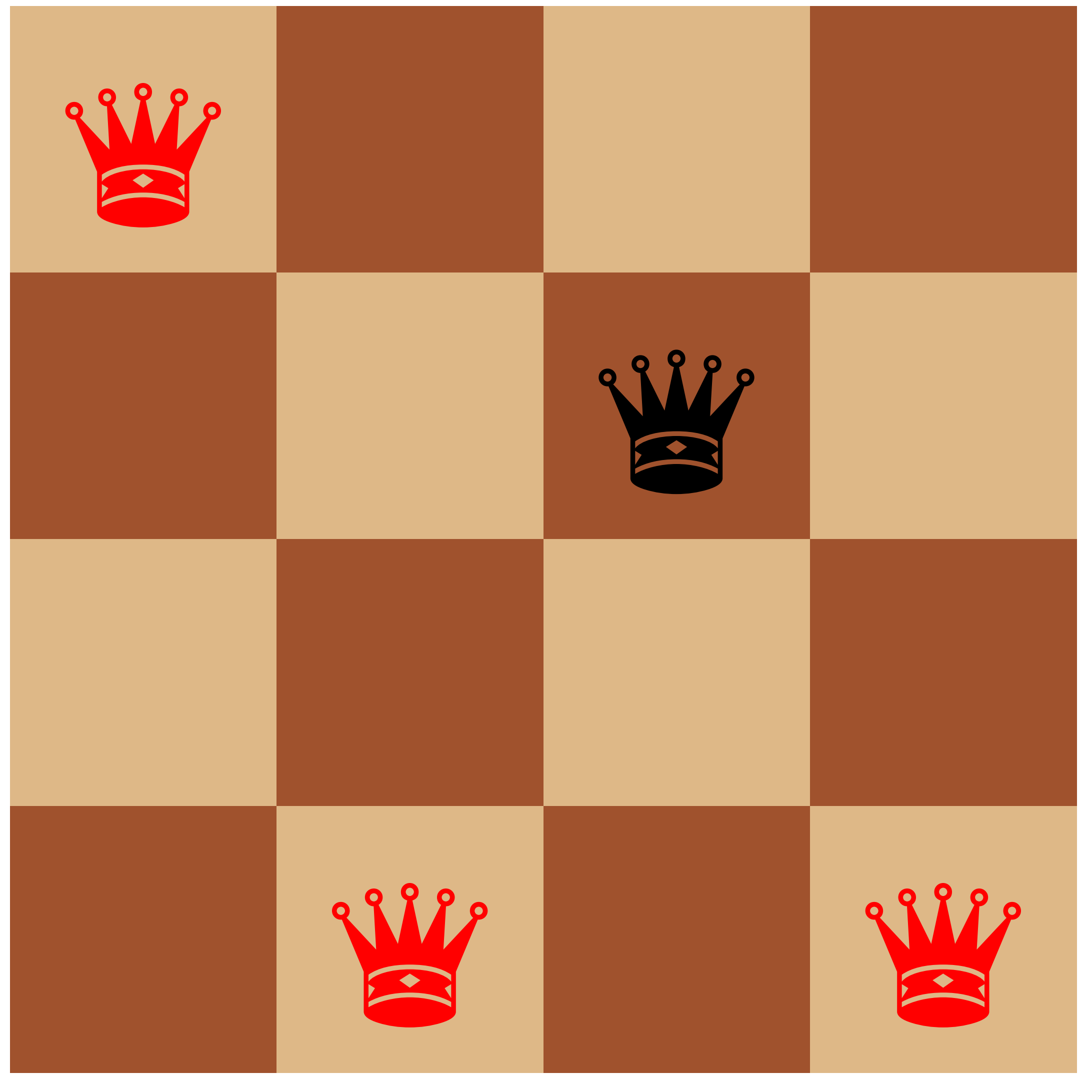
Step 13

Example: $n = 4$



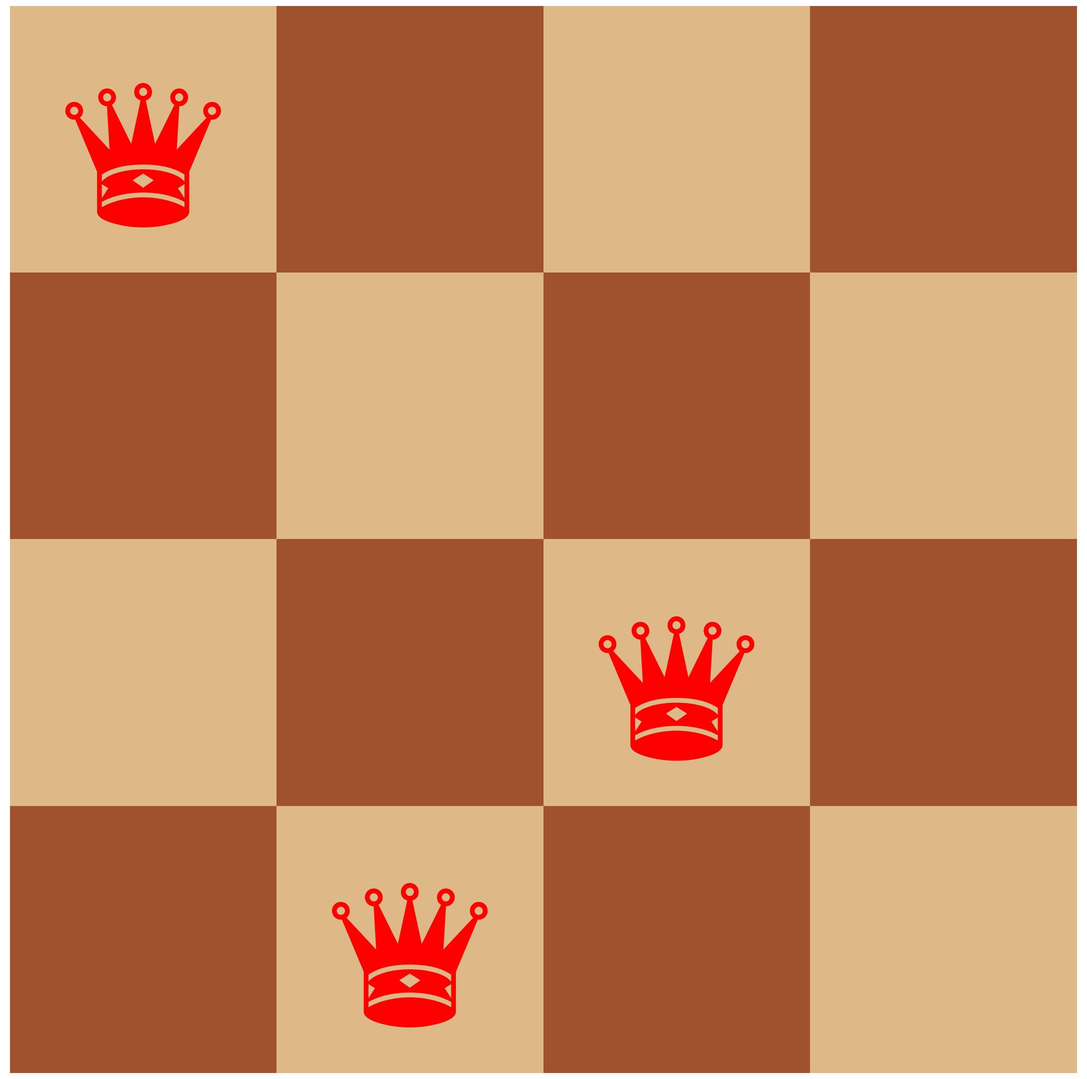
Step 14

Example: n = 4



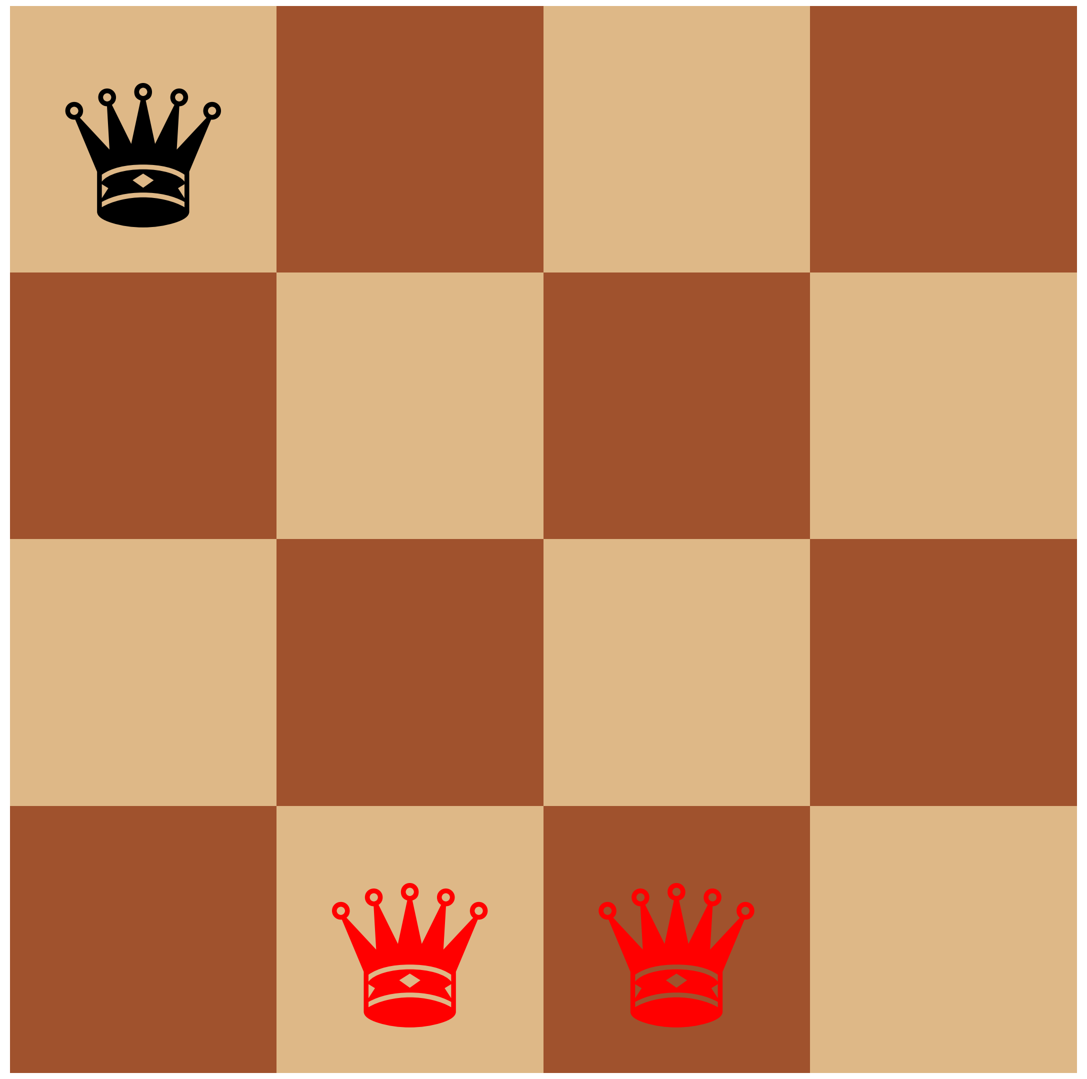
Step 15

Example: $n = 4$



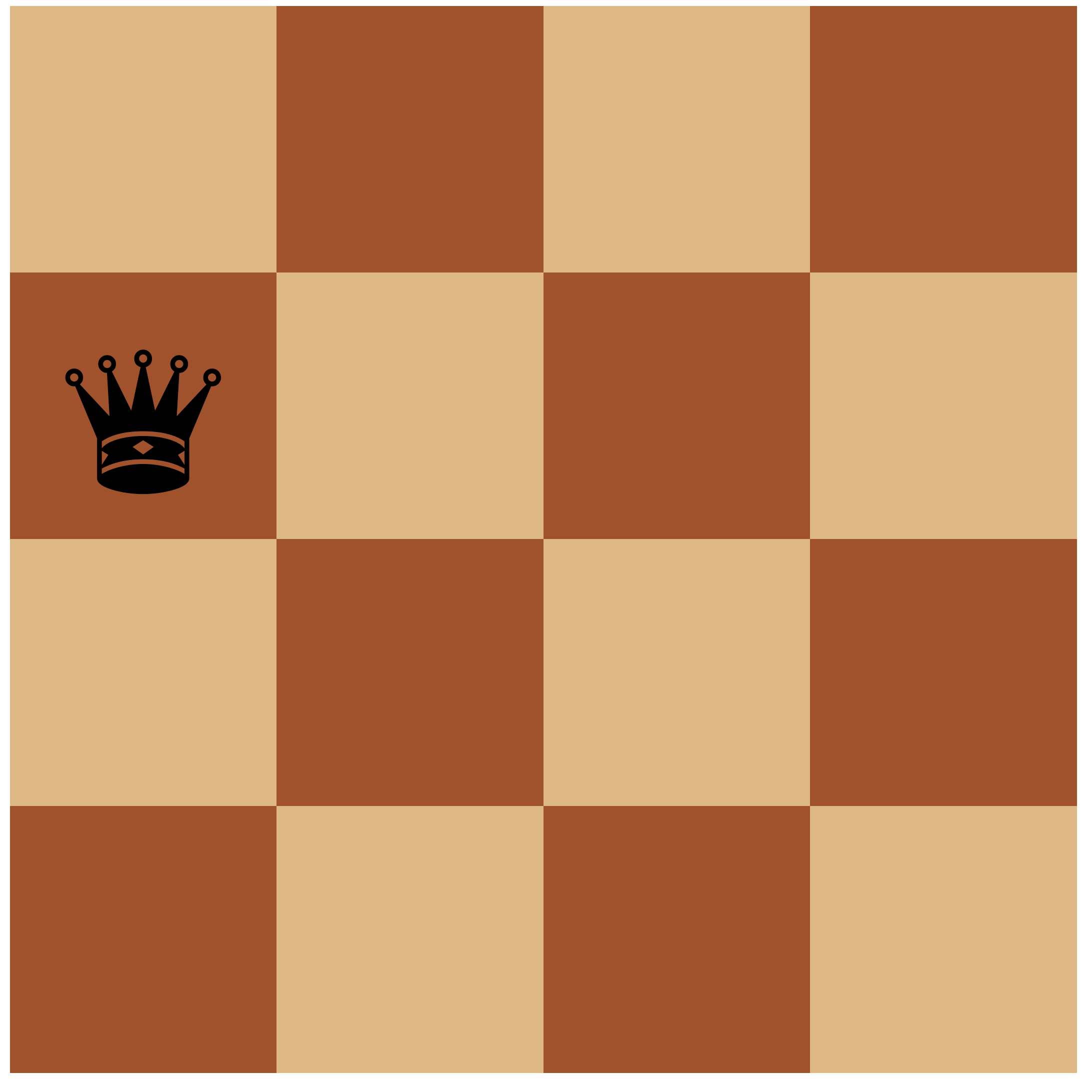
Step 16

Example: $n = 4$



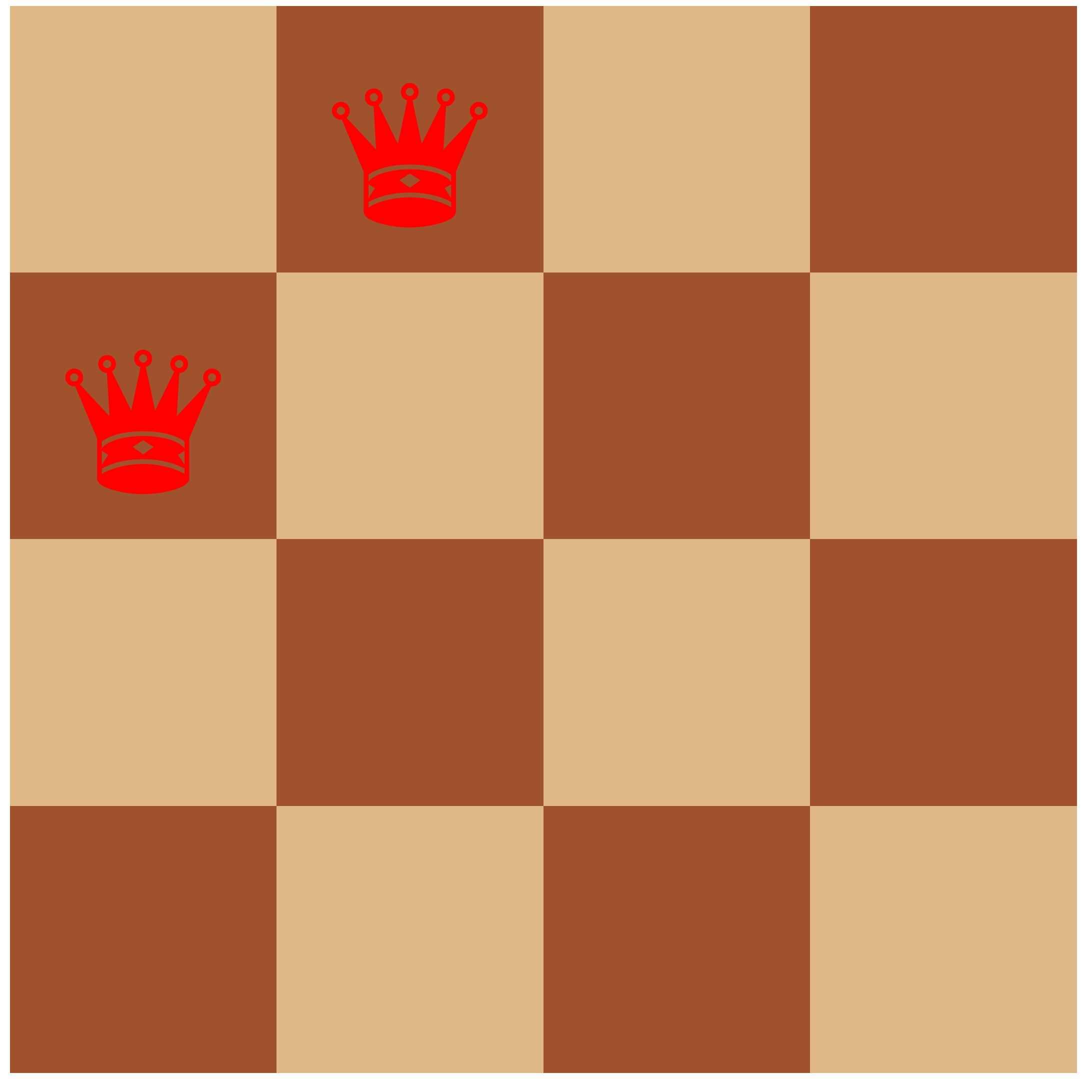
Step 17

Example: $n = 4$



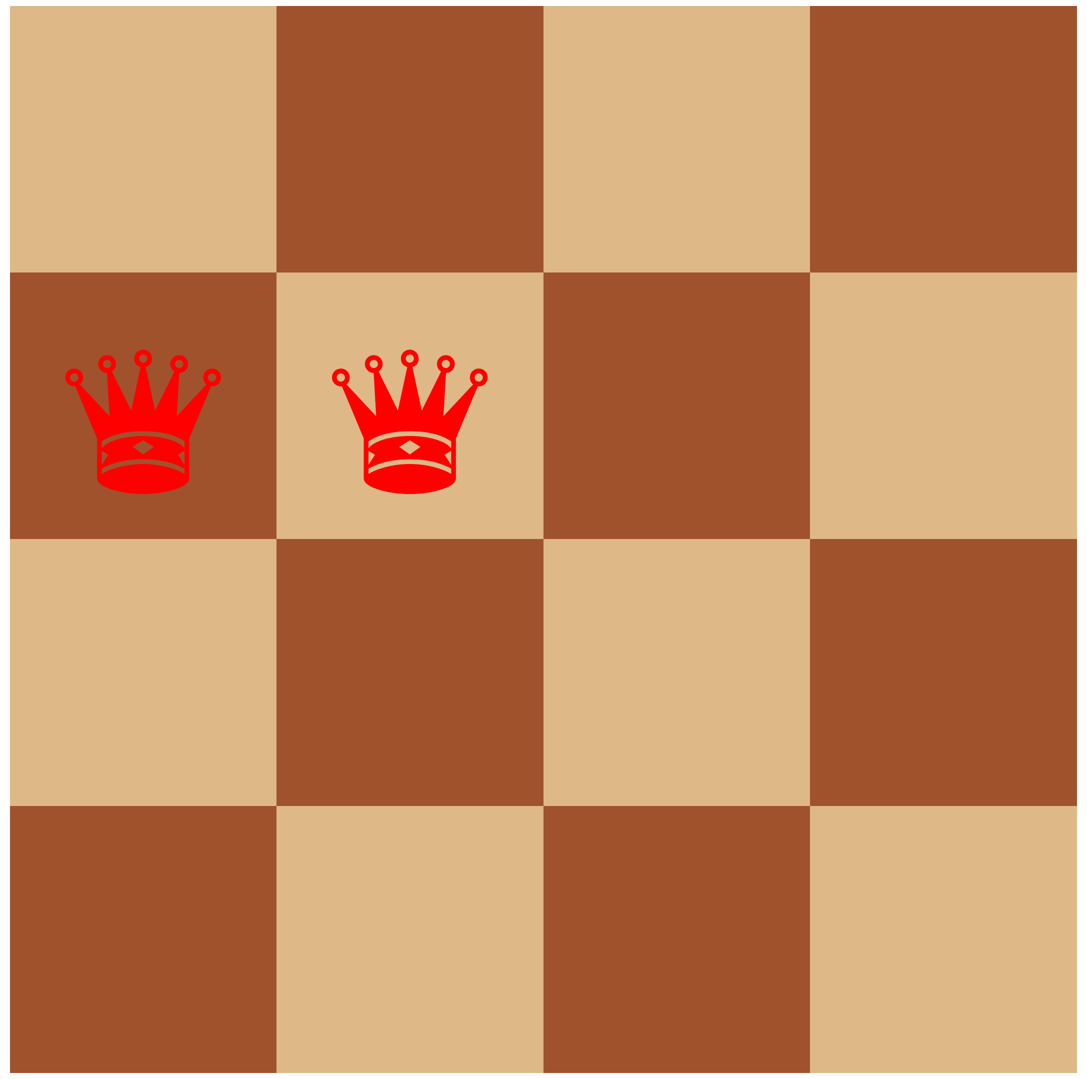
Step 18

Example: $n = 4$



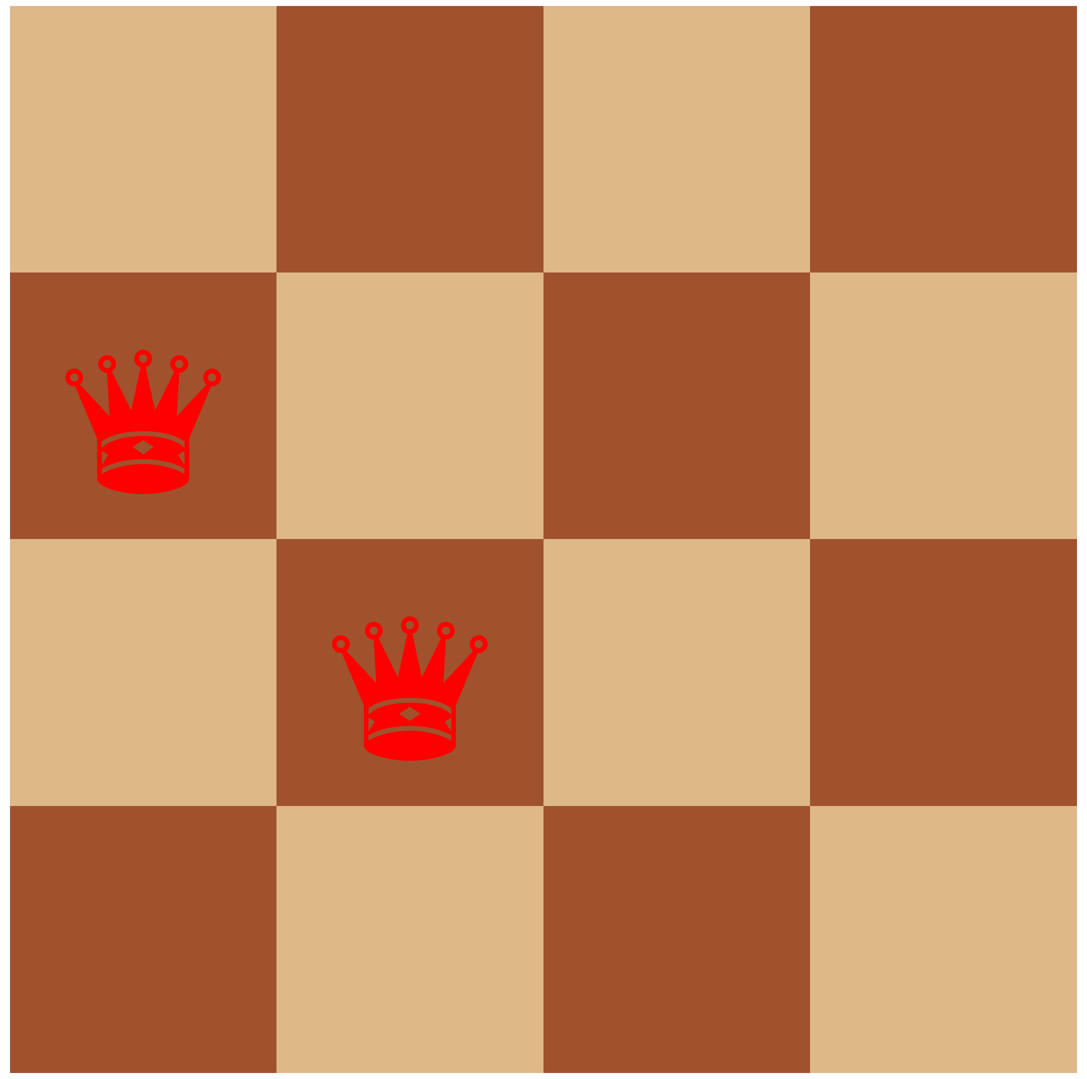
Step 19

Example: $n = 4$



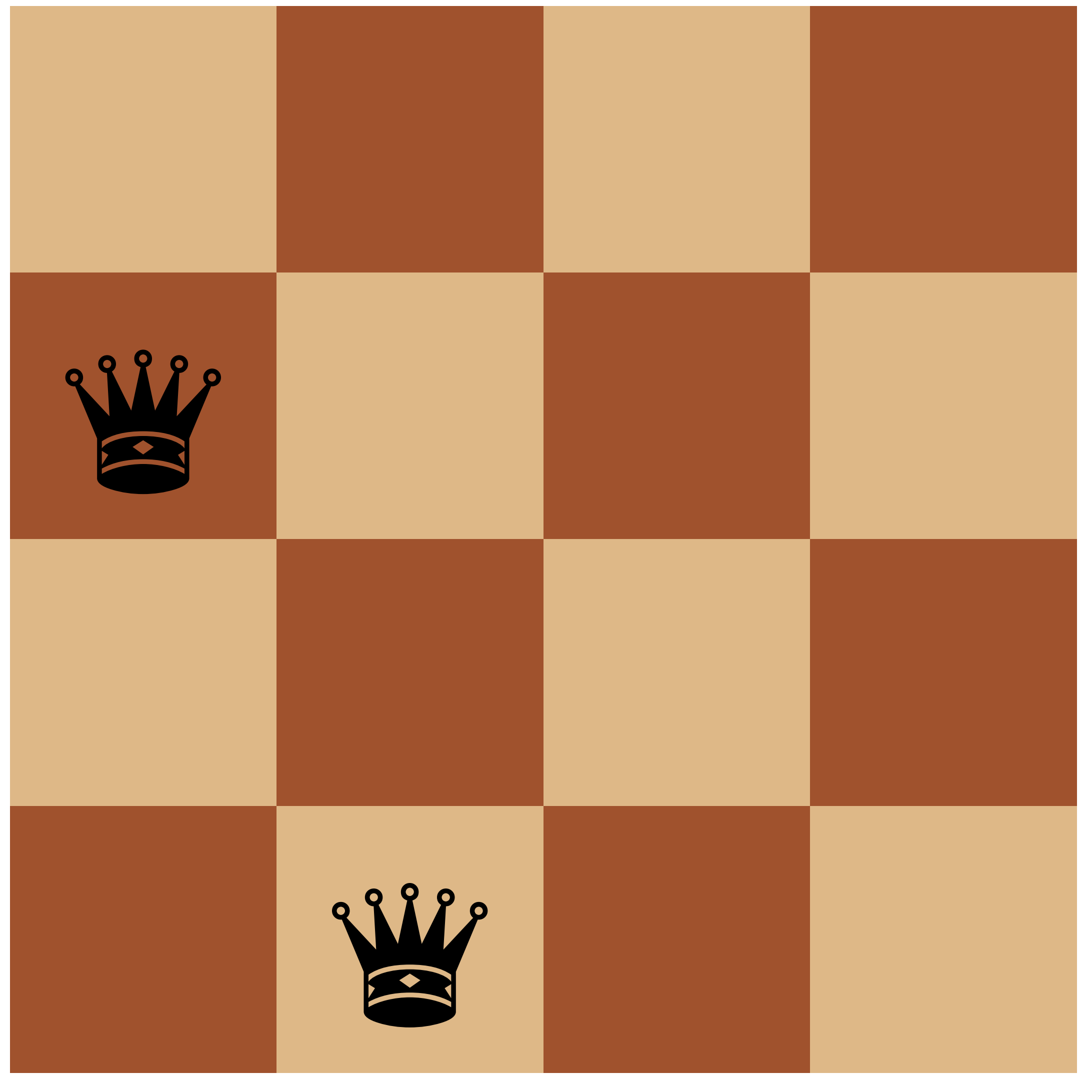
Step 20

Example: n = 4



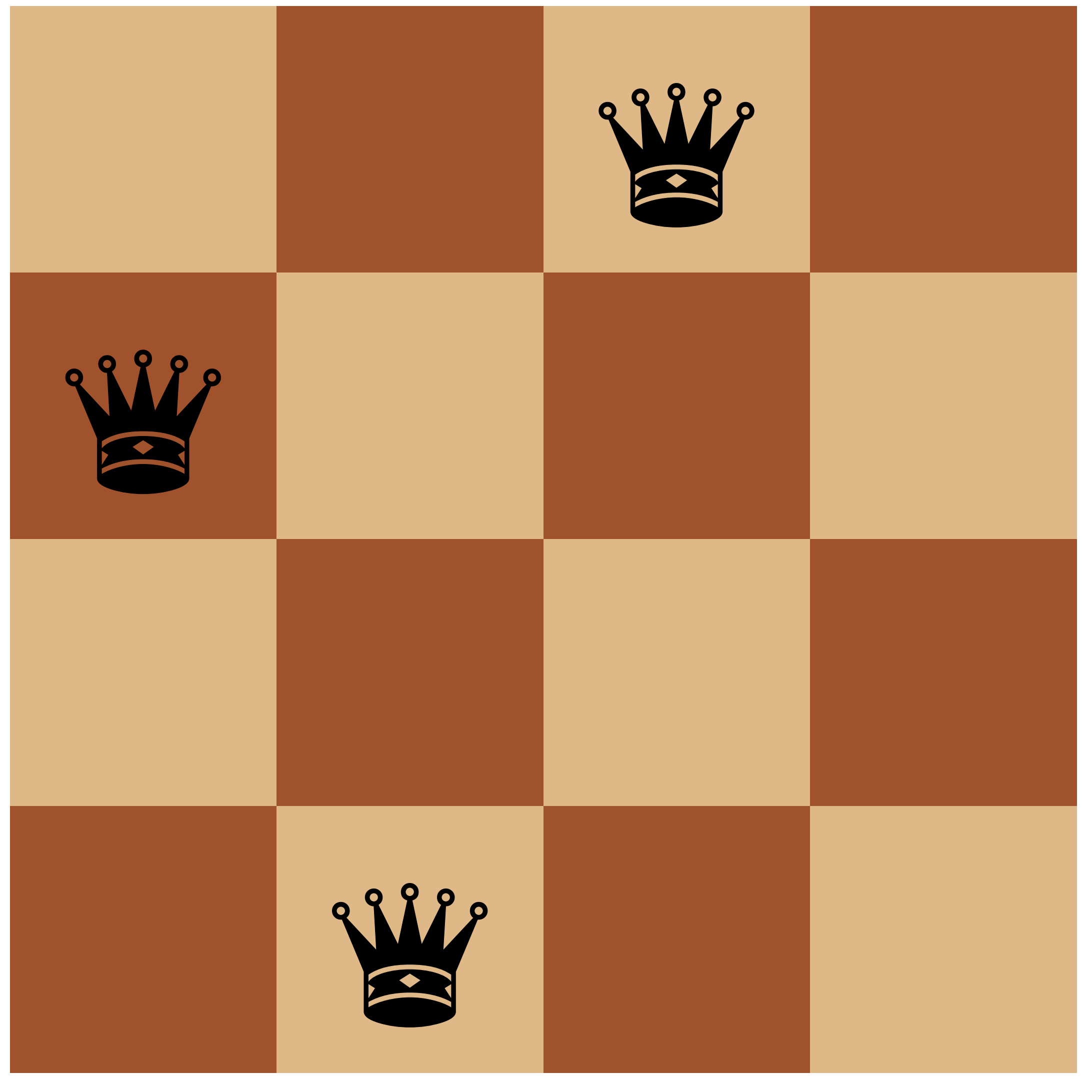
Step 21

Example: n = 4



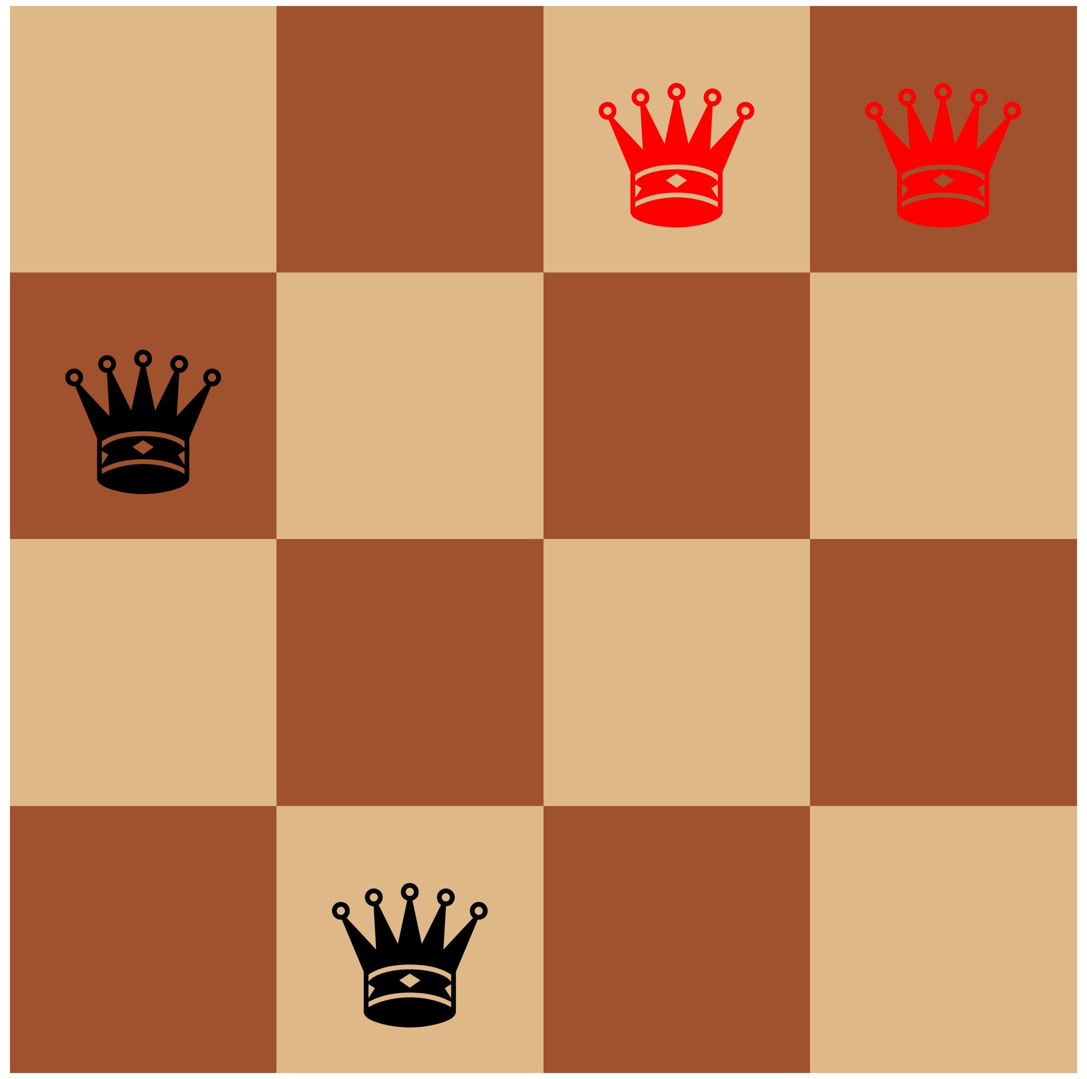
Step 22

Example: $n = 4$



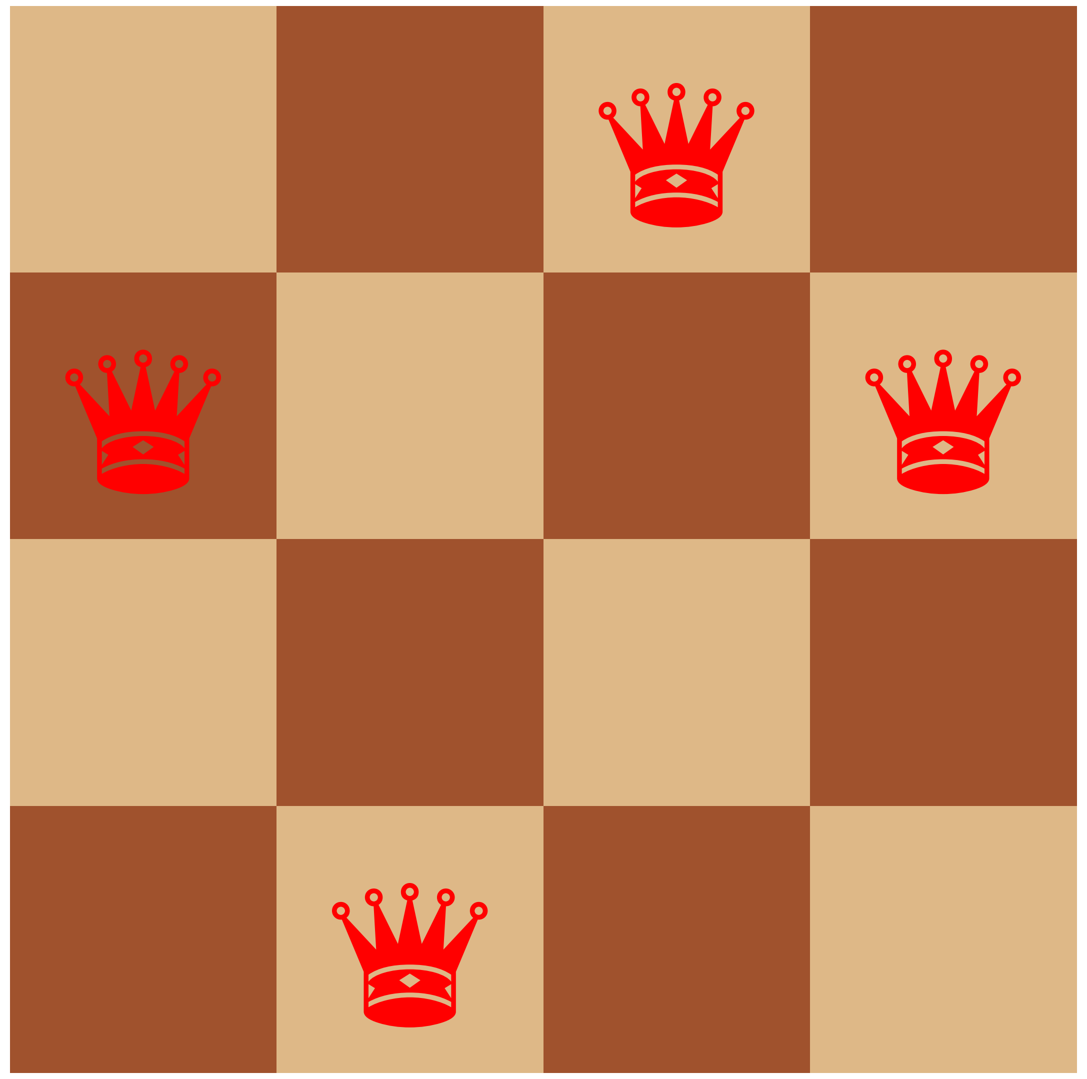
Step 23

Example: $n = 4$



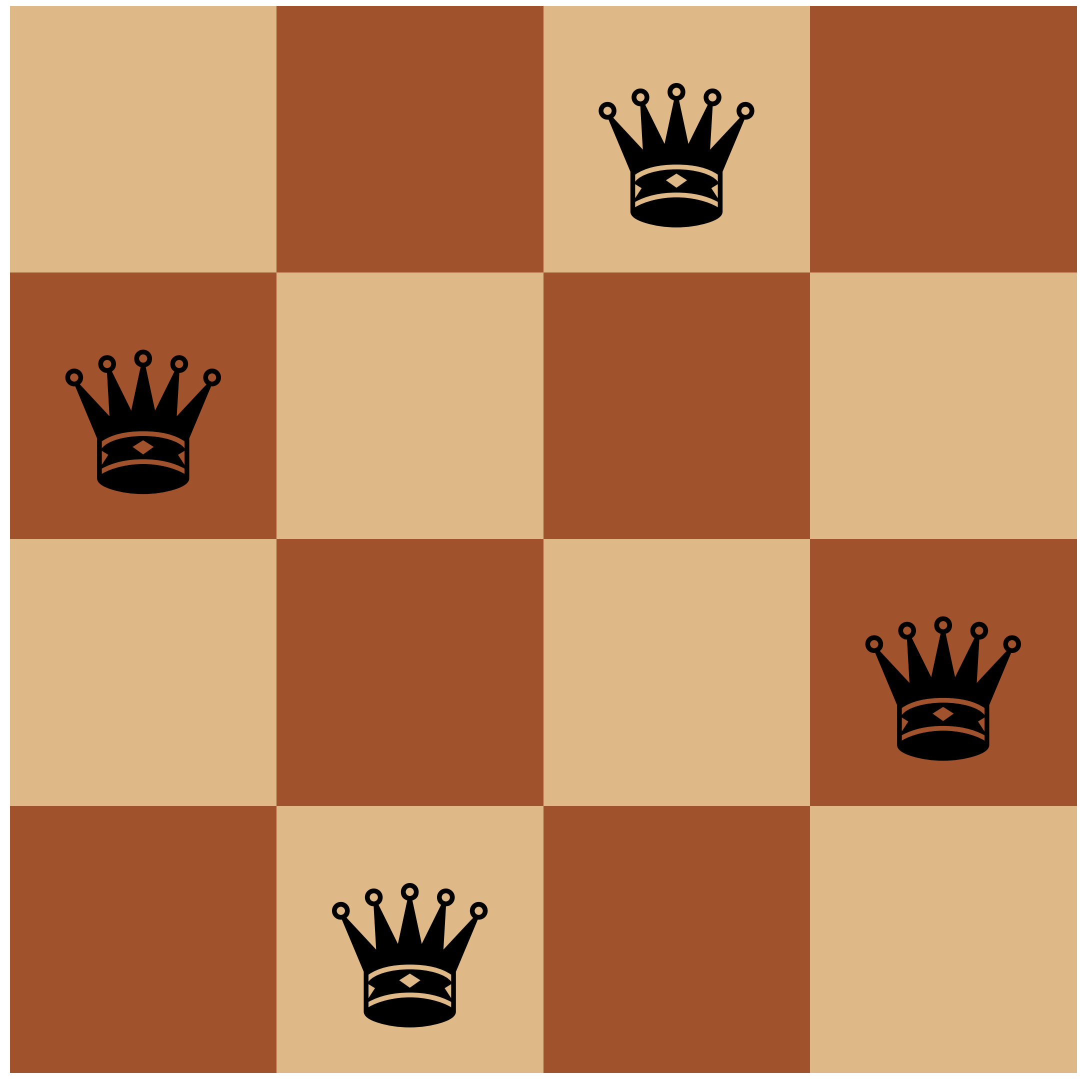
Step 24

Example: n = 4



Step 25

Example: n = 4



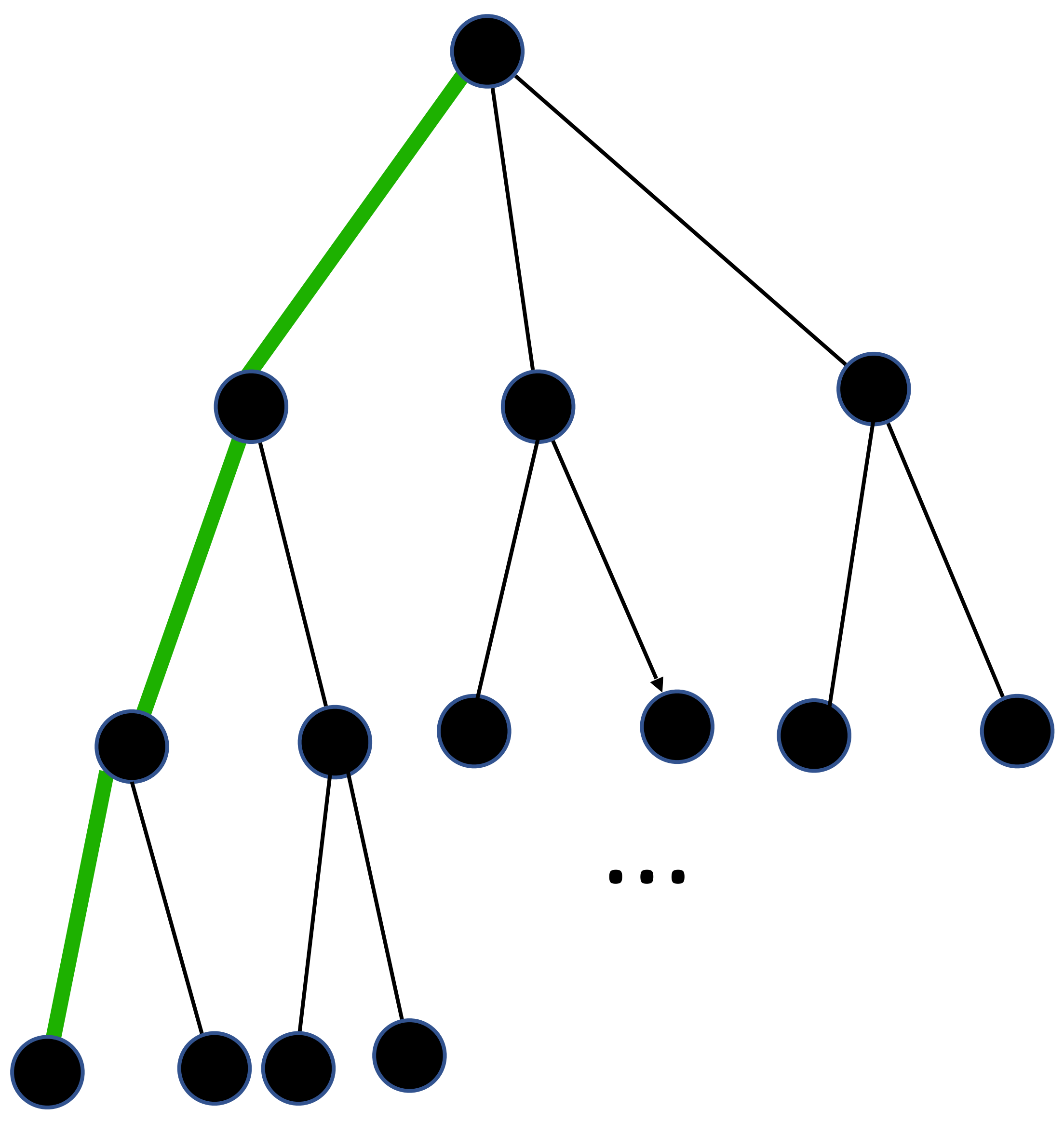
Step 26

Success!

Backtracking as search

Backtracking performs a depth-first search through the solution space

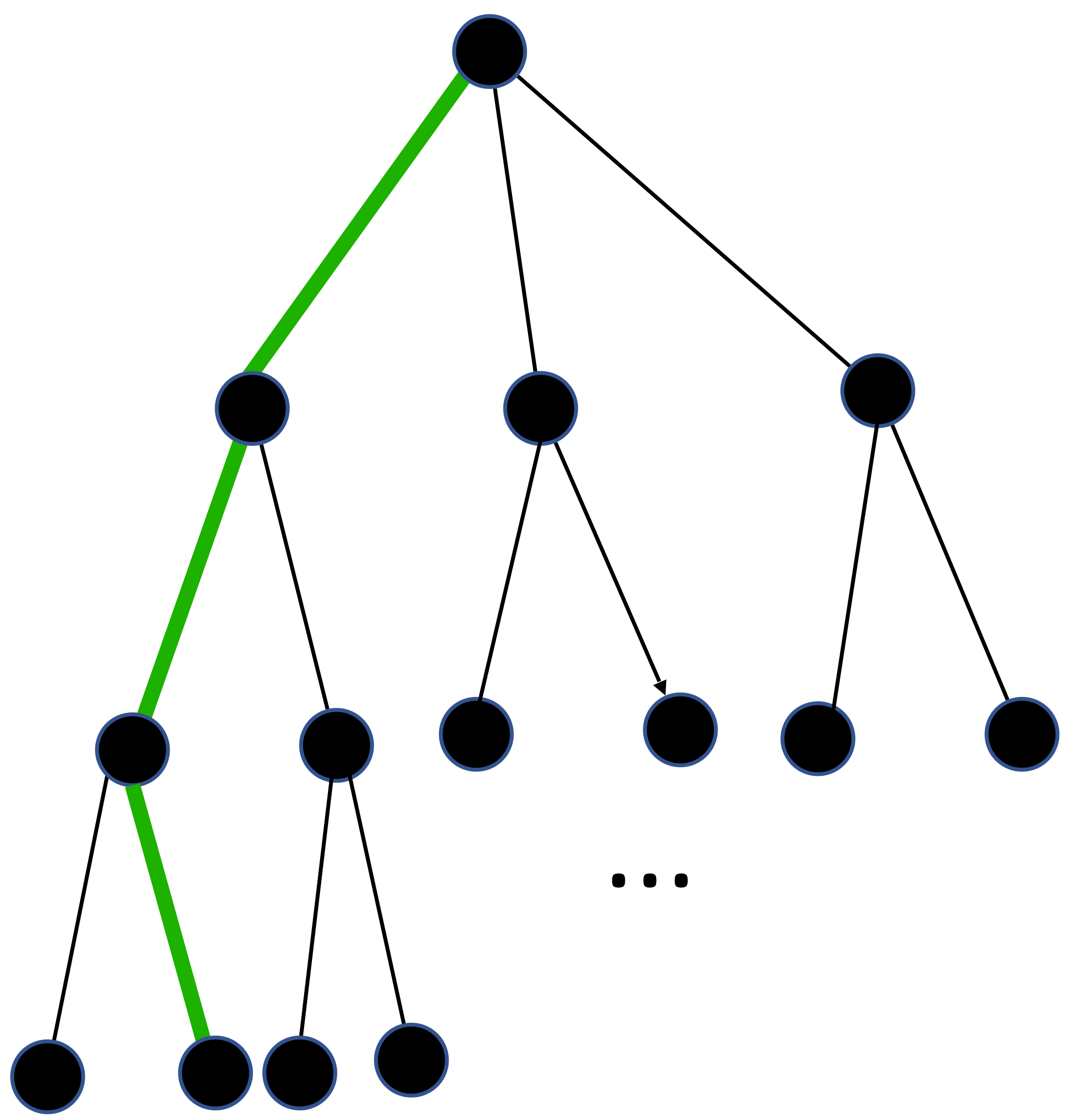
- It tries the first possible value for the first step
- Then the first value for the second step
- And so on



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

Backtracking as search

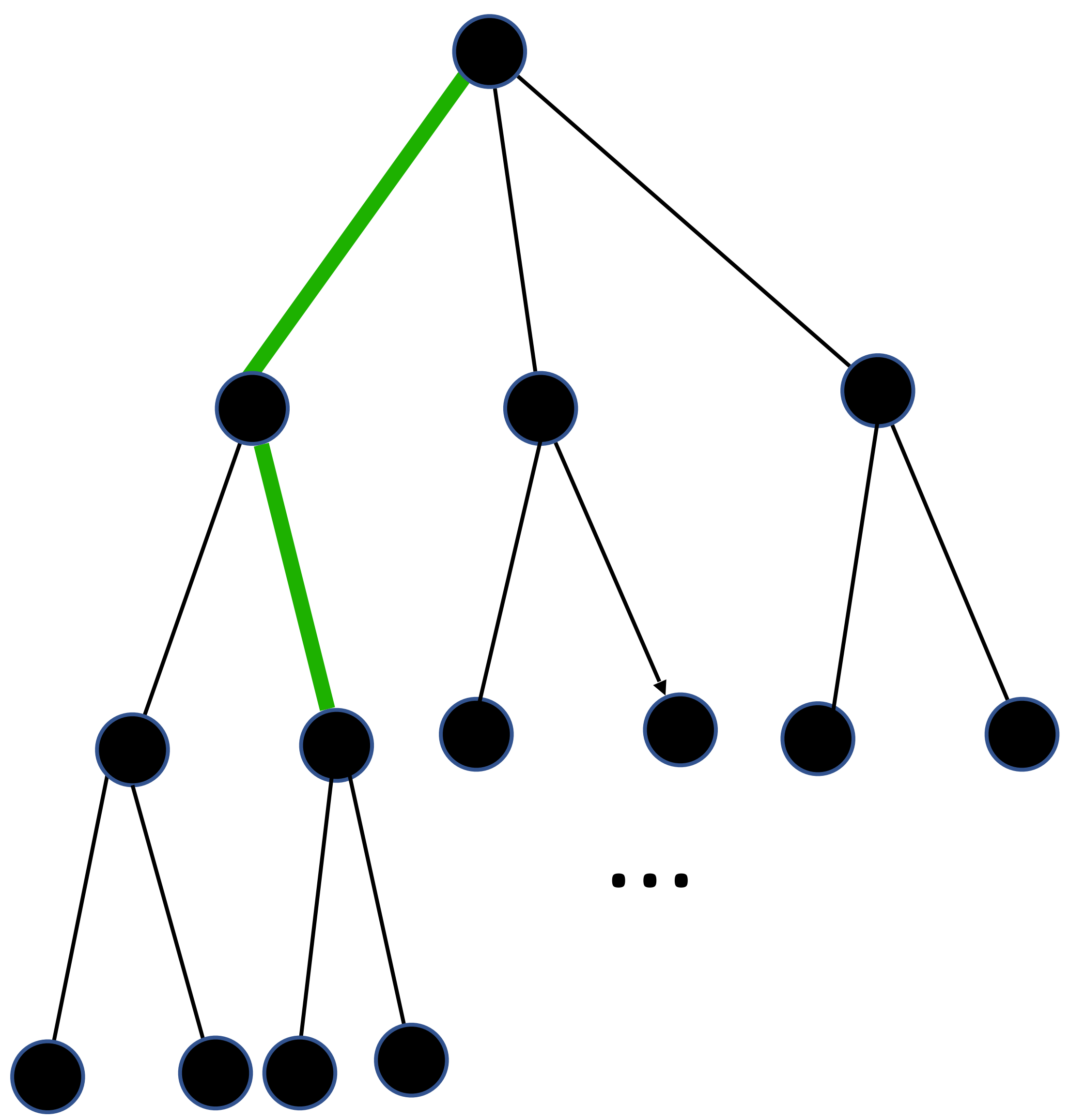
If it's not a valid solution, we back up and make a different choice



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 valid

- 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

The 4-queens example did exactly this

- As soon as a partial solution contained two queens in the same row or on the same diagonal, it moved on

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⟩)
```

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⟩)
```

Backtracking examples

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

(Not the most efficient way)

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

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

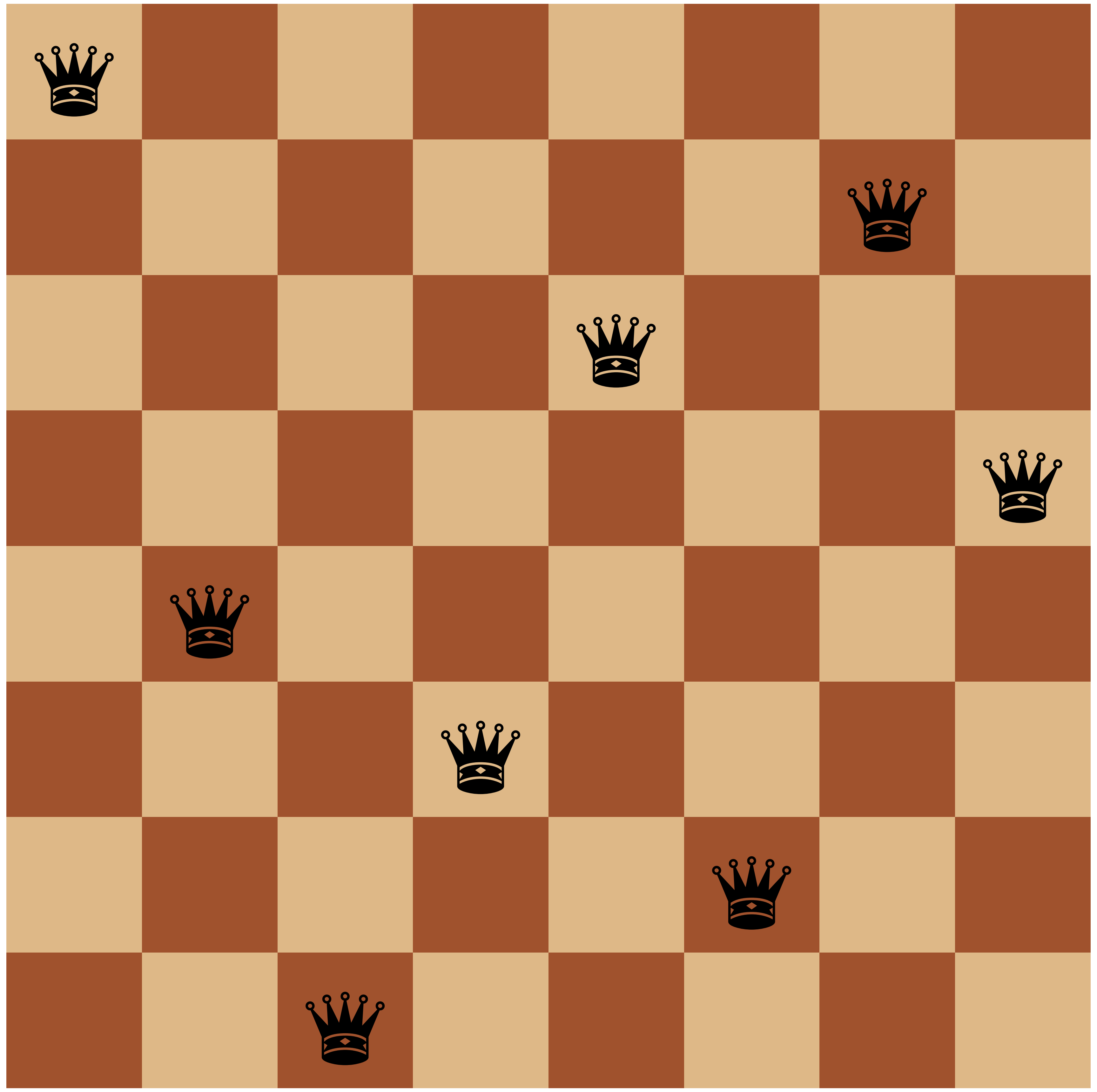
We just need to deal with the problem-specific parts

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

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 column
  - 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 column
  - 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 column
  - 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))
```

Find all solutions

No harder, we just need to plug in the usual parts

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

Data types

What do we need to implement a data type?

Three kinds of procedures

- Recognizers: Is this thing an object of type X?
- Constructors: Create an object of type X
- Accessors: Get field Y from an object of type X

Since we're working functionally, we don't need to *set* any fields, we would just create a new object with the appropriate field

Recognizers in the interpreter project

The interpreter you'll write will parse s-expressions into trees and then evaluate the trees

Each expression will have a different type with different fields

- if-then-else: a condition, a then-branch, and an else-branch
- let: list of bindings and a body
- procedure application: a list of expressions, the first being the procedure
- etc

You'll need to to use be able to tell what type of expression you're working with

```
(cond [(ite-exp? exp) …]
      [(let-exp? exp) …]
      …)
```

Before we continue…

```
(struct name field-a field-b…)
```

Racket has a very general mechanism for creating structures and the associated procedures

We're not going to use it in this course because it's worth learning how we can do this all ourselves

If you were writing production Racket code, you would definitely want to use this rather than doing it yourself!

Data types in general

We're going to construct data types out of lists (of course)

The first element in the list is going to be a symbol that's the name of the data type

Depending on the specific data type, the other elements in the list will either be the elements that comprise an instance or fields of the data type

Example: set

Let's represent a set as a list of unique elements with `'set` `cons`ed onto the front

- Empty set: (list 'set); equivalently '(set)
- Singleton set: (list 'set 'x); equivalently '(set x)

Example: set

Procedures we'll want

- Constructor:

```
(define (set elements)
  (cons 'set (remove-duplicates elements))
```

- Recognizer

```
(define (set? obj)
  (and (pair? obj) (eq? (first obj) 'set)))
```

- Accessor

```
(define (set-members s)
  (if (set? s)
      (rest s)
      (error 'set-members "s is not a set")))
```

Additional procedures

```
(define (set-contains? x s)
  (member? x (set-members s)))

(define (set-insert x s)
  (if (set-contains? x s)
    s
    (cons 'set (cons x (set-members s)))))
; We could have used (set (cons x (set-members s))) too

(define (set-union s1 s2)
  (foldl set-insert s1 (set-members s2)))

; And so on. Note that these aren't super efficient
```

Data type with fields

Let's create a type to represent a course

```
(define (course dept num name)
  (list 'course dept num name))

(define (course? c)
  (and (pair? c) (= (first c) 'course)))

(define (course-dept c)
  (if (course? c)
      (second c)
      (error 'course-dept "c is not a course")))
```

```
(define (course-num c)
  (if (course? c)
      (third c)
      (error 'course-num "c is not a course")))

(define (course-name c)
  (if (course? c)
      (fourth c)
      (error 'course-name "c is not a course")))
```

Interpreter project

Project overview

In the next few homeworks, you'll write a small Scheme interpreter

The project has two primary functions

- `(parse exp)` creates a tree structure that represents the expression `exp`
- `(eval-exp tree environment)` evaluates the given expression `tree` within the given `environment` and returns its value

Environments

Environments are used repeatedly in eval-exp to look up the value bound to a symbol

There are two functionalities we need with environments

The first is we want to look up the value bound to a symbol; e.g.,

```
(let ([x 3])
  (let ([x 4])
    (+ x 5)))
```

should return 9 since the innermost binding of x is 4

Environments

Second, we need to produce new environments by extending existing ones

```
(let ([x 3])
  (+ (let ([x 10])
       (* 2 x))
     x))
```

evaluates to 23

- If `E0` is the top-level environment, then the first let extends `E0` with a binding of `x` to 3
- If `E1` is the new environment, we write `E1 = E0[x ↦ 3]`
- The second let creates a new environment `E2 = E1[x ↦ 10]`
- The `(* 2 x)` is evaluated using `E2`
- The final `x` is evaluated using `E1`

Extending environments

There are only two places where an environment is extended

Extending environments

Procedure call

The first is a procedure call
(exp0 exp1 … expn)

exp0 should evaluate to a closure with three parts

- the procedure's parameter list;
- it's body; and
- the environment in which it was created

The other expressions are the arguments

The closure's environment needs to be extended with the parameters bound to the arguments

Extending environments

Procedure call

For example imagine the parameter list was `'(x y z)` and the arguments evaluated to `2`, `8`, and `'(1 2)`

If `E` is the closure's environment, then the closure's body should be evaluated with the environment

```
E[x ↦ 2, y ↦ 8, z ↦ '(1 2)]
```

Extending environments

Let expressions

The other situation where we extend an environment is a let expression

Consider

```
(let ([x (+ 3 4)]
      [y 5]
      [z (foo 8)])
  body)
```

The binding list is, of course, just `(second let-exp)` where `let-exp` is the whole let expression

The symbols can are `(map first binding-list)`

The binding expressions are `(map second binding-list)`

Extending environments

Let expressions

Evaluating the expressions is easy

```
(map (λ (exp)
       (eval-exp exp current-env))
     (map second binding-list))
```

Now we bind the parameters to these values

Extending environments

In both cases

- We have a list of symbols
- We have a list of values

This suggests a data type (that looks slightly different from what we've seen before)

An empty environment can be represented with `null`

An extended environment can be represented with
`(list 'env symbols values old-env)`

Looking up a binding

Looking up `x` in an environment has two cases

If the environment is empty, then we know `x` isn't bound there so it's an error

Otherwise we look in the list of symbols of an extended environment

- If the symbol `x` appears in the list, then great, we have the value
- If the symbol `x` doesn't appear, then we lookup `x` in the `old-env`