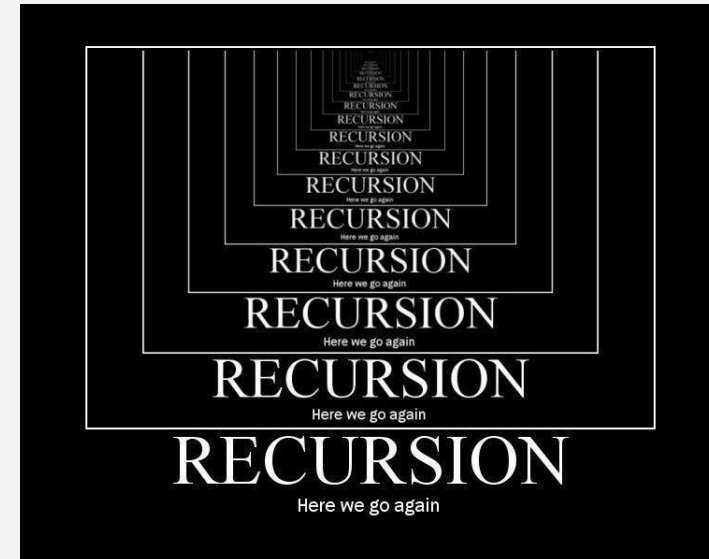


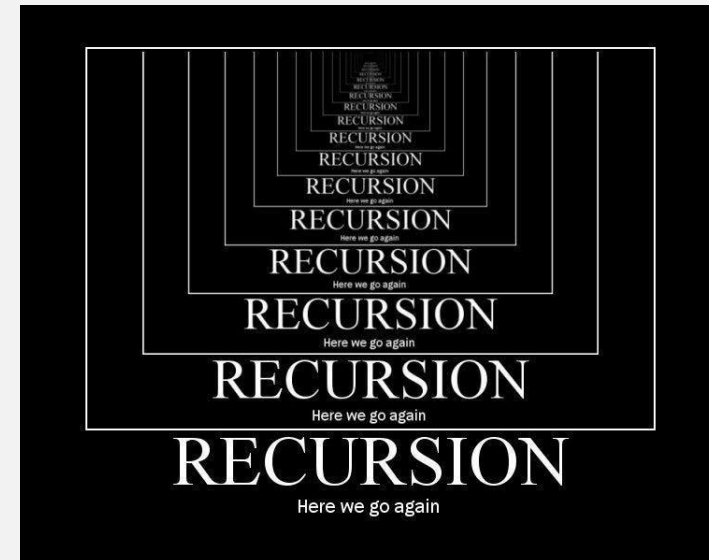
UMass Boston Computer Science
CS450 High Level Languages
**Generative Recursion,
Backtracking**
Tuesday, May 6, 2025



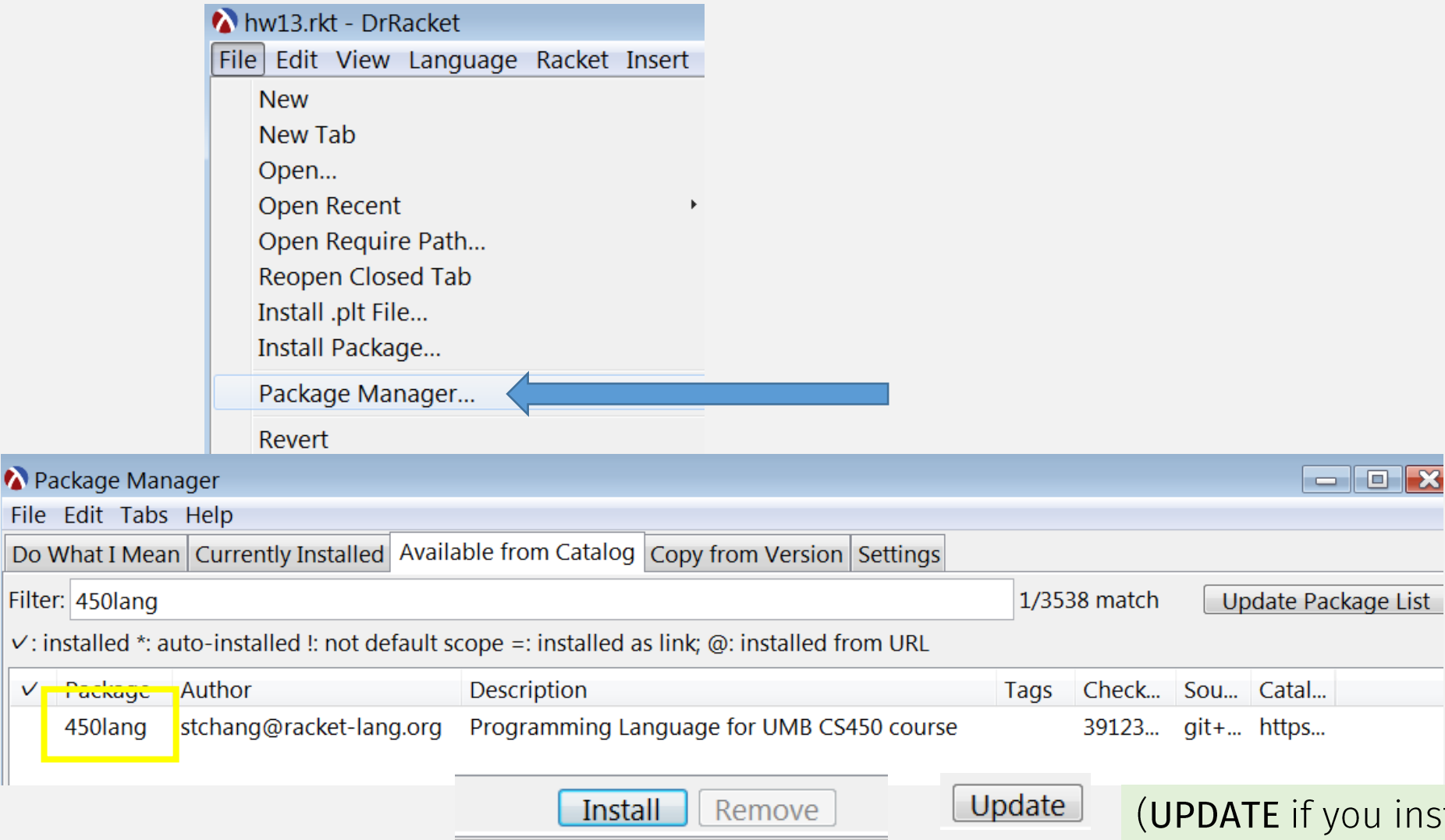
Logistics

- HW 12 in
 - ~~Due: Tues 5/6 11am EST~~
- HW 13 out
 - Due: Tues 5/13 11am EST
 - Last hw!
 - Must use #lang 450lang

(improper base case!)



Installing “450 Lang”



The image shows the DrRacket interface. The top window, titled "hw13.rkt - DrRacket", has a menu bar with "File", "Edit", "View", "Language", "Racket", and "Insert". The "File" menu is open, showing options like "New", "New Tab", "Open...", "Open Recent", "Open Require Path...", "Reopen Closed Tab", "Install .plt File...", "Install Package...", "Package Manager...", and "Revert". A blue arrow points to the "Package Manager..." option.

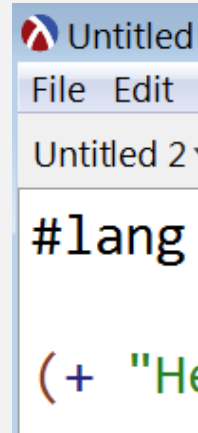
Below the main window is the "Package Manager" window. It has a menu bar with "File", "Edit", "Tabs", and "Help". Below the menu bar are tabs: "Do What I Mean", "Currently Installed", "Available from Catalog", "Copy from Version", and "Settings". The "Available from Catalog" tab is selected. The filter is set to "450lang", showing "1/3538 match". There is a button "Update Package List". Below the filter is a legend: "✓: installed *: auto-installed !: not default scope =: installed as link; @: installed from URL".

✓	Package	Author	Description	Tags	Check...	Sou...	Catal...
✓	450lang	stchang@racket-lang.org	Programming Language for UMB CS450 course		39123...	git+...	https...

At the bottom of the Package Manager window are three buttons: "Install", "Remove", and "Update". A green box highlights the "Update" button with the text "(UPDATE if you installed last week)".

Using “450 Lang”

Read the [Programming Language Specification](#) linked from HW description!



“quotes” are

Programming language specification

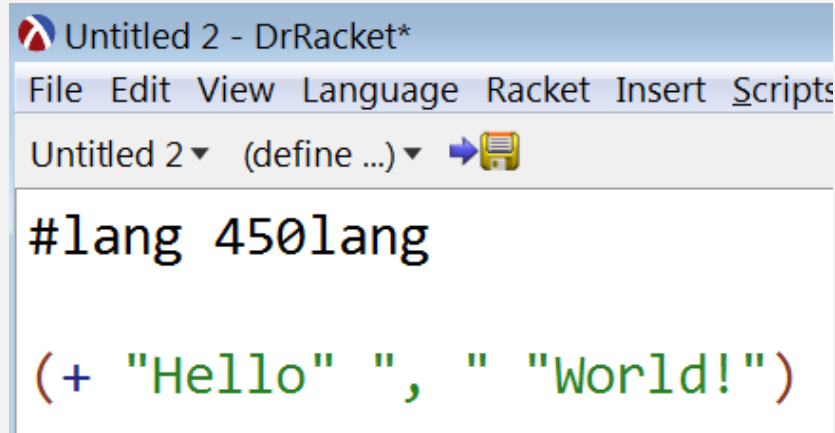
From Wikipedia, the free encyclopedia

In [computer programming](#), a **programming language specification** (or **standard** or **definition**) is a [documentation](#) artifact that defines a [programming language](#) so that [users](#) and [implementors](#) can agree on what programs in that language mean.

Specifications are typically detailed and formal, and primarily used by implementors, with users referring to them in case of ambiguity; the [C++](#) specification is frequently cited by users, for instance, due to the complexity. Related documentation includes a [programming language reference](#), which is intended expressly for users, and a programming language rationale, which explains why the specification is written as it is; these are typically more informal than a specification.

A specification is more formal than user reference documentation!

Using “450 Lang”



```
Untitled 2 - DrRacket*
File Edit View Language Racket Insert Scripts
Untitled 2 ▾ (define ...) ▾ ➡ 📁
#lang 450lang

(+ "Hello" ", " "World!")
```

“quotes” are implicitly inserted by the language

Taking requests ...

Ask for additional primitives in **INIT-ENV**

Read the **Programming Language Specification** linked from HW description!

Added features:

- Lists
- More arith fns: `-`, `abs`
- Logical operations: `¬`, `∧`, `∨`
- “top-level” `bind/rec` Like `define`
- `rackunit` equivalents

Not as “good” as Racket

Design Recipe even more important now

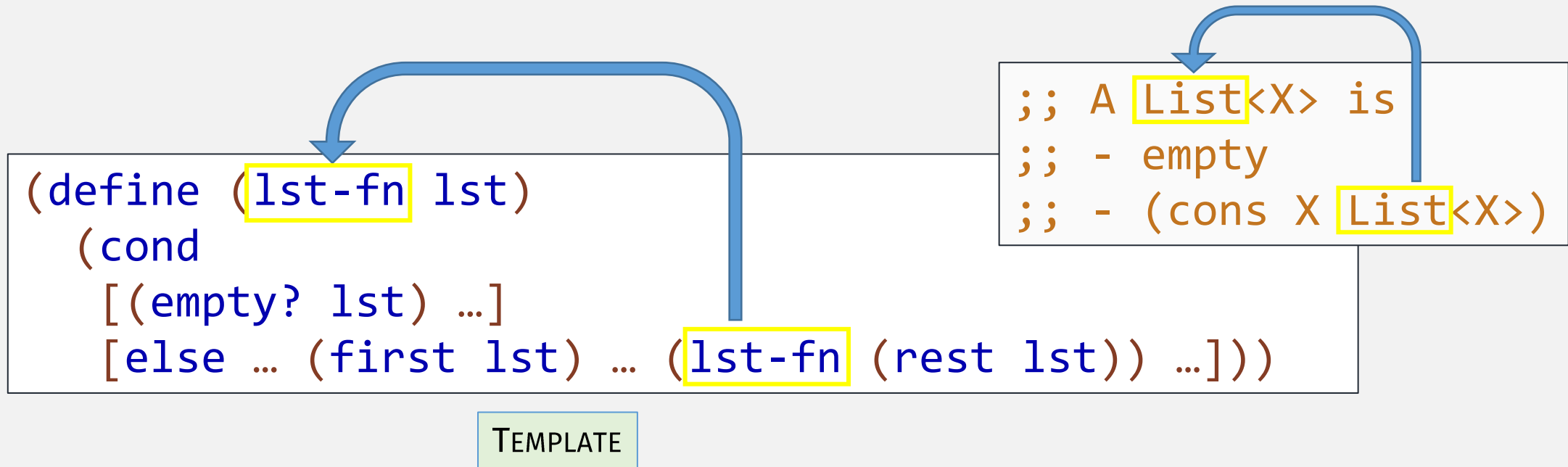
**DO NOT “save”
writing tests until
the end!!**

(you’ve been warned)

Previously

Recursion review

- Most **recursion** is **structural** (i.e., comes from **data definitions**)!



A Different Kind of Recursion!

- Not all recursion is structural (i.e., comes from data definitions)!

A Different Kind of Recursion!

- Not all recursion is structural (i.e., comes from data definitions)!

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid's algorithm
;; termination argument:
;; m is halved (at least) even via modulo fn)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))))
```

What template is this following??

A Different Kind of Recursion!

- Non-structural recursion (i.e., doesn't come from data definitions) is called **generative recursion**
- no template? ... requires **Termination Argument**
 - Explains why the function terminates – because recursive call is “smaller”!

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid's algorithm
;; termination argument:
;; m is halved (at least) every iteration (via modulo)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))))
```

But how to develop an algorithm like this??

Recursive call must be on “smaller” version of the problem

Generative (non-structural) Recursion Design Recipe

1. Name, Signature
2. Description
 - Must include **Termination Argument**
3. Examples
 - Even more important now!
4. **Code** (No structural template, but can use a “general” template)
5. Tests

Generative (non-structural) Recursion Design Recipe

1. Name, Signature
2. Description
 - Must include **Termination Argument**
3. Examples
 - Even more important now!
4. **Code** (No structural template, but can use a “general” template)
 - a) Break problems into smaller problems to (recursively) solve
 - b) Determine how to combine smaller solutions
 - c) “trivially solvable” problem is base case!
5. Tests


Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a “general” template)
 - a) Break problems into smaller problems to (recursively) solve
 - b) Determine how to combine smaller solutions
 - c) “trivially solvable” problem is base case!

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a “general” template)
- a) Break problems into **smaller** problems to **(recursively)** solve
 - b) Determine how to combine smaller solutions
 - c) “trivially solvable” problem is base case!

```
;; genrec-algo: ??? -> ???  
;; termination argument: recursive calls are “smaller” bc ...  
(define (genrec-algo problem)  
  (cond  
    [(trivial? problem) (solve-easy problem)] ;; base case  
    [else (combine-solutions  
              (genrec-algo (create-smaller-1 problem))  
              ...  
              (genrec-algo (create-smaller-n problem)))])])
```



Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a “general” template)
- a) Break problems into smaller problems to (recursively) solve
 - b) Determine how to **combine** smaller solutions
 - c) “trivially solvable” problem is base case!

```
;; genrec-algo: ??? -> ???  
;; termination argument: recursive calls are “smaller” bc ...  
(define (genrec-algo problem)  
  (cond  
    [(trivial? problem) (solve-easy problem)] ;; base case  
    [else (combine-solutions  
              (genrec-algo (create-smaller-1 problem))  
              ...  
              (genrec-algo (create-smaller-n problem))))]))
```

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a “general” template)
- a) Break problems into smaller problems to (recursively) solve
 - b) Determine how to combine smaller solutions
 - c) “trivially solvable” problem is base case!

```
;; genrec-algo: ??? -> ???  
;; termination argument: recursive calls are “smaller” bc ...  
(define (genrec-algo problem)  
  (cond  
    [(trivial? problem) (solve-easy problem)] ;; base case  
    [else (combine-solutions  
              (genrec-algo (create-smaller-1 problem))  
              ...  
              (genrec-algo (create-smaller-n problem))))]))
```

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a “general” template)
- a) Break problems into smaller problems to (recursively) solve
 - b) Determine how to combine smaller solutions
 - c) “trivially solvable” problem is base case!

```
;; genrec-algo: ??? -> ???  
;; termination argument: recursive calls are “smaller” bc ...  
(define (genrec-algo problem)  
  (cond  
    [(trivial? problem) (solve-easy problem)] ;; base case  
    [else (combine-solutions  
              (genrec-algo (create-smaller-1 problem))  
              ...  
              (genrec-algo (create-smaller-n problem))))]))
```


GenRec Template Generalizes Structural!

```
(define (lst-fn lst)
  (cond
    [(empty? lst) ...]
    [else ... (first lst) ... (lst-fn (rest lst)) ...]))
```

- Trivial solution = data def base case
- Recursive smaller problem = data def smaller piece
- Left to figure out “Combining” pieces

```
;; genrec-algo: ??? -> ???

(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
            (genrec-algo (create-smaller-1 problem))
            ...
            (genrec-algo (create-smaller-n problem)))]))
```

Previously

Generative Recursion Example!

(Functional) Quicksort

```
;; smaller-than: ListofInt Int -> ListofInt  
;; Returns a list containing elements of given list  
;; that are less than the given int
```

```
(check-equal?  
  (smaller-than (list 1 3 4 5 9) 4)  
  (list 1 3))
```

```
;; larger-than: ListofInt Int -> ListofInt  
;; Returns a list containing elements of given list  
;; that are greater than the given int
```

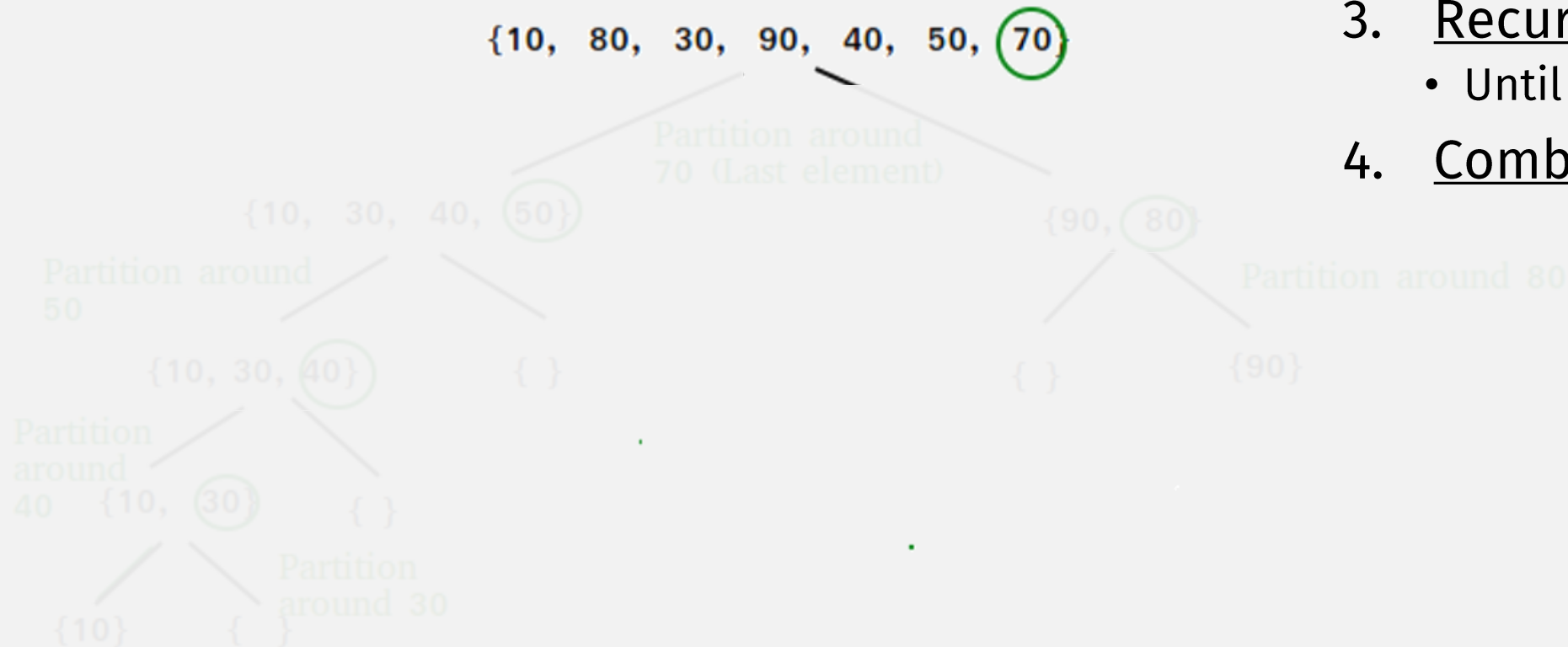
```
(check-equal?  
  (greater-than (list 1 3 4 5 9) 4)  
  (list 5 9))
```

```
;; qsort: ListofInt -> ListofInt  
;; sorts the given list of ints in ascending order
```

```
(define (qsort lst)  
  (define pivot (random lst))  
  (append (qsort (smaller-than lst pivot))  
          (list pivot)  
          (qsort (greater-than lst pivot))))
```

Quicksort overview (“divide and conquer”)

1. Choose “pivot” element
2. Partition into smaller lists:
 - $<$ pivot
 - \geq pivot
3. Recurse on smaller lists
 - Until base case
4. Combine small solutions



Gen Rec Example: (functional) quicksort

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
    [(trivial? problem) (solve-easy lst)] ;; base case
    [else
     (define pivot (first lst))
     (combine-solutions
      (qsort (smaller-problem-1 lst))
      ...
      (qsort (smaller-problem-n lst))))]))
```

1. Choose “pivot” element
2. Partition into smaller lsts:
 - < pivot
 - >= pivot
3. Recurse until base case
4. Combine small solutions

Gen Rec Example: (functional) quicksort

1. Choose “pivot” element
2. Partition into smaller lsts:
 - < pivot
 - >= pivot
3. Recurse until base case
4. Combine small solutions

```
;; qsort: List<Int> -> List<Int>  
;; termination: Function “arithmetic”!
```

Result is a function!

```
(curry f arg1)
```

=

```
(lambda (arg2) (f arg1 arg2))
```

Curry = “partial apply”

```
(first lst))
```

```
(combine-solutions
```

```
(qsort (filter (curry > pivot) (rest lst))
```

...

```
(qsort (filter (curry <= pivot) (rest lst))
```

“less than”

“greater than”

```
(curry > pivot)  
=  
(lambda (x) (> pivot x))
```

Gen Rec Example: (functional) quicksort

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
    [(trivial? problem) (solve-easy lst)] ;; base case
    [else
     (define pivot (first lst))
     (combine-solutions
      (qsort (filter (curry > pivot) (rest lst)) "less than")
      ...
      (qsort (filter (curry <= pivot) (rest lst)) "greater than")])])
```


1. Choose "pivot" element
2. Partition into smaller lsts:
 - < pivot
 - >= pivot
3. Recurse until base case
4. Combine small solutions



Gen Rec Example: (functional) quicksort

1. Choose “pivot” element
2. Partition into smaller lsts:
 - < pivot
 - >= pivot
3. Recurse until base case
4. Combine small solutions

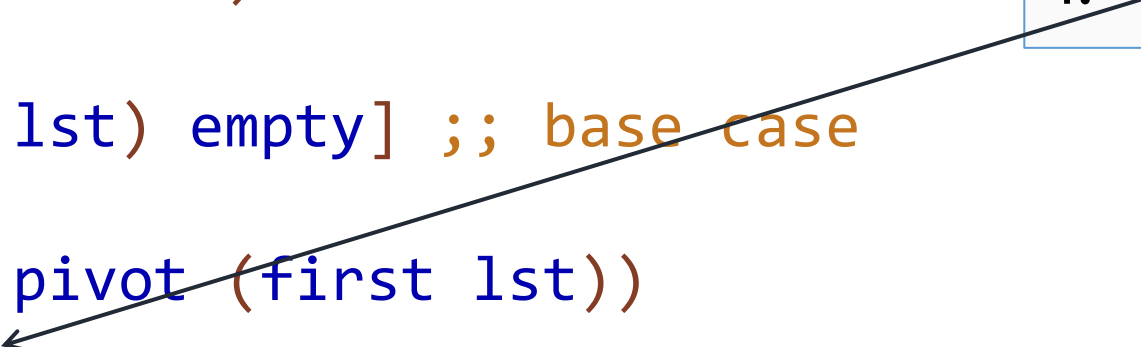
```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
    [(empty? lst) empty] ;; base case
    [else
     (define pivot (first lst))
     (combine-solutions
      (qsort (filter (curry > pivot) (rest lst)))
      ...
      (qsort (filter (curry <= pivot) (rest lst))))))]))
```



Gen Rec Example: (functional) quicksort

1. Choose “pivot” element
2. Partition into smaller lsts:
 - < pivot
 - >= pivot
3. Recurse until base case
4. Combine small solutions

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
    [(empty? lst) empty] ;; base case
    [else
     (define pivot (first lst))
     (append
      (qsort (filter (curry > pivot) (rest lst)))
      (list pivot)
      (qsort (filter (curry <= pivot) (rest lst)))))]))
```



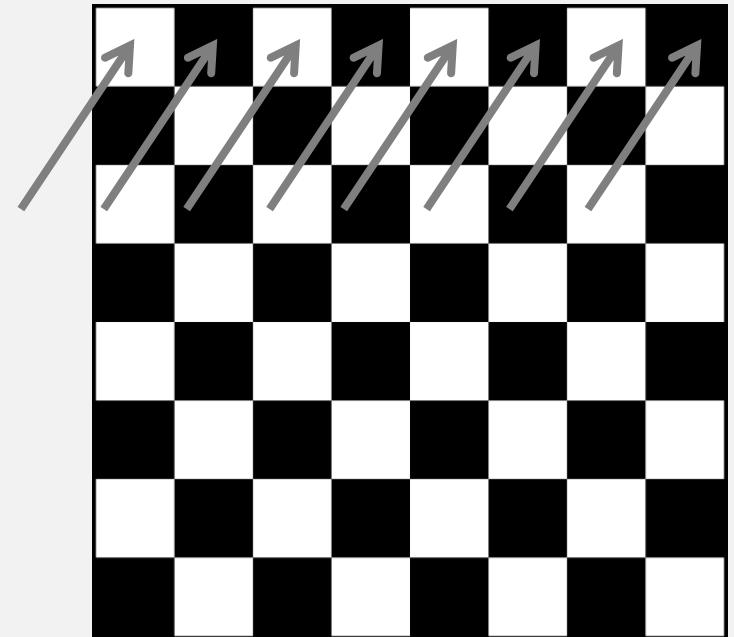
Gen Rec Example: (functional) quicksort

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls “smaller” bc at least one item dropped (pivot)
(define (qsort lst)
  (cond
    [(empty? lst) empty] ;; base case
    [else
     (define pivot (first lst))
     (append
      (qsort (filter (curry > pivot) (rest lst)))
      (list pivot)
      (qsort (filter (curry <= pivot) (rest lst)))))]))
```

Not always obvious!

```
;; termination argument:  
;; recursive calls “smaller” bc ...
```

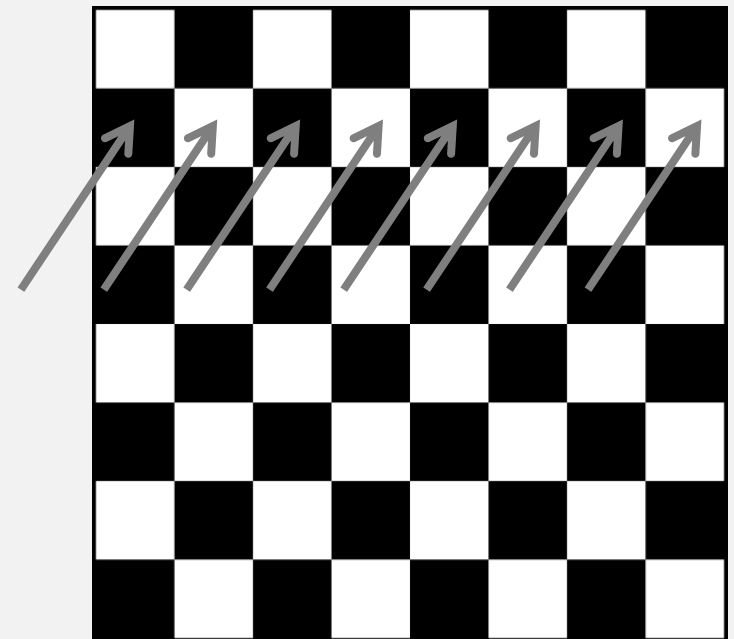
Example: traversing a game board ...

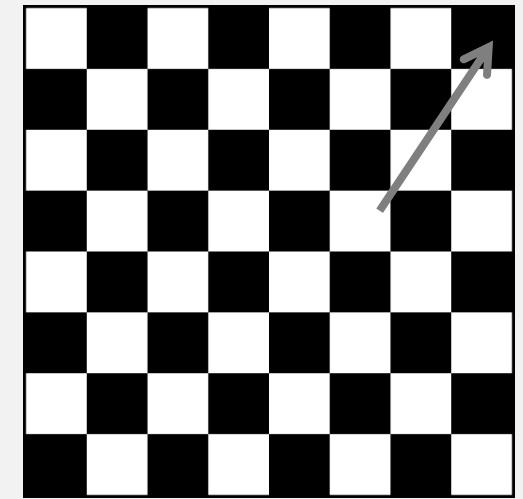


Not always obvious!

```
;; termination argument:  
;; recursive calls “smaller” bc ...
```

Example: traversing a game board ...





Not always obvious!

```
;; termination argument:
;; recursive calls "smaller" bc ... "distance" to last square gets "smaller" ???
(define (find-sol row col)
  (cond
    [(found-sol? row col ...) ... DONE ...] ;; base case
    [(at-last-col? ... col ...) (find-sol (next row) FIRST-COLUMN)]
    [(at-last-row? ... row ...) ... NO-SOLUTION ... ]
    [else
     ...
     ]))
```

What is the "smaller" problem???

Is this always true???

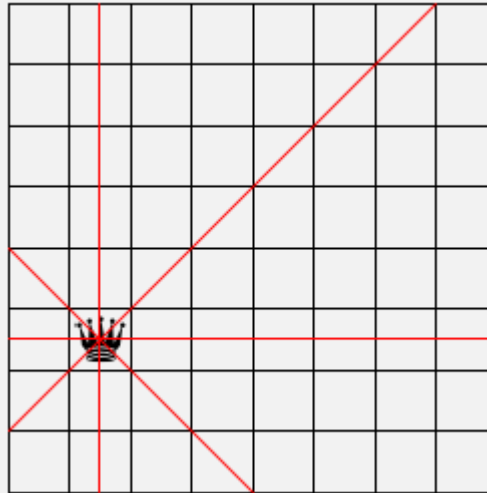
“distance” to last square gets “smaller” ???

```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol row col)
  (cond
    [(found-sol? row col ...) ... DONE ...] ;; base case
    [(at-last-col? ... col ...) (find-sol (next row) FIRST-COLUMN)]
    [(at-last-row? ... row ...) ... NO-SOLUTION ... ]
    [else
     ...
     ]))
```

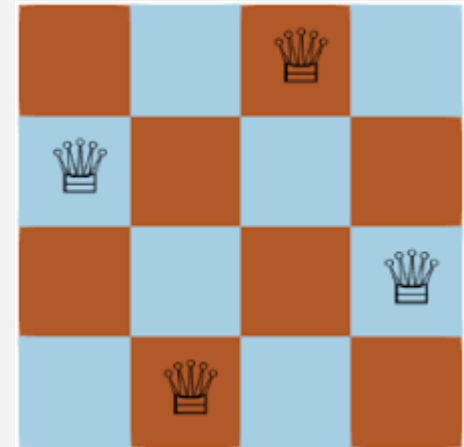
What is the “smaller” problem???

N-Queens problem

- Place n queens on an $n \times n$ chess board so that no queen “threatens” another ...



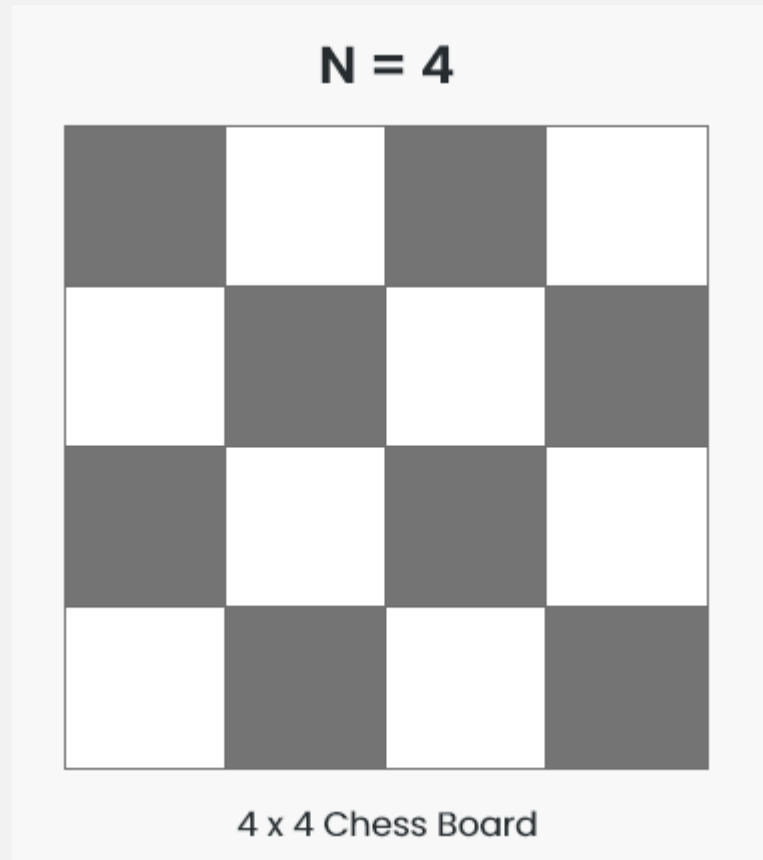
All the positions “threatened” by a queen



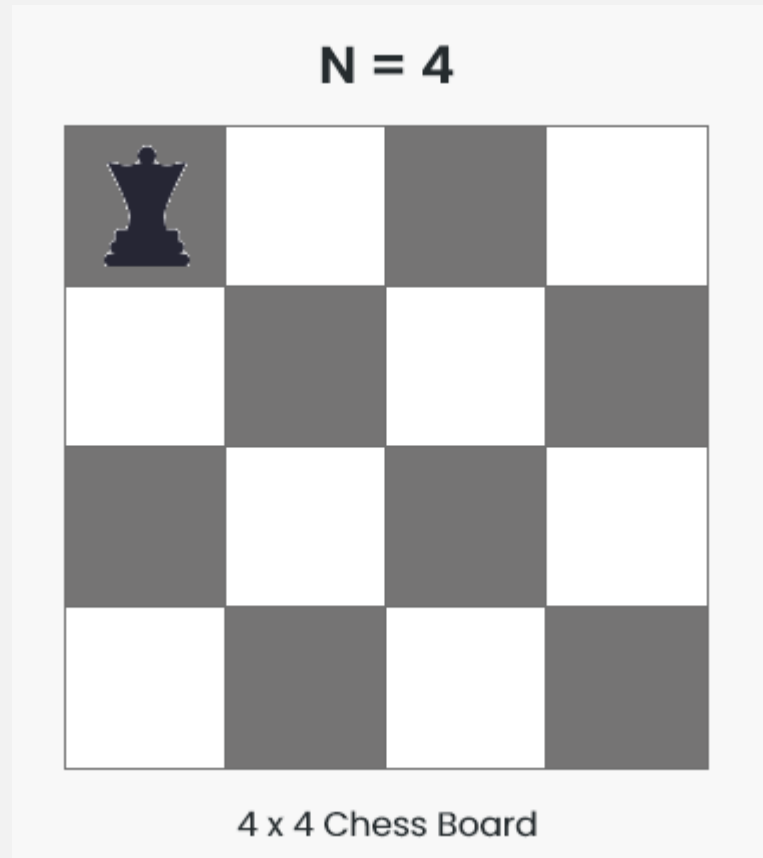
N-Queens problem – solving ...

- Place n queens on an $n \times n$ chess board so that no queen “threatens” another ...
- To find a solution ...
- ... optimistically “place” each queen in non-threatening position on board ...
- ... and hope it works out ???

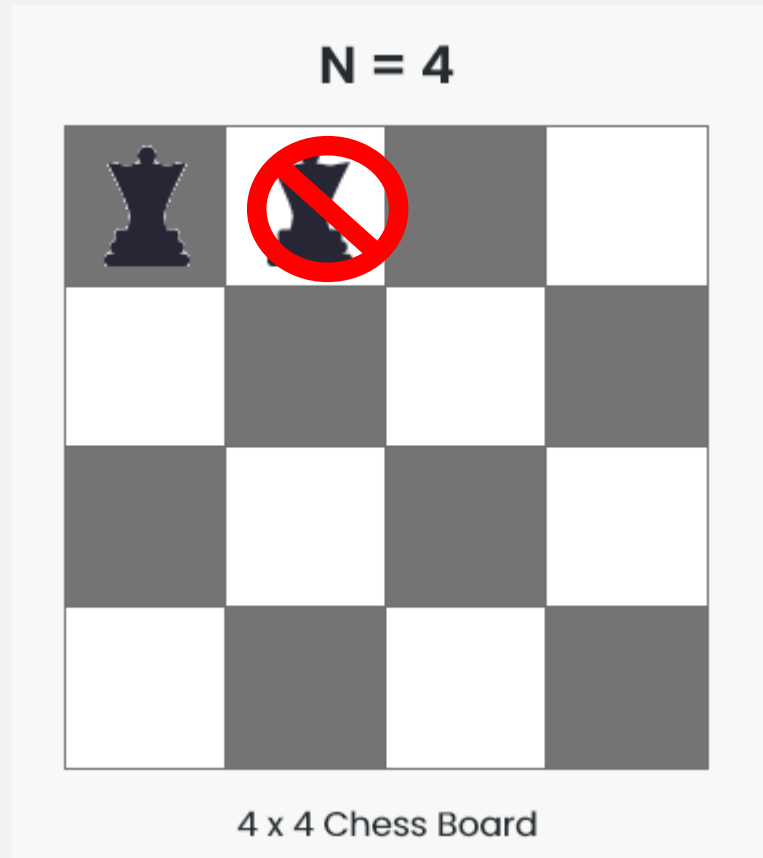
Example: 4-queens



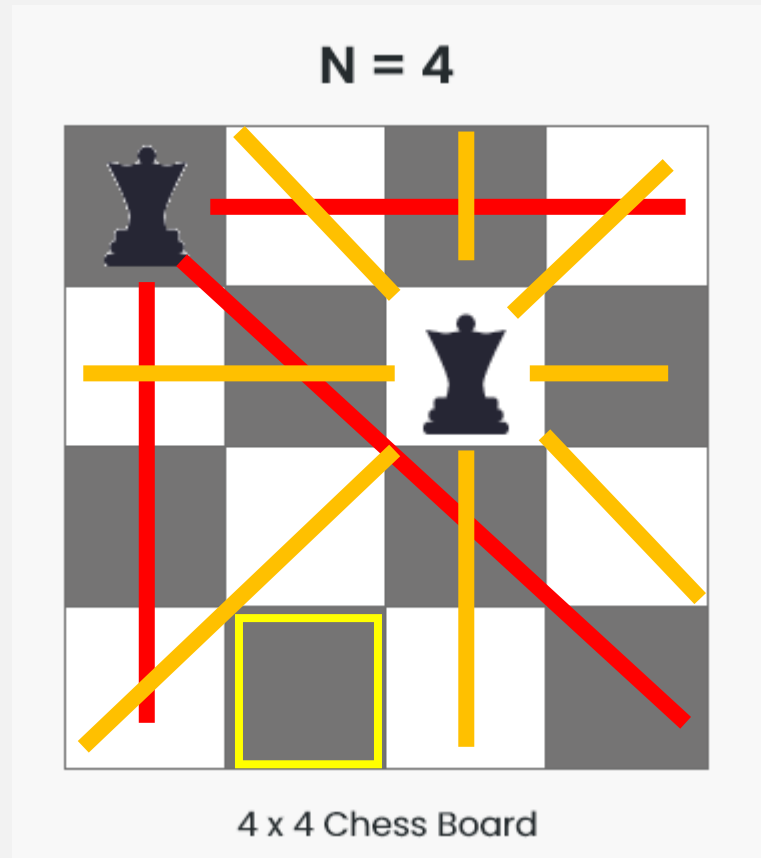
Example: 4-queens



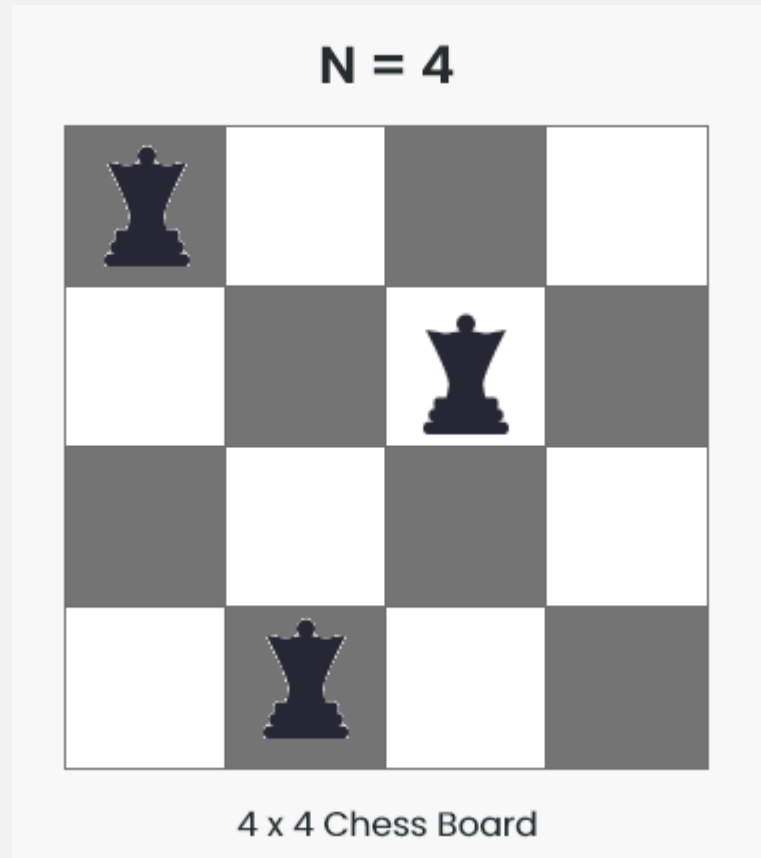
Example: 4-queens



Example: 4-queens



Example: 4-queens



But ... need to place 4 queens!

FAIL???

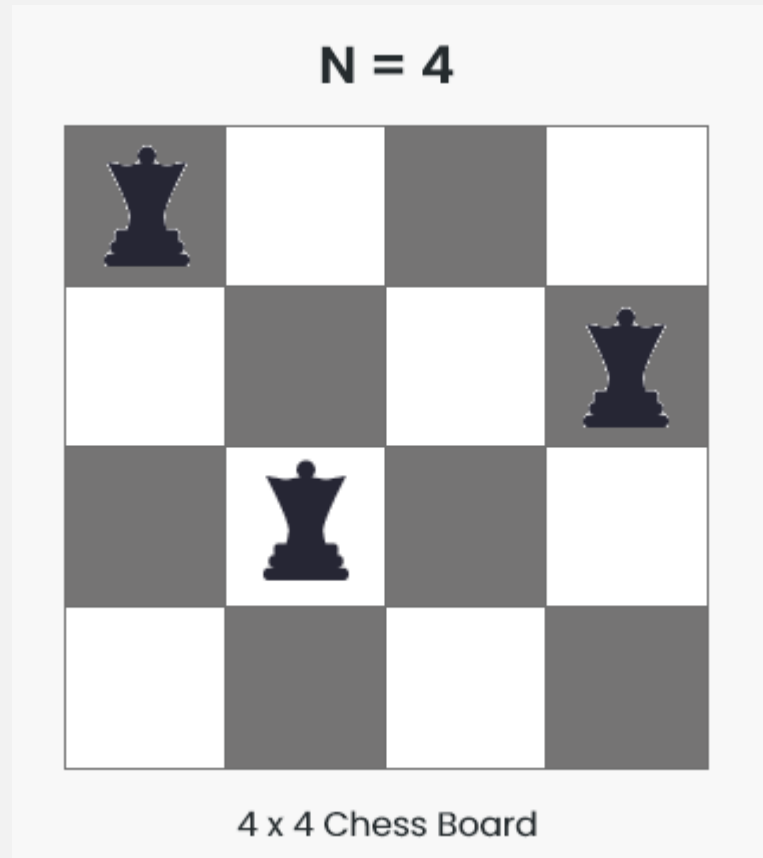
No, we havent tried all solutions ...

... need to go backwards

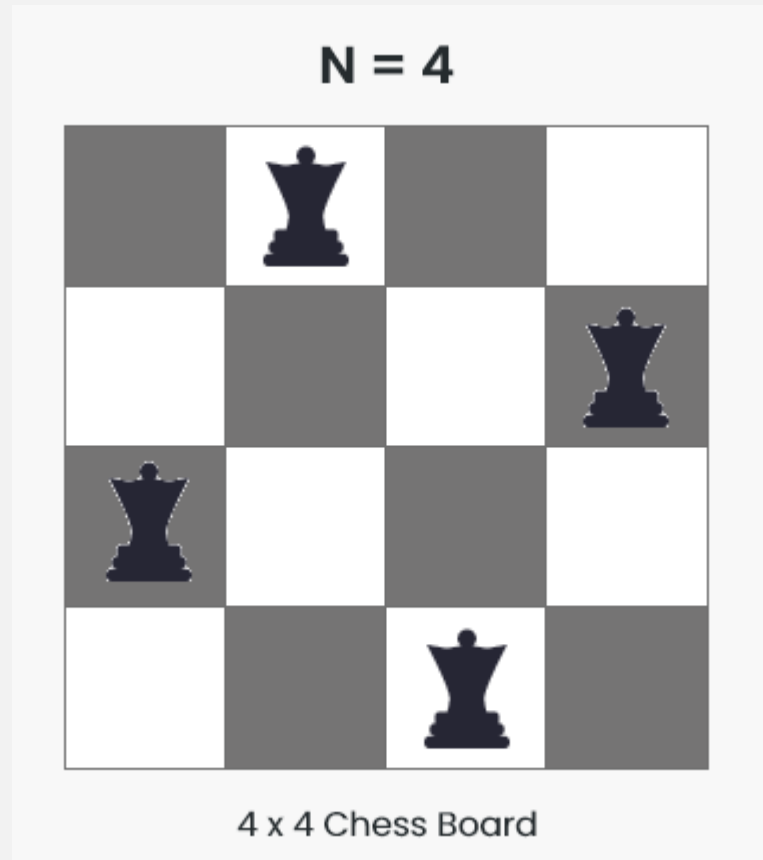
Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Example: 4-queens - Backtracking




Example: 4-queens – as code

```
;; termination argument:  
;; recursive calls “smaller” bc ...  
(define (find-sol x y ...)  
  (cond  
    [(done? x y curr-solution ...) ... DONE ...] ;; base case  
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]  
    [(at-last-row? ... y ...) ... NO-SOLUTION ... ]
```


Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
    [(done? x y curr-solution ...) ... DONE ...] ;; base case
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(at-last-row? ... y ...) ... NO-SOLUTION ... ]
```



Accumulator!

Example: 4-queens – as code

```
;; termination argument: ???
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
    [(done? x y curr-solution ...) ... DONE ...] ;; base case
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(at-last-row? ... y ...) ... NO-SOLUTION ... ]
    [else
     (if (no-threaten? x y current-solution)
         (let ([maybe-sol
                (find-sol x (next y) (update x y curr-solution))])
           (if (valid? maybe-sol)
               maybe-sol
               (find-sol (next x) y curr-solution)))
         (find-sol (next x) y curr-solution))]))
```

Number of “possible solutions to try” is reduced

Optimistically place queen

Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
    [(done? x y curr-solution ...) ... DONE ...] ;; base case
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(at-last-row? ... y ...) ... NO-SOLUTION ... ]
    [else
     (if (no-threaten? x y current-solution)
         (let ([maybe-sol
                (find-sol x (next y) (update x y curr-solution))])
           (if (valid? maybe-sol)
               maybe-sol
               (find-sol (next x) y curr-solution)))
         (find-sol (next x) y curr-solution))]))
```

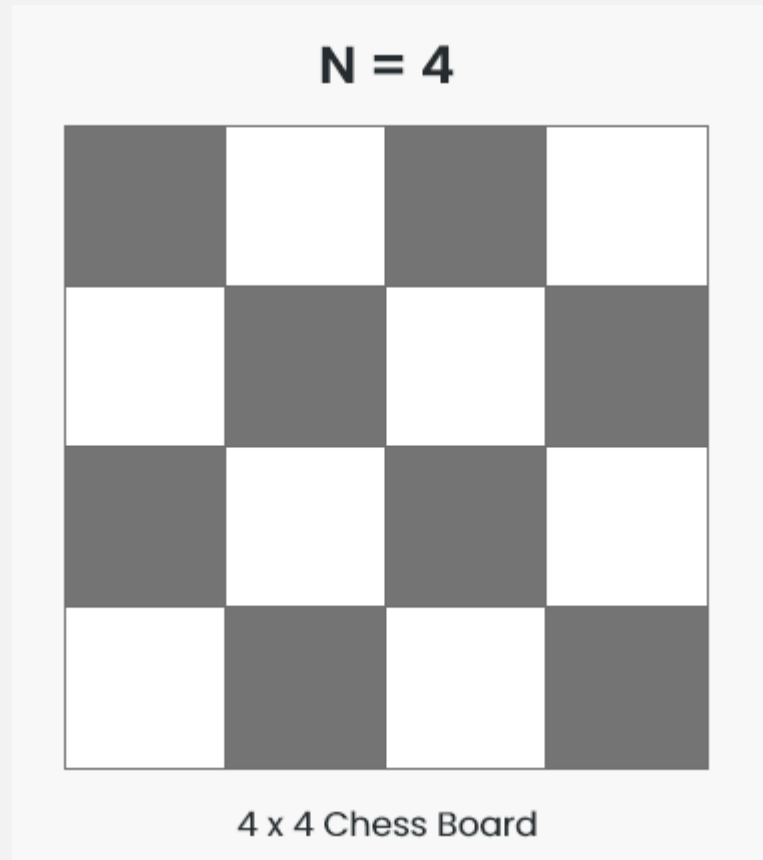
Number of “possible solutions to try” is reduced

Optimistically place queen

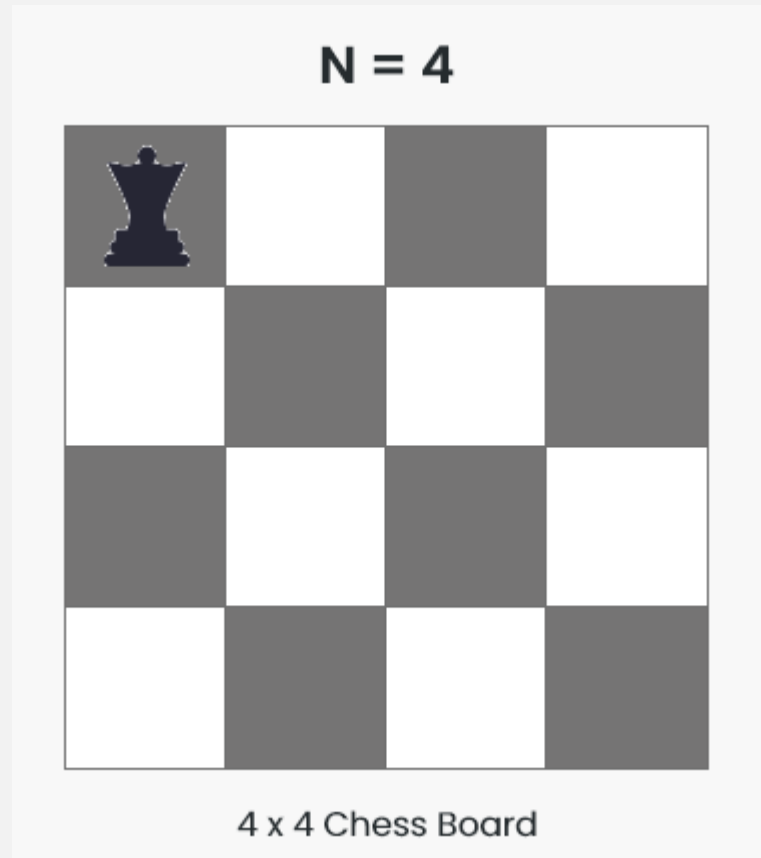
Need to check solution actually worked ...

Backtrack if it fails

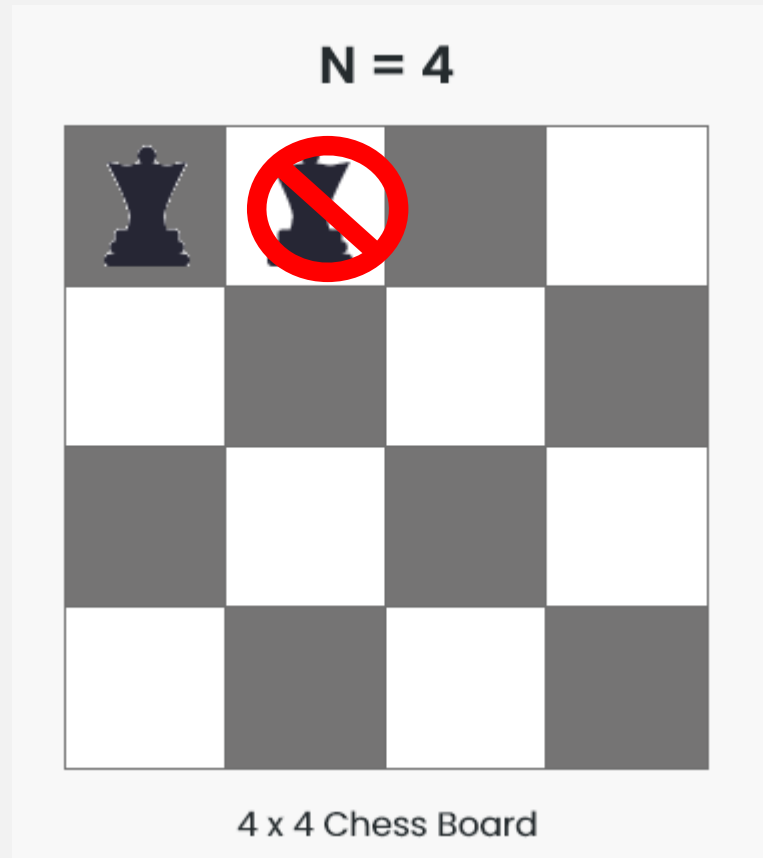
Example: 4-queens



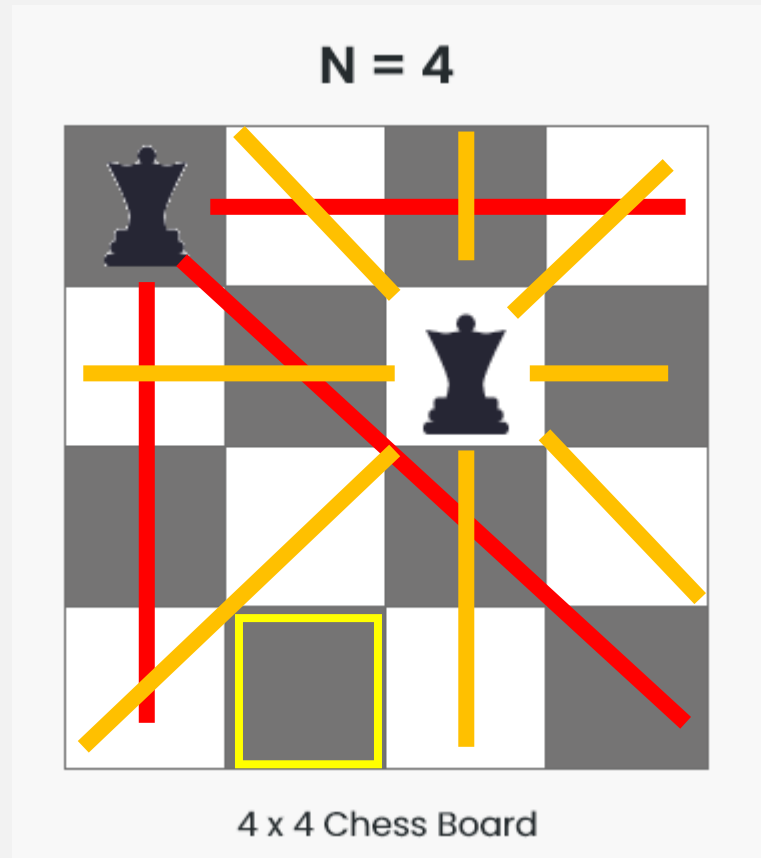
Example: 4-queens



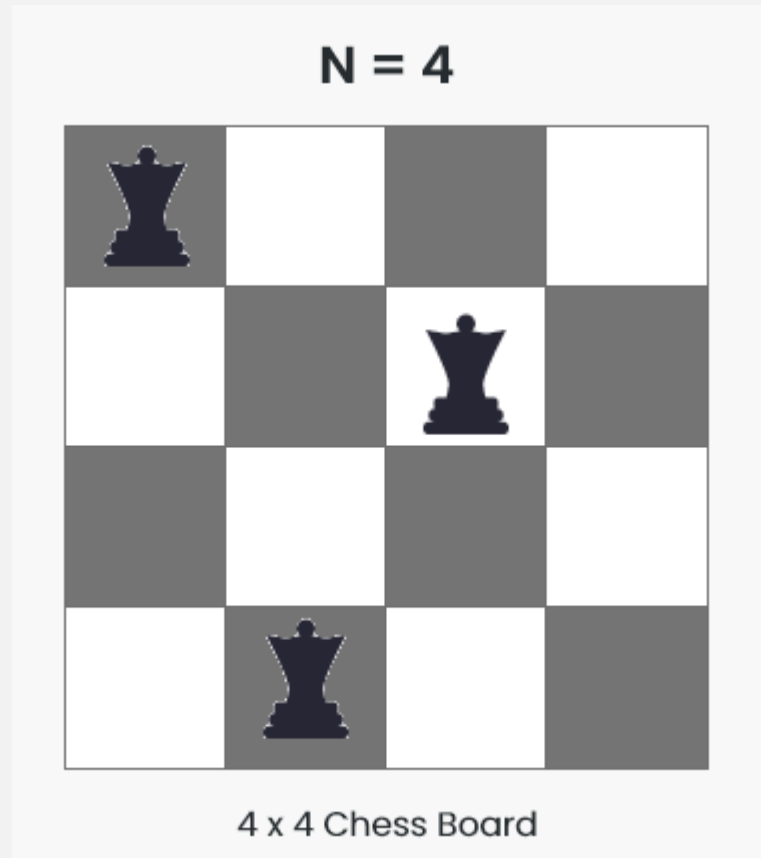
Example: 4-queens



Example: 4-queens



Example: 4-queens



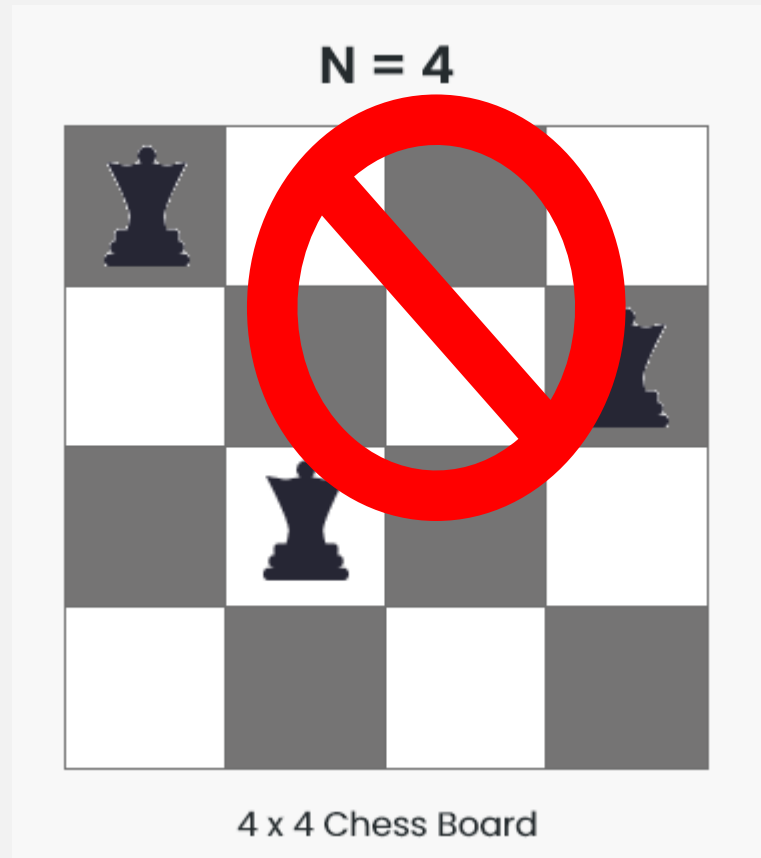
But ... need to place 4 queens!

FAIL???

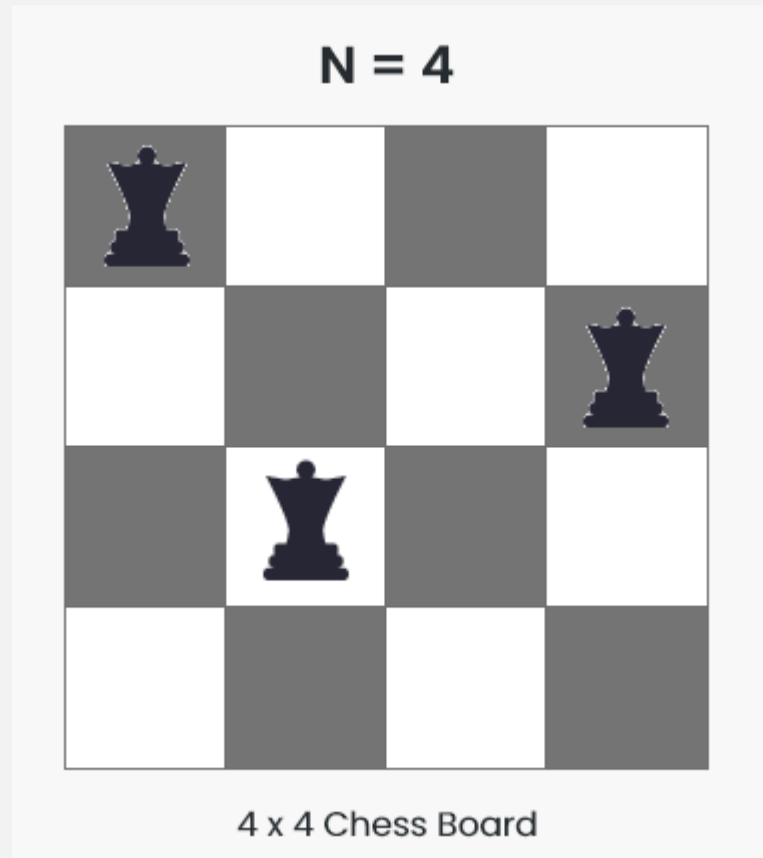
No, we havent tried all solutions ...

... need to go backwards

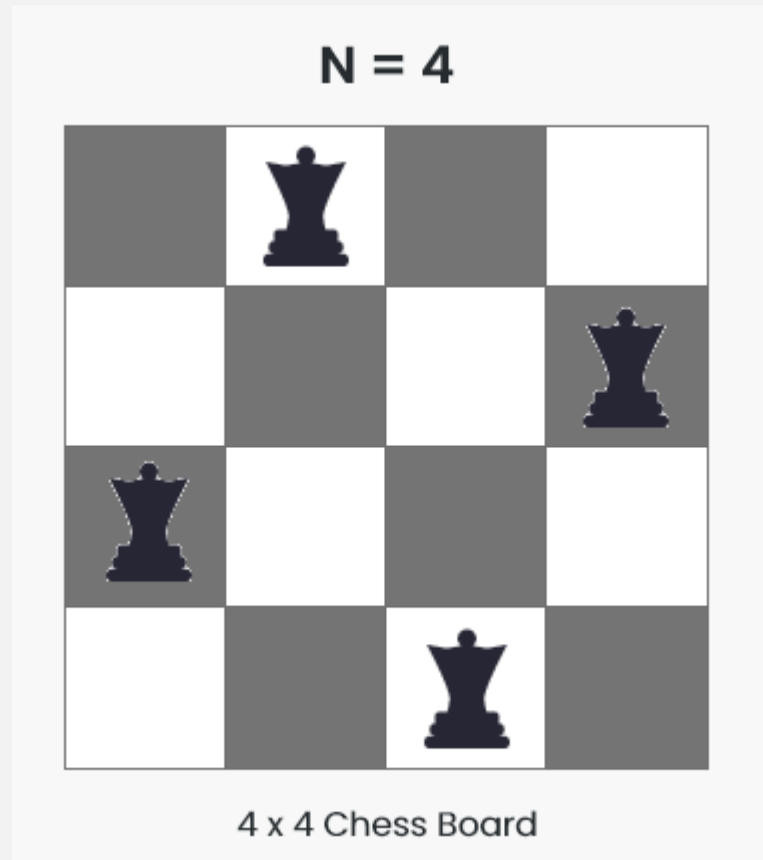
Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



In-class: Install “450 Lang”

Read the **Programming Language Specification** linked from HW description!

The screenshot shows the DrRacket interface with the 'Package Manager' window open. The 'File' menu is open, and 'Package Manager...' is highlighted with a blue arrow. The Package Manager window shows a search filter of '450lang' with 1/3538 matches. The '450lang' package is listed with a checkmark in the 'Package' column, highlighted by a yellow box. Below the list are buttons for 'Install', 'Remove', and 'Update'.

hw13.rkt - DrRacket

File Edit View Language Racket Insert

- New
- New Tab
- Open...
- Open Recent
- Open Require Path...
- Reopen Closed Tab
- Install .plt File...
- Install Package...
- Package Manager...
- Revert

Package Manager

File Edit Tabs Help

Do What I Mean Currently Installed Available from Catalog Copy from Version Settings

Filter: 450lang 1/3538 match Update Package List

✓: installed *: auto-installed !: not default scope =: installed as link; @: installed from URL

Package	Author	Description	Tags	Check...	Sou...	Catal...
✓ 450lang	stchang@racket-lang.org	Programming Language for UMB CS450 course		39123...	git+...	https...

Install Remove Update

(UPDATE if you installed last week)