

UMass Boston Computer Science
CS450 High Level Languages

Backtracking Design Recipe

Thursday, May 8, 2025

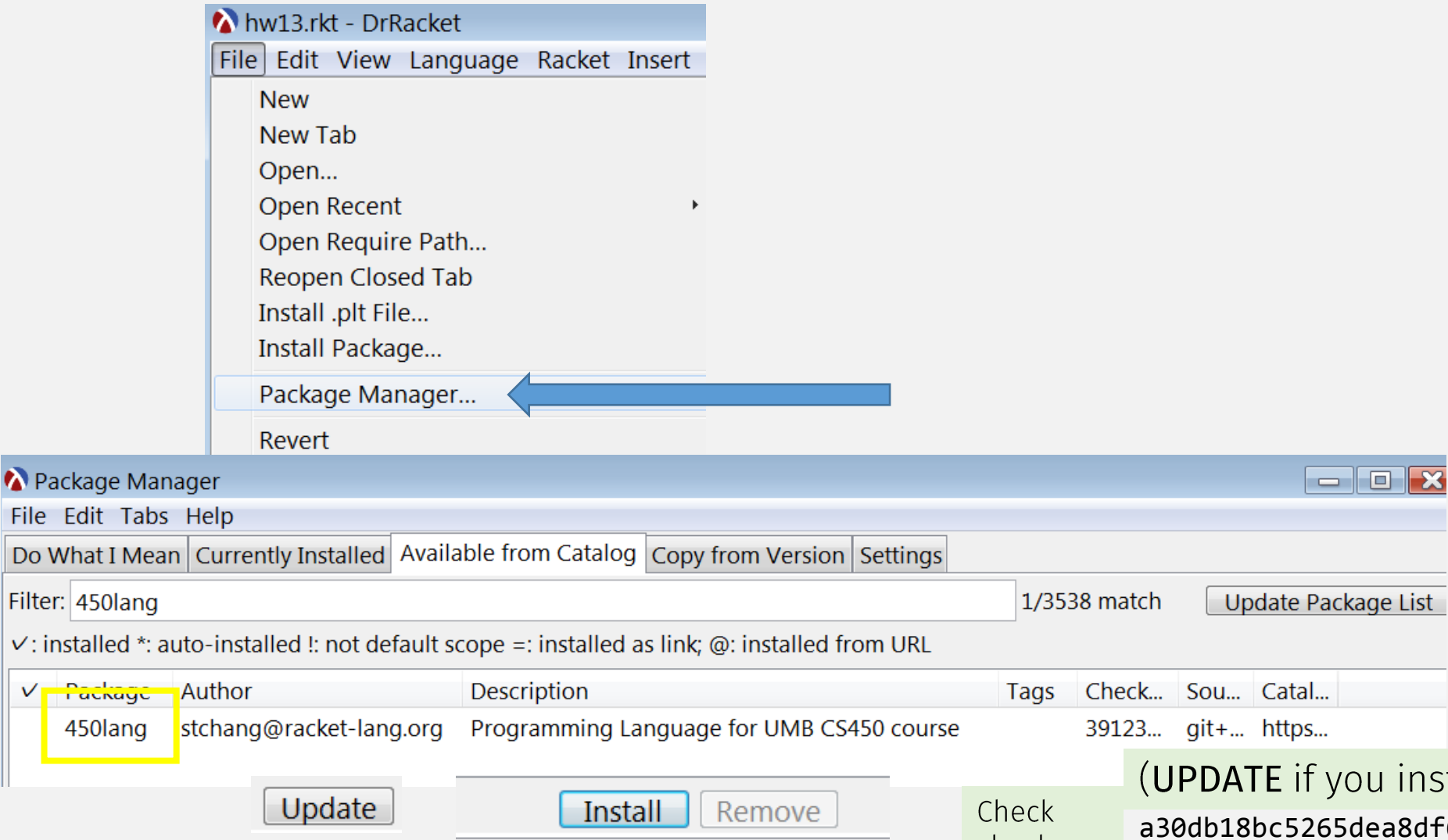
5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Logistics

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

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Installing “450 Lang”



The screenshot shows the DrRacket interface. The 'File' menu is open, and 'Package Manager...' is highlighted with a blue arrow. Below it, the Package Manager window is open, showing a list of packages. The '450lang' package is selected and highlighted with a yellow box. The package details show it was installed by 'stchang@racket-lang.org' and is described as 'Programming Language for UMB CS450 course'. The package has 39123 tags, a checksum of 'a30db18bc5265dea8df6d619959c67869d4b333f', and is available from 'git+' and 'https' sources. The 'Update' button is highlighted with a green box.

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...	

Update Install Remove

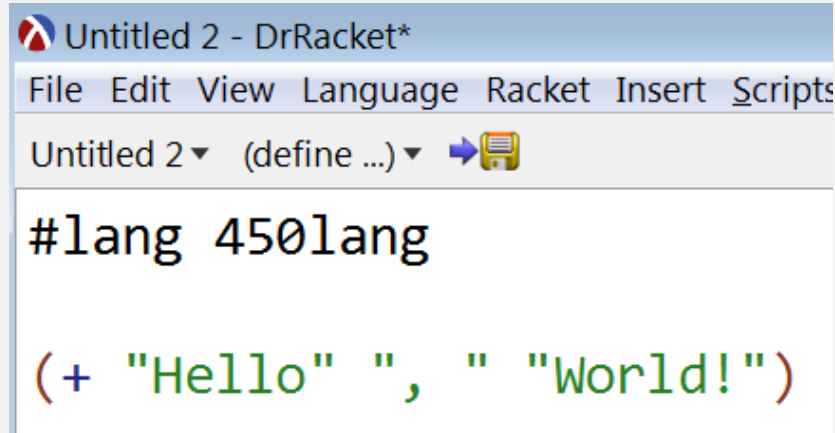
Check checksum

(UPDATE if you installed last week)

a30db18bc5265dea8df6d619959c67869d4b333f

<https://pkgs.racket-lang.org/package/450lang>

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)

Generative (non-structural) Recursion Design Recipe

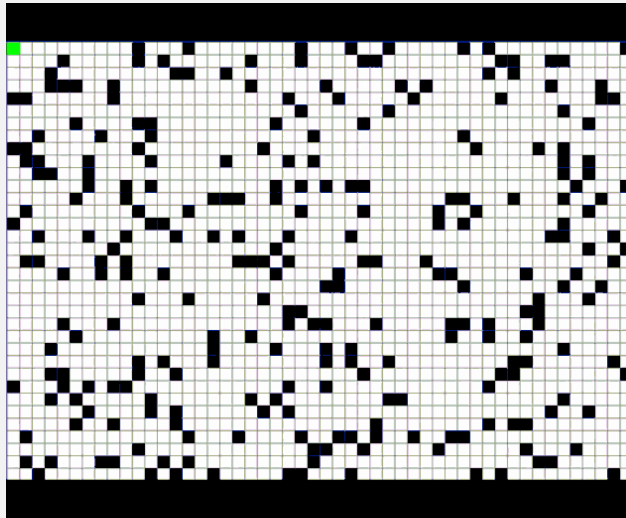
1. Name, Signature
2. Description
 - Must include **Termination Argument**
3. Examples
 - Even more important now!
4. **Code** (No structural template, but can use a “general” template)
5. Tests

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

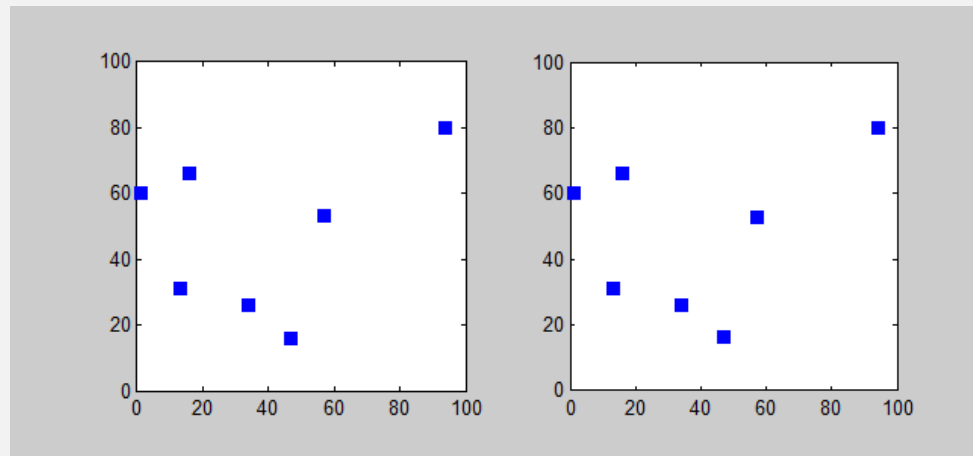
Backtracking

- A recursive algorithm for finding solutions to many computational problems that ...
 - ... tries potential solutions optimistically ... but “backtracks” when stuck
 - Graph algorithms, e.g., Path finding



Backtracking

- A recursive algorithm for finding solutions to many computational problems that ...
 - ... tries potential solutions optimistically ... but “backtracks” when stuck
 - Graph algorithms, e.g., Path finding
 - Optimization, e.g., knapsack, “traveling salesman”



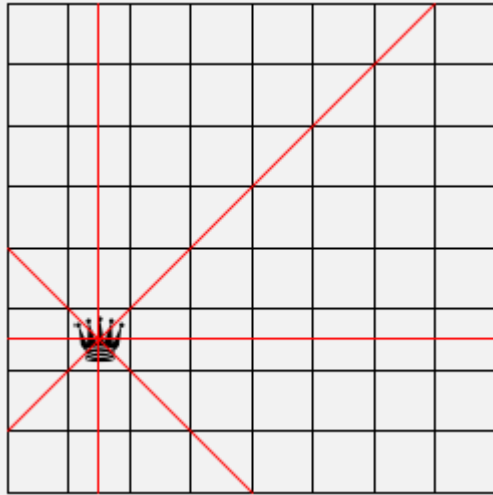
Backtracking

- A recursive algorithm for finding solutions to many computational problems that ...
 - ... tries potential solutions optimistically ... but “backtracks” when stuck
 - Graph algorithms, e.g., Path finding
 - Optimization, e.g., knapsack, “traveling salesman”
 - Solving puzzles, e.g., Sudoku, n-queens

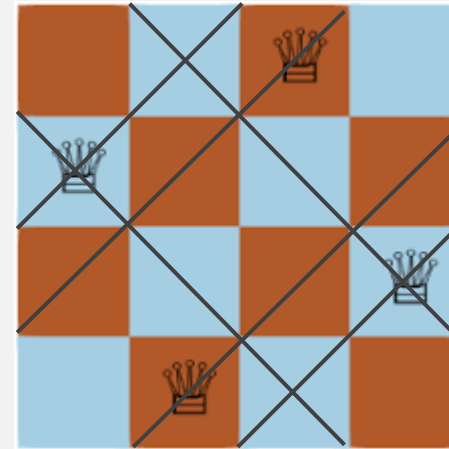
5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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

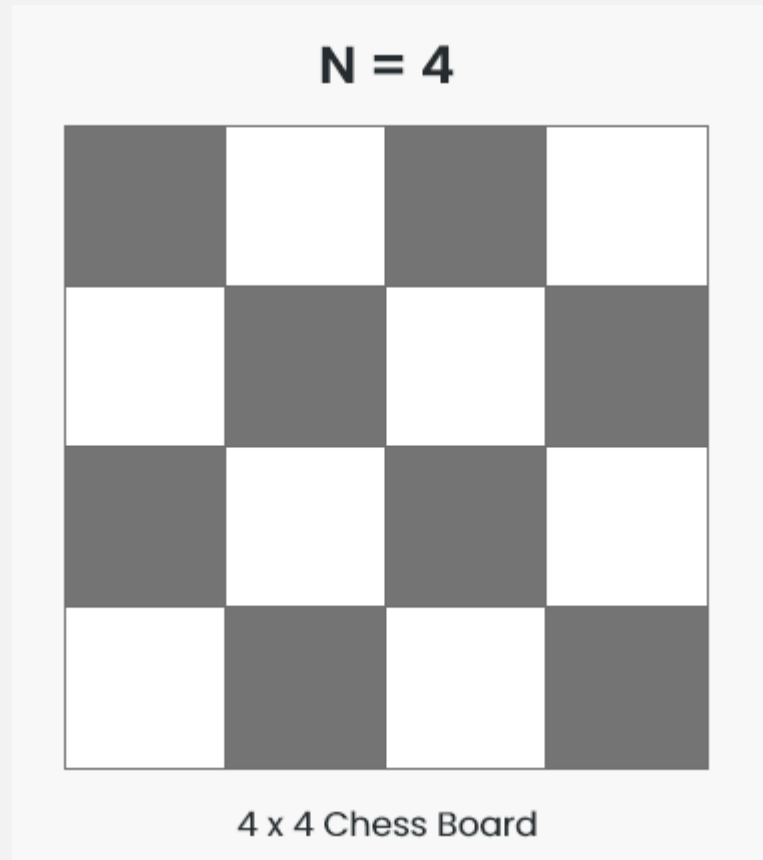


All queens safe

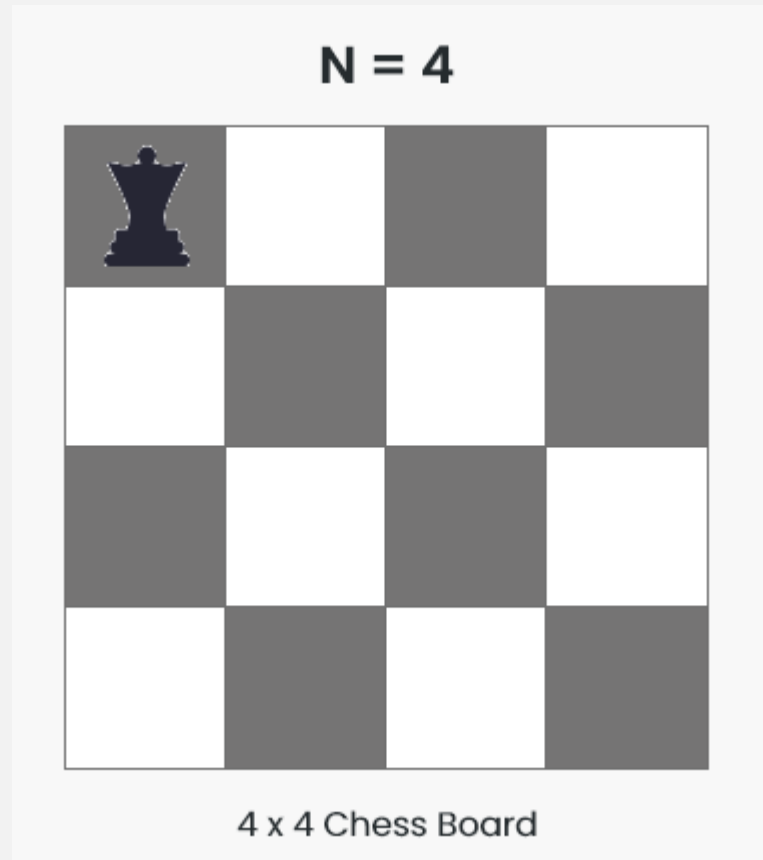
N-Queens problem – solving ...

- 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 ???

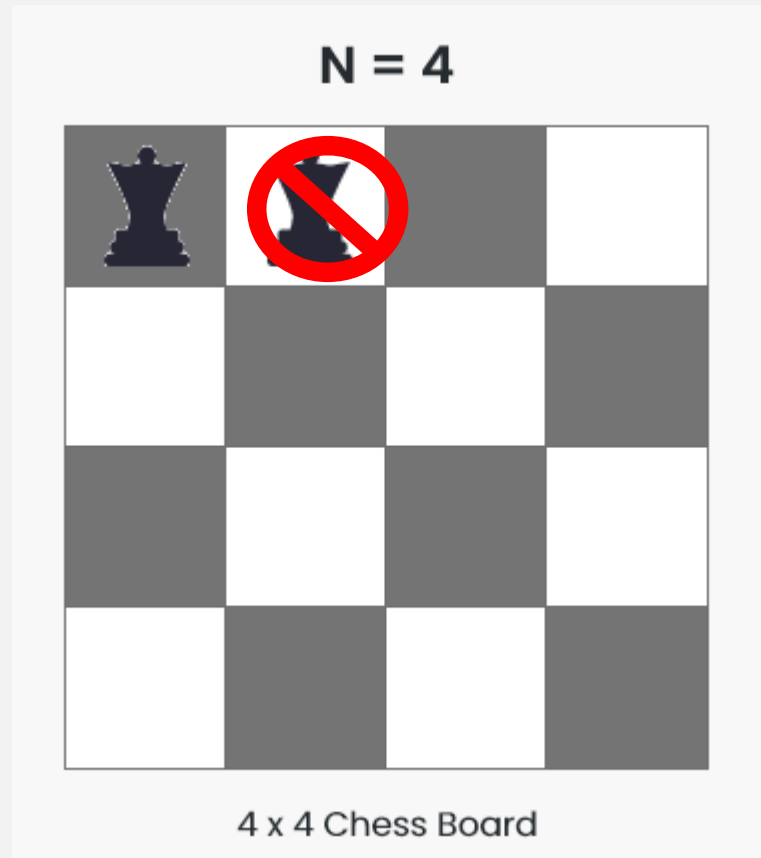
Example: 4-queens



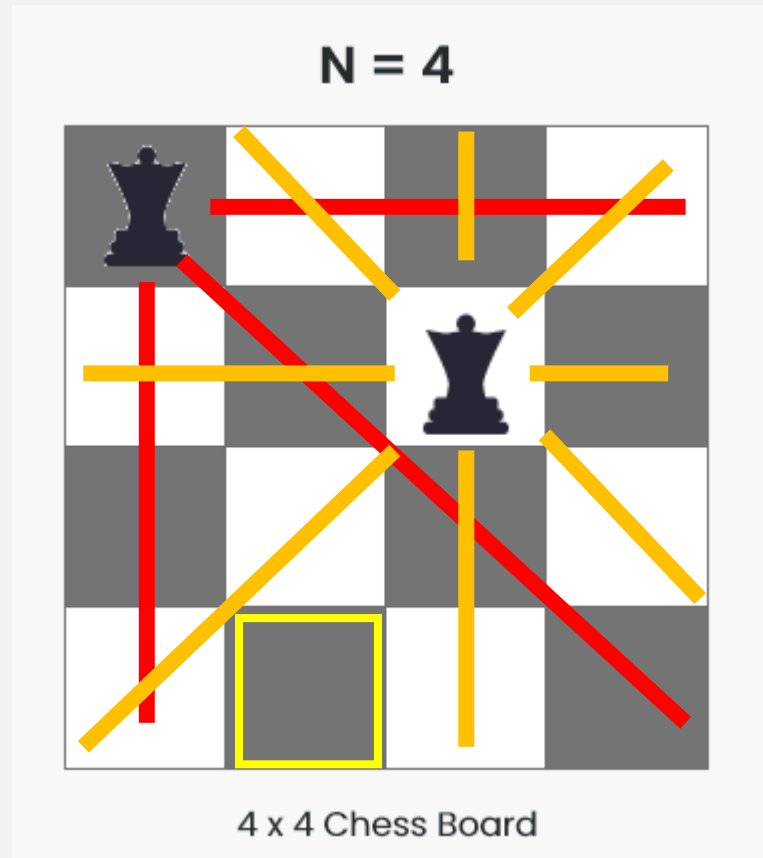
Example: 4-queens



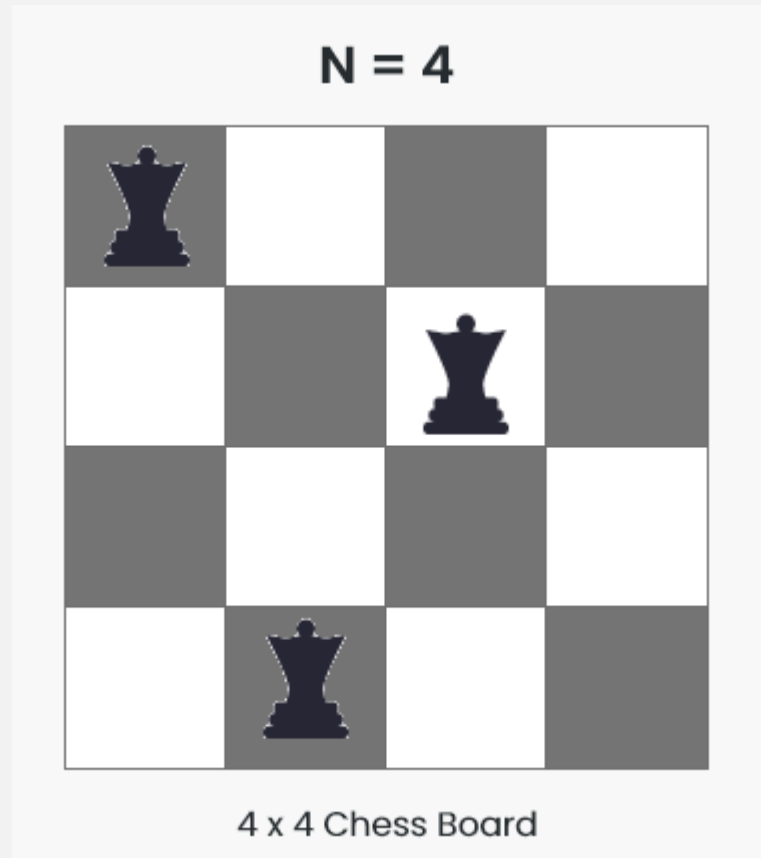
Example: 4-queens



Example: 4-queens



Example: 4-queens



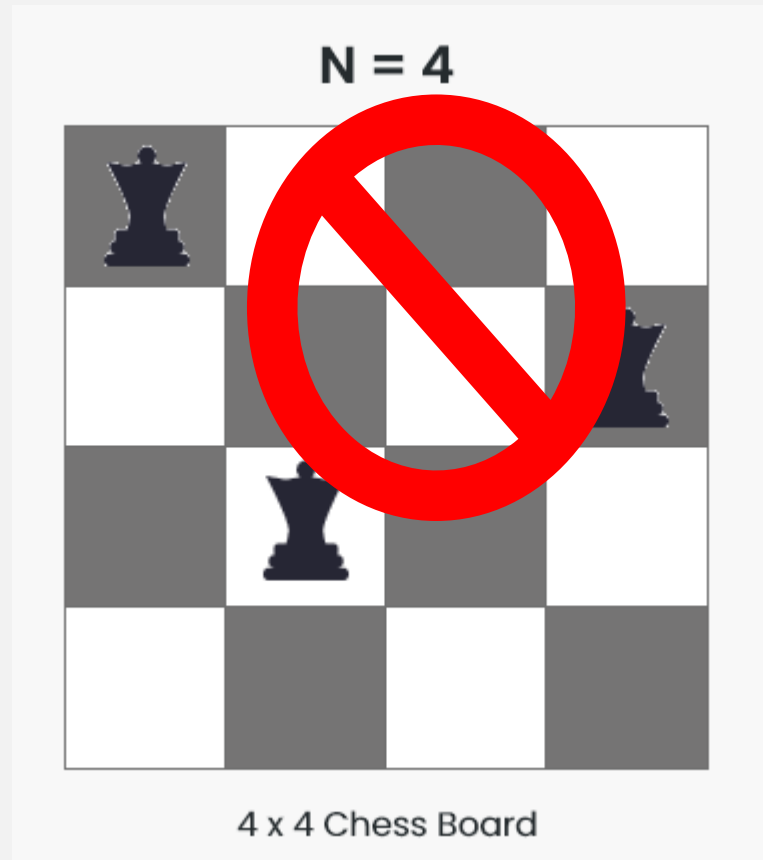
But ... need to place 4 queens!

FAIL???

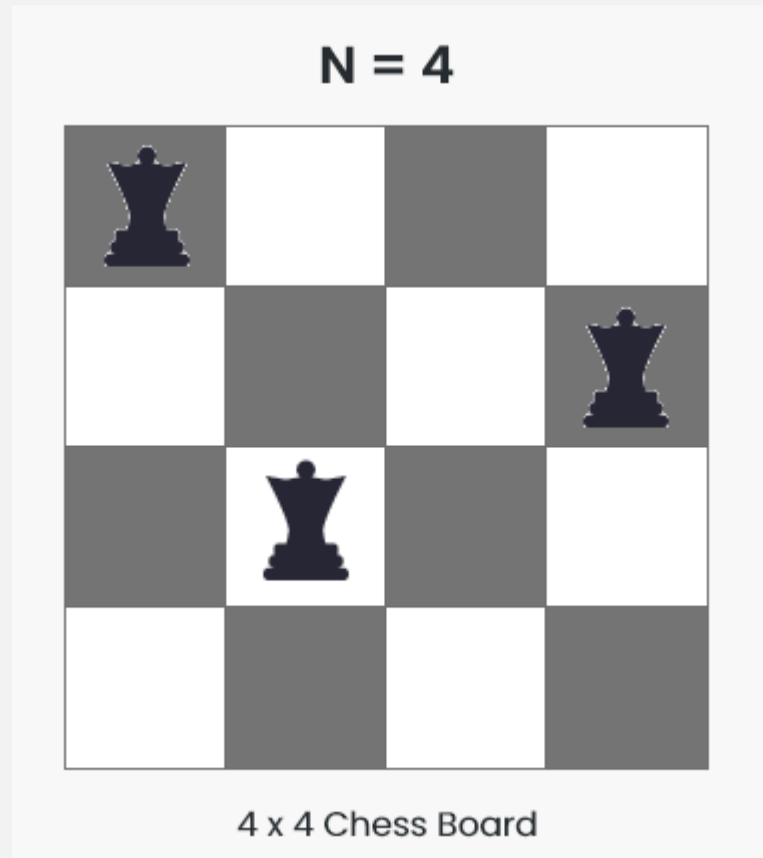
No, we havent tried all solutions ...

... need to go backwards

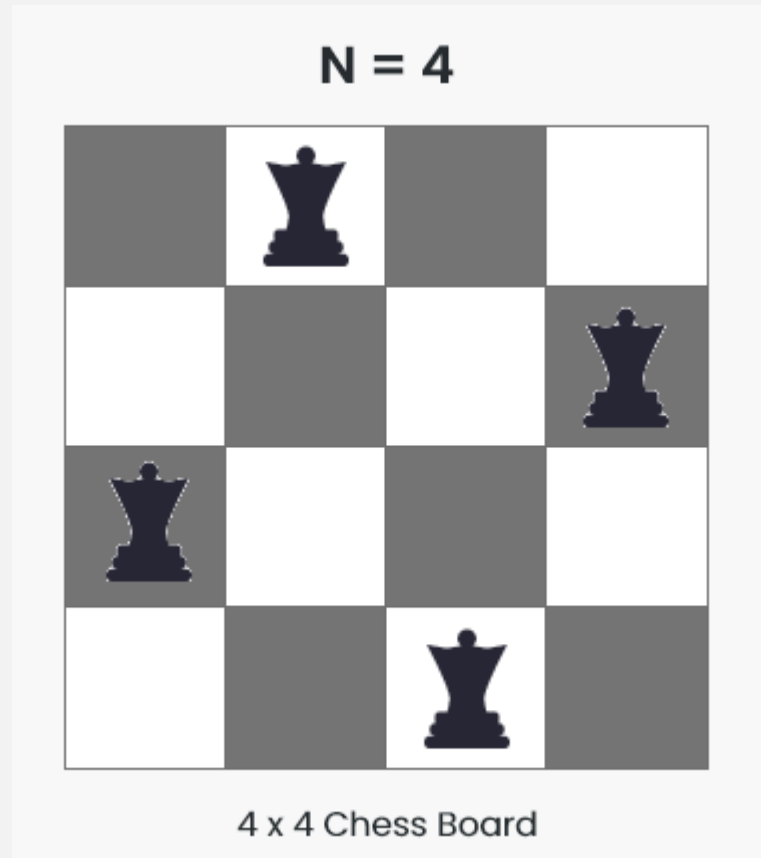
Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Backtracking Design Recipe

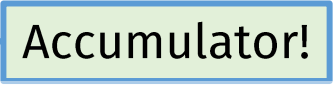
- Combination of other “recipes”
 - **Accumulator** – for “current solution”
 - **Generative Recursion**
 - Description must include **Termination Argument**
- Code “Template”
 - 2 base cases
 - Success
 - Fail
 - Recursive call ...
 - Should optimistically move forward towards potential solution by placing a queen ...
 - ... but result must be checked! And **backtrack** if fail ...

Example: 4-queens – as code

```
;; termination argument:  
;; recursive calls “smaller” bc ...  
(define (find-sol x y ...)  
  (cond  
    [(done? ...) ... DONE ...]  
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]  
    [(no-solution? ... ) ... FAIL-RESULT ... ]
```

Example: 4-queens – as code

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



Accumulator!

Example: 4-queens – as code

```
;; termination argument: ???
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
    [(done? curr-solution ...) ... DONE ...]
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(no-solution? ... ) ... FAIL-RESULT ... ]
    [else
     (if (no-threaten? x y curr-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? curr-solution ...) ... DONE ...]
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(no-solution? ... ) ... FAIL-RESULT ... ]
    [else
     (if (no-threaten? x y curr-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

Backtracking
algorithm must be
able to quickly
validate a
potential solution

Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ... less possible solutions to try
(define (find-sol x y curr-solution)
  (cond
    [(done? curr-solution ...) ... DONE ...]
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
    [(no-solution? ... ) ... FAIL-RESULT ... ]
    [else
     (if (no-threaten? x y curr-solution)
         (let ([maybe-sol
                (find-sol x (next y) (update x y curr-solution))])
           (if (false? maybe-sol)
               maybe-sol
               (find-sol (next x) y curr-solution)))
         (find-sol (next x) y curr-solution))]))
```

Produce “false” value to indicate no solution

Backtracking
algorithm must be
able to quickly
validate a
potential solution

Example: 4-queens – as code

```
;; nqueens : Nat -> List<Queen>
```

```
;; ...
```

```
(define (find-sol x y curr-sol)
```

;; A Queen is a
;; ... row and column ...

```
(cond
```

```
[(done? curr-solution ...) ... DONE ...]
```

```
[(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
```

```
[(no-solution? ... ) ... FAIL-RESULT ... ]
```

```
[else
```

Produce “false” value to indicate no solution

```
(if (no-threaten? x y curr-solution)
```

```
(let ([maybe-sol
```

```
(find-sol x (next y) (update x y curr-solution))])
```

```
(if (false? maybe-sol)
```

```
maybe-sol
```

```
(find-sol (next x) y curr-solution))
```

```
(find-sol (next x) y curr-solution))]))
```

Example: 4-queens – as code

```
;; nqueens : Nat -> Maybe<List<Queen>>
```

```
;; ...
```

```
(define (find-sol x y curr-solution)
```

```
  (cond
```

```
    [(done? curr-solution ...) ... DONE ...]
```

```
    [(at-last-col? ... x ...) (find-sol FIRST-X (next y) ...)]
```

```
    [(no-solution? ... ) ... FAIL-RESULT ... ]
```

```
    [else
```

```
      (if (no-threaten? x y curr-solution)
```

```
        (let ([maybe-sol
```

```
              (find-sol x (next y) (update x y curr-solution)))]
```

```
        (if (false? maybe-sol)
```

```
            maybe-sol
```

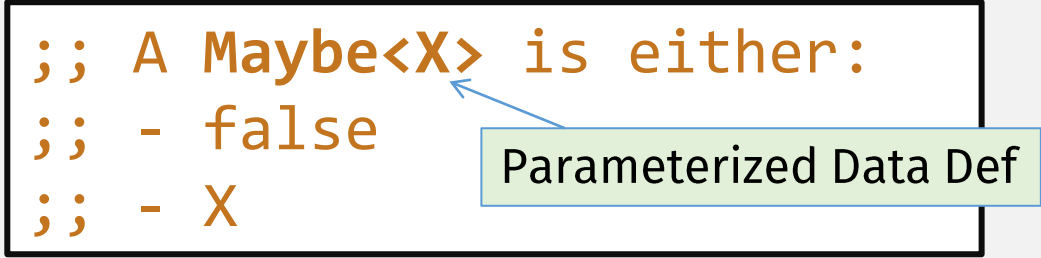
```
            (find-sol (next x) y curr-solution))
```

```
        (find-sol (next x) y curr-solution))]))
```

Produce “false” value to indicate no solution

Maybe Data Definitions

```
;; nqueens : Nat -> Maybe<List<Queen>>  
;; ...
```



A black arrow points from the `Maybe` type in the first code block to the top of this callout box. Inside the box, the definition of `Maybe` is given. A light blue arrow points from the text 'Parameterized Data Def' to the `X` in the definition.

```
;; A Maybe<X> is either:  
;; - false  
;; - X
```

Parameterized Data Def

N-queens Solution Validation

- Still useful to write a `valid?` predicate, i.e., for testing
- A “valid” n-queens solution has
 - n (unique) queens
 - No queens threaten any other



```
(define (2queens-safe? q1 q2) (not (threaten? q1 q2)))
```

```
(define (threaten? q1 q2)  
  (or (same-row? q1 q2)  
      (same-col? q1 q2)  
      (same-diag? q1 q2)))
```

```
(define (queenlist-safe? qlst)  
  (andmap ... 2queens-safe? ... qlst ... ))
```