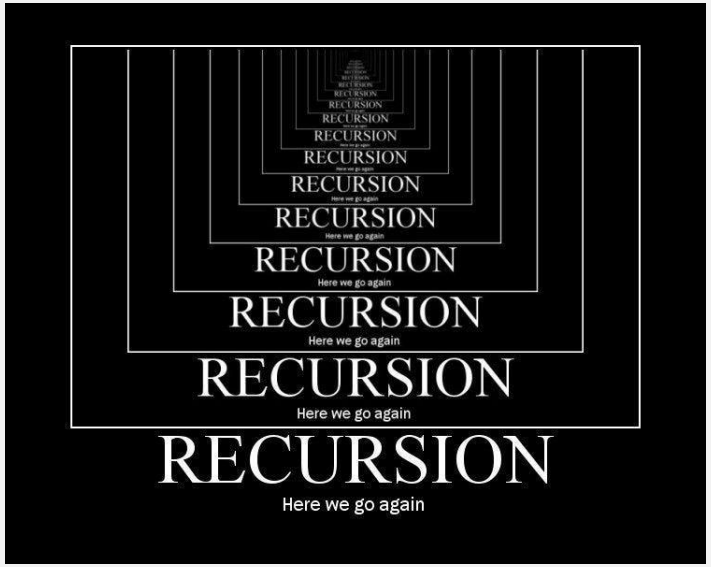
UMass Boston Computer Science

CS450 High Level Languages

Generative Recursion

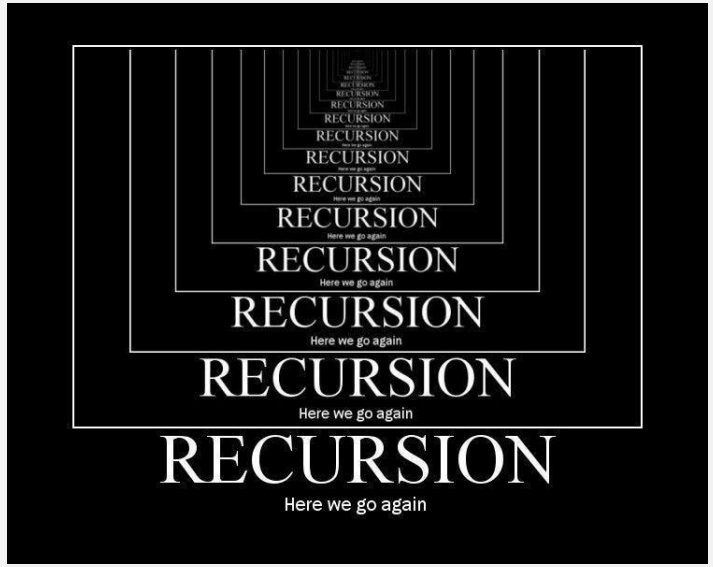
(going beyond the template?!)

Tuesday, April 28, 2026



Logistics

- HW 11 out
  - due: Thu 4/30 11am EST



Previously

`bind` in "CS450" Lang

```
;; A Variable (Var) is a Symbol
```

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
;; …
```

Reference a variable binding

new binding is in-scope (can be referenced) here

Create a new variable binding

new binding is **not** in-scope here

Previously

`bind` examples

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
;; …
```

new binding is **not** in-scope here

```
(check-equal?
  (eval450
   '(bind [x (+ x 20)]
      x))
   UNDEFINED-ERROR )
```

???

`bind` examples, with functions

```
;; A Prog is one of:
;; …
;; - Var
;; - `(bind [,Var ,Prog] ,Prog)
   `(lm ,List<Var> ,Prog)
   (cons Prog List<Prog>)
;; …
```

"lambda" function

function call

function

arguments

```
(check-equal?
  (eval450
   '(bind [f (lm (x) (+ x 4))]
      (f 6)))
   10 )
```

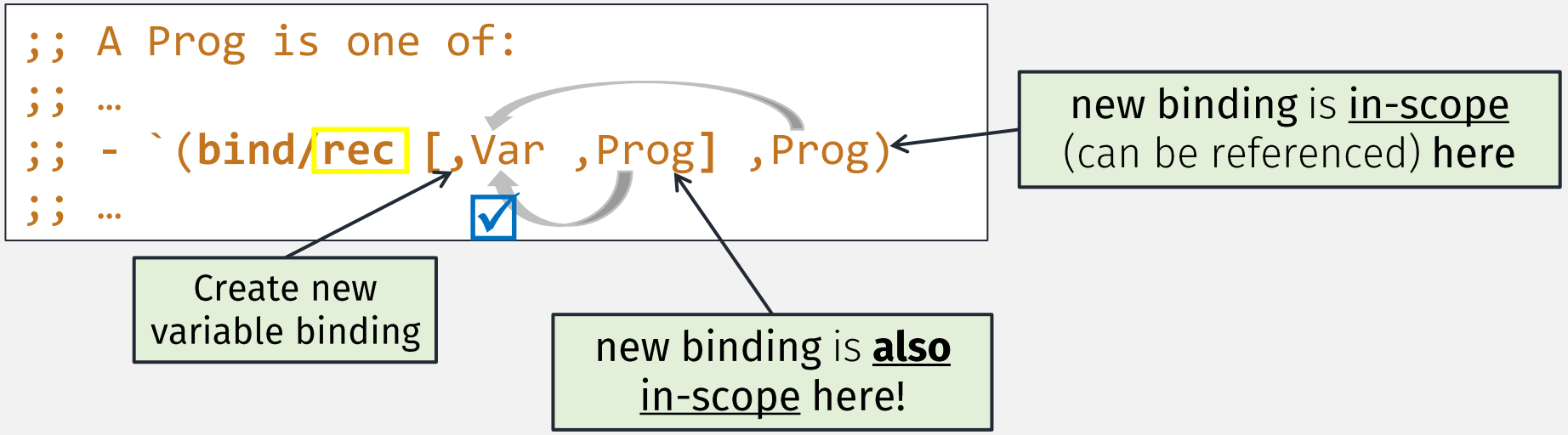
`f` not in-scope here
(so function can't be recursive!)

```
(check-equal?
  (eval450
   '(bind [f (lm (x) (f x))]
      (f 6)))
   UNDEF-ERR)
```

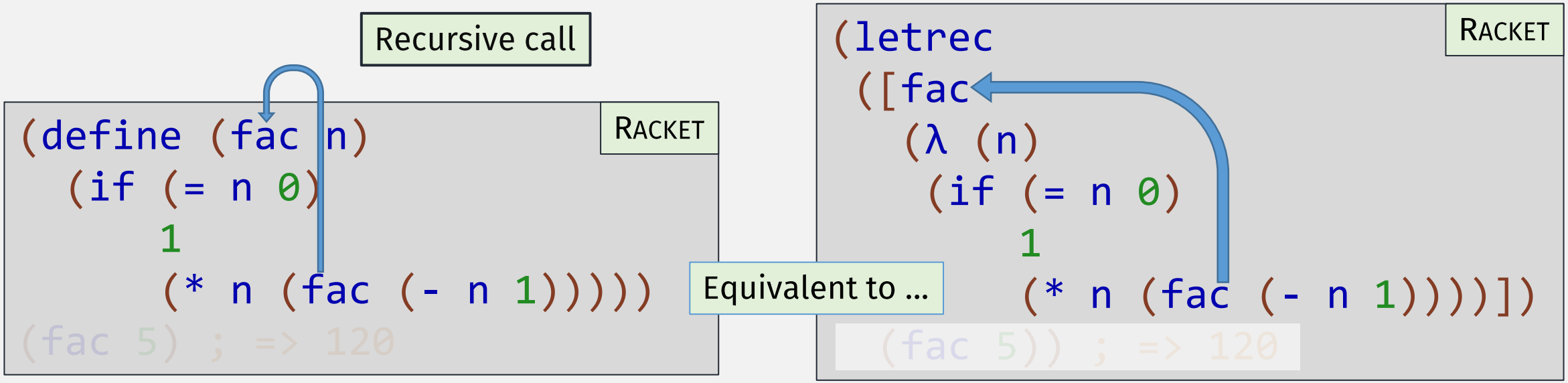
`f` not in-scope here
(so function can't be recursive!)

How to add recursion to our language?

`bind/rec` in “CS450” Lang



Racket recursive function examples



`bind/rec` examples

```
;; A Prog is one of:
;; …
;; - `(bind/rec [,Var ,Prog] ,Prog)
;; - `(iffy ,Prog ,Prog ,Prog)
;; …
```

JS “truthy if” (hw10)

RACKET

```
(letrec
 ([fac
   (λ (n)
    (if (= n 0)
        1
        (* n (fac (- n 1)))))])
  (fac 5)) ; => 120
```

Equivalent to …

CS450LANG

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

Zero is “falsy” (hw10)

Assume “-” primitive in `INIT-ENV`

Racket `define` is `lambda` …

Racket

```
(define (f n)
  (- n 1))
```

Equivalent to …

Racket

```
(define f
  (λ (n)
    (- n 1)))
```

RACKET `define` is `lambda` and `letrec`

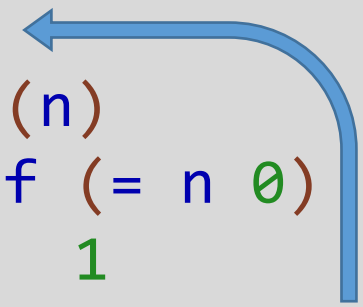
RACKET

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Equivalent to …

RACKET

```
(define factorial
  (letrec
   ([fac
     (λ (n)
      (if (= n 0)
          1
          (* n (fac (- n 1)))))])
    fac))
```



The entire **`letrec`** evaluates to the **recursive `fac` function**

Running **`bind/rec`** programs

```
;; A Prog is one of:
;; …
;; - `(bind/rec [,Var ,Prog] ,Prog)
;; …
```

parse

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

run

```
;; A Result is a:
;; - …
```

Running **bind/rec** programs

TEMPLATE ?

```
;; run: AST -> Result
;; Computes result of
running CS450 Lang AST
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

run

```
;; A Result is a:
;; - …
```

Running **bind/rec**

TEMPLATE : extract pieces

```
;; run: AST -> Result
(define (run p)

  (define (run/e p env)
    (match p
      …
     [(recb x e body) ?? x   ??      e       ??      body ))))]
      … ))
 (run/e p  ???  ))
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

Running **bind/rec**

TEMPLATE : recursive call

```
;; run: AST -> Result
(define (run p)

  (define (run/e p env)
    (match p
      …
     [(recb x e body) ?? x ?? (run/e e ??) ?? (run/e body ??) ))]
      … ))
 (run/e p  ???  ))
```

```
;; An AST is one of:
;; …
;; - (mk-recb Symbol AST AST)
;; …
(struct recb [var expr body])
```

Running **`bind/rec`**, using old Environment (doesn't work)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
      [(recb x e body) ?? x ?? (run/e e ??) ?? (run/e body ??) ))]
      … ))
 (run/e p INIT-ENV ))
```

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

Running **`bind/rec`**, using old Environment (doesn't work)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(recb x e body)
      (define env/x (env-add env x (run/e e env))
      (run/e body env/x)]
      … ))
 (run/e p INIT-ENV ))
```

2. add x binding to environment

1. Compute **`Result`** for x

Running **bind/rec**, using old Environment (doesn't work)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
      [(recb x e body)
       (define env/x (env-add env x (run/e e
       (run/e body env/x)]
      … ))
  (run/e p INIT-ENV))
```

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

CS450LANG Recursive Example

fac should be in-scope here

??? This is circular! (no base case)

env/x

PROBLEM:
x should be in-scope here too!

Compute **body** with **x** in-scope

Running **`bind/rec`**, using recursive Environment

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p    …
     [(recb x e body)

      (define env/x (env-add env x (run/e e env/x)))


      (run/env body env/x)]
                …  )
 (run/e p INIT-ENV )
```

??? This is **circular**! (no base case)

Compute **body** with **x** in-scope

PROBLEM:
x should be in-scope here too!

Running **bind/rec**, using recursive Environment

```
(bind/rec [f f] f)
 ; => CIRCULAR-ERROR
```

CS450LANG

Non-function, circular recursive references (no base case) should **produce error result!**

```
;; A Result is a:
;; - Number
;; - FunctionResult
;; - ErrorResult
```

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))

      (run/env body env/x)]
                …  ))
 (run/e p INIT-ENV ))
```

Creates **mutable box.** Makes **mutation explicit**

```
;; An ErrorResult is a:
;; - UNDEFINED-ERROR
;; - ARITY-ERROR
;; - CIRCULAR-ERROR
```

Running **`bind/rec`**, using recursive Environment

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p    …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder)
                    …
      (run/env body env/x)]
                  … ))
 (run/e p INIT-ENV ))
```

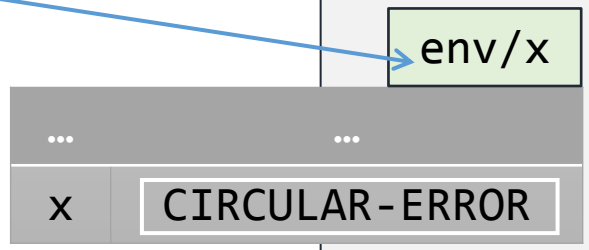
```
;; An Environment (OLD) (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

???

(how would **`env-add`** and **`env-lookup`** need to change?)

```
;; An Environment (Env) is a: Listof<(list Var EnvVal)>
```

```
;; An EnvVal is one of:
;; - Result
;; - Box<Result>
```



Running **`bind/rec`**, using recursive Environment

CS450LANG

```
(bind/rec [f f] f)
 ; => CIRCULAR-ERROR
```

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))
      (define x-result (run/env e env/x))

      (run/env body env/x)]
                …  ))
 (run/e p INIT-ENV ))
```

Non-function, circular recursive references (no base case) should produce error result!

Compute **x**'s **Result** with **x** in-scope!

env/x

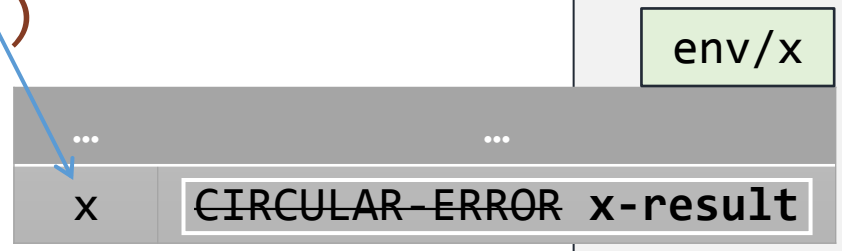
…	…
x	CIRCULAR-ERROR

Running **`bind/rec`**, using recursive Environment

```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR)
      (define env/x (env-add env x placeholder)
      (define x-result (run/env e env/x)
      (set-box! placeholder x-result)
      (run/env body env/x)]
                …  ))
  (run/e p INIT-ENV ))
```

Close the (**circular data** structure) **loop,** with **mutation!**

Explicitly **mutate** mutable box



Running **bind/rec**, using recursive Environment

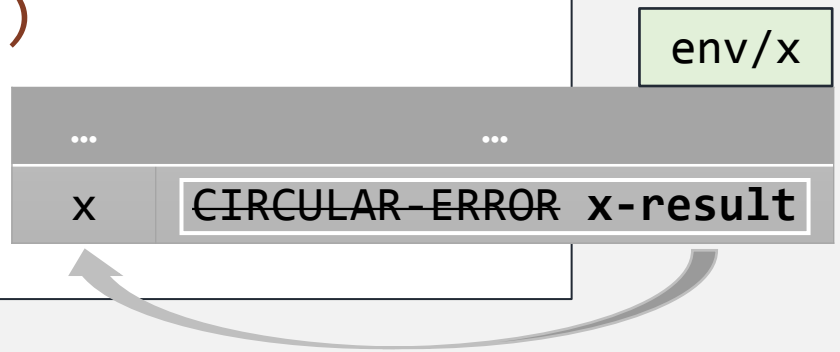
```
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p     …
     [(recb x e body)
      (define placeholder (box CIRCULAR-ERROR))
      (define env/x (env-add env x placeholder))
      (define x-result (run/env e env/x))
      (set-box! placeholder x-result)
      (run/env body env/x)]
                    …  ))
  (run/e p INIT-ENV ))
```

Compute body with **x** in-scope

CS450LANG

```
(bind/rec
 [fac
  (lm (n)
    (iffy n
          (* n (fac (- n 1)))
          1))]
  (fac 5)) ; => 120
```

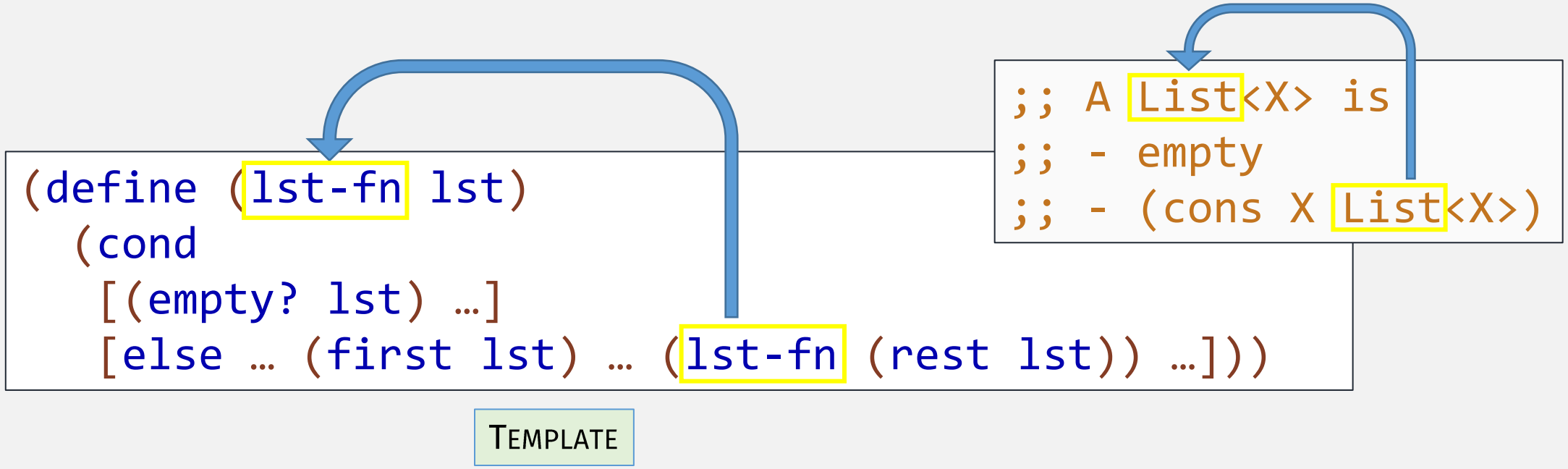
This should only "**work**" for **recursive function definitions!**



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) ev                    ia 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., not following **data defs**) is called **generative recursion**
- **no template? = no guaranteed termination … requires Termination Argument**
  - Explains how each recursive gets "smaller"**!**

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

(Functional) Quicksort

Generative (non-template) **Recursion Example!**

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

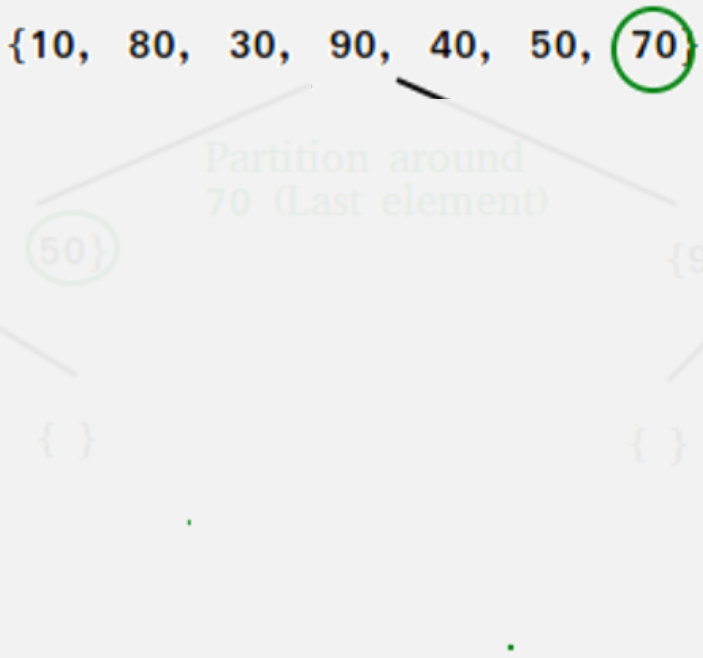
```
(check-equal?
  (smaller-than (list 1 3 4 5 9) 4)
                (list 1 3))
```

```
;; 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 lsts:
   - < pivot
   - >= pivot
3. Recurse on smaller lists
   - 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
   [(trivial? problem) (solve-easy lst)] ;; base case
   [else
    (define pivot (first lst))
    (combine-solutions
      (qsort (smaller-problem-1 lst))
                    …
      (qsort (smaller-problem-n 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>
;; terminati
;;                                        vot
   (c
    [(trivial: problem) (solve easy lst)] ;
    [else
     (combine-solutions (first lst))
       (qsort (filter (curry > pivot) (rest lst))
                      ...
       (qsort (filter (curry <= pivot) (rest lst)
```

Function "arithmetic"!

```
(curry f arg1)
=
(lambda (arg2) (f arg1 arg2))
```

Result is a function!

Curry = "partial apply"

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

"less than"

"greater than"

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
   [(trivial? problem) (solve-easy lst)] ;; base case
   [else
    (define pivot (first lst))
    (combine-solutions
      (qsort (filter (curry > pivot) (rest lst))
                     ...
      (qsort (filter (curry <= pivot) (rest lst)
```

"less than"

"greater than"

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))))]))
```

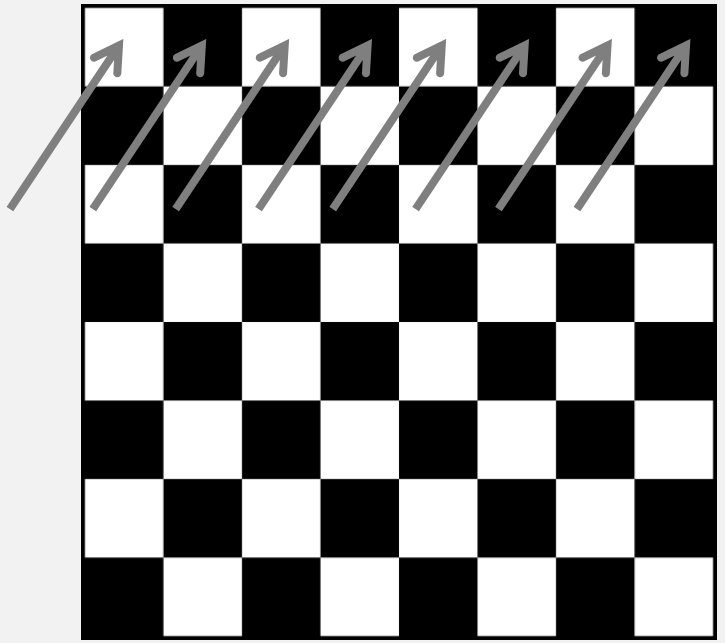
Gen Rec Example: (functional) quicksort

```
;; 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 ...
```

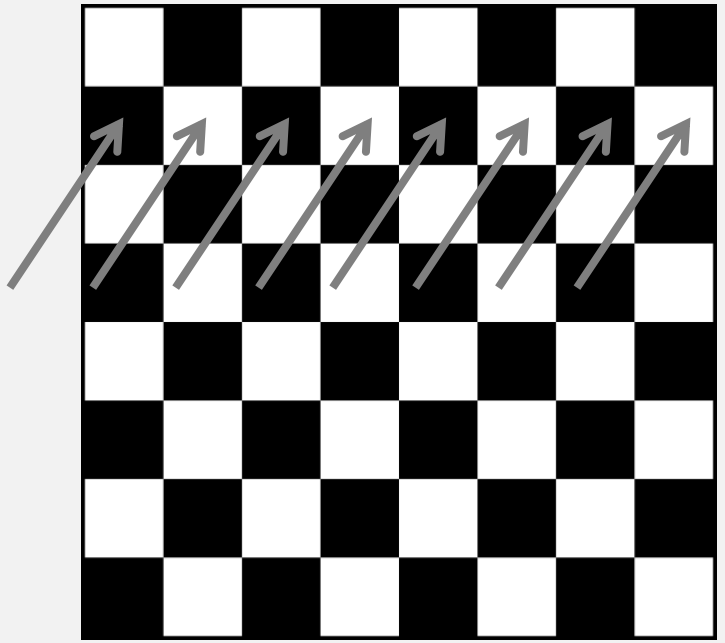
Example: **traversing a game board ...**

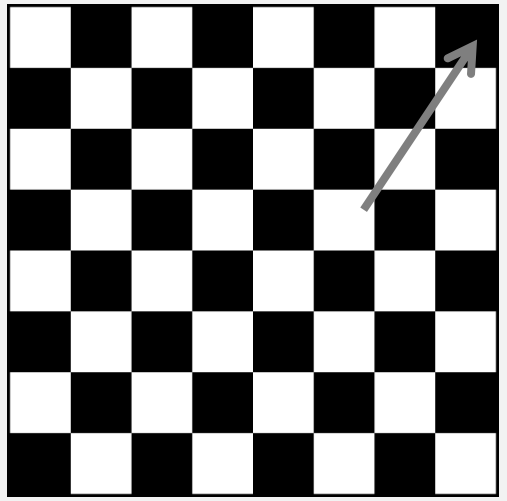


Not always obvious!

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

Example: **traversing a game board ...**





Not always obvious!

```
;; termination argument:
;; recursive calls "smaller" bc ...
(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
            …
                                                    ]))
```

Some "distance" to last square gets "smaller" ???

What is the "smaller" problem???

```
;; 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
                …
                                                    ]))
```

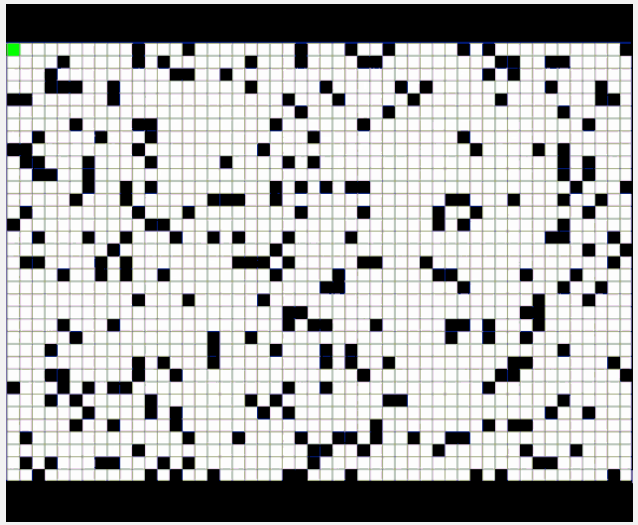
Is this always true???

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

What is the “smaller” problem???

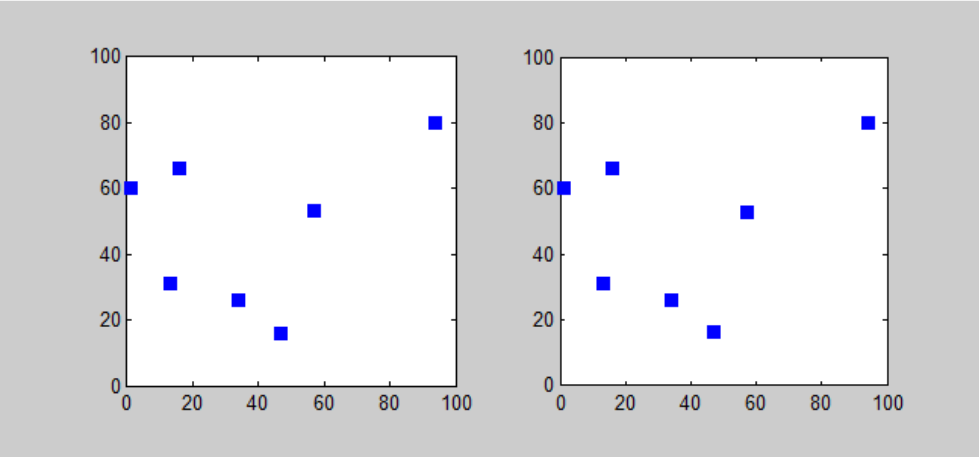
Backtracking

- 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



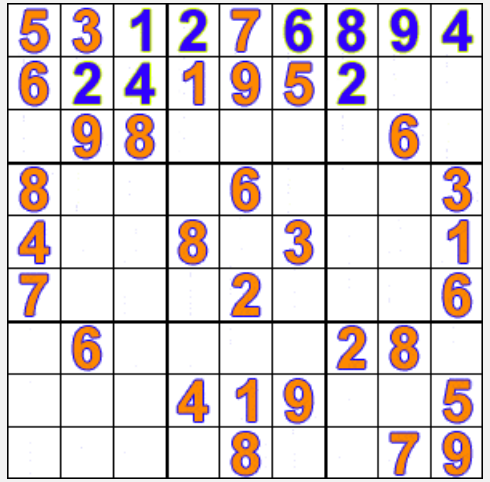
Backtracking

- 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"



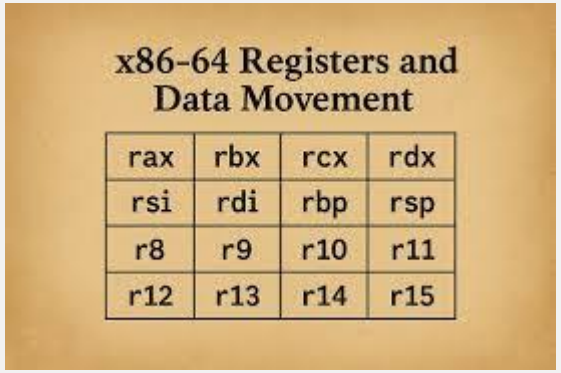
Backtracking

- 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



Backtracking

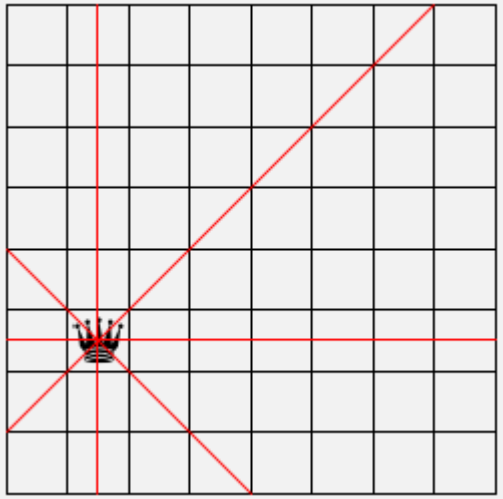
- A **recursive algorithm** for **finding solutions to many computational problems that …**
  - **… tries potential solutions optimistically …** but **“backtracks” when stuck**
  - **Graph algorithms**, e.g., **Path finding**
  - **Optimization**, e.g., **knapsack, “traveling salesman”**
  - **Solving puzzles**, e.g., **Sudoku, n-queens**
  - **In Programming Languages and Compilers!**
    - Register allocation
    - Pattern Matching!



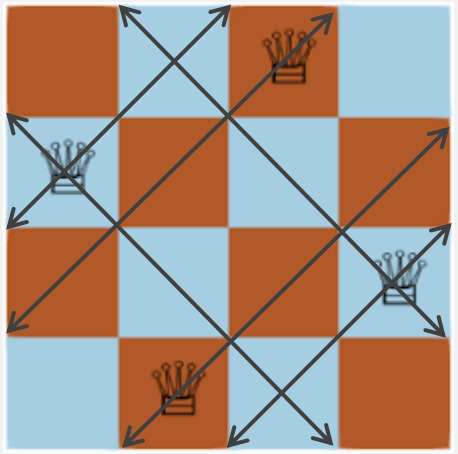

```
;; parse : Expr -> AST
(define (parse s)
  (match s
    [(? hw11-Atom?) (mk-arr s)] ; shortcut,
    [(? symbol?) (vari s)]
    [`(arr ,a) (mk-arr (parse-arraystx a))]
    [`(bind [,x ,e] . ,bods) (bind x (parse
    [`(bind . ,_)
     (raise-syntax-error
      'parse "bind: invalid syntax, expecte
      #:exn exn:fail:syntax:cs450)]
    [`(bind/rec [,x ,e] . ,bods) (recb x (p
    [`(bind/rec . ,_)
     (raise-syntax-error
```

N-Queens problem

- **Place n queens** on an **n x n chess board** so that **no queen "threatens" another …**



All the positions "threatened" by a queen



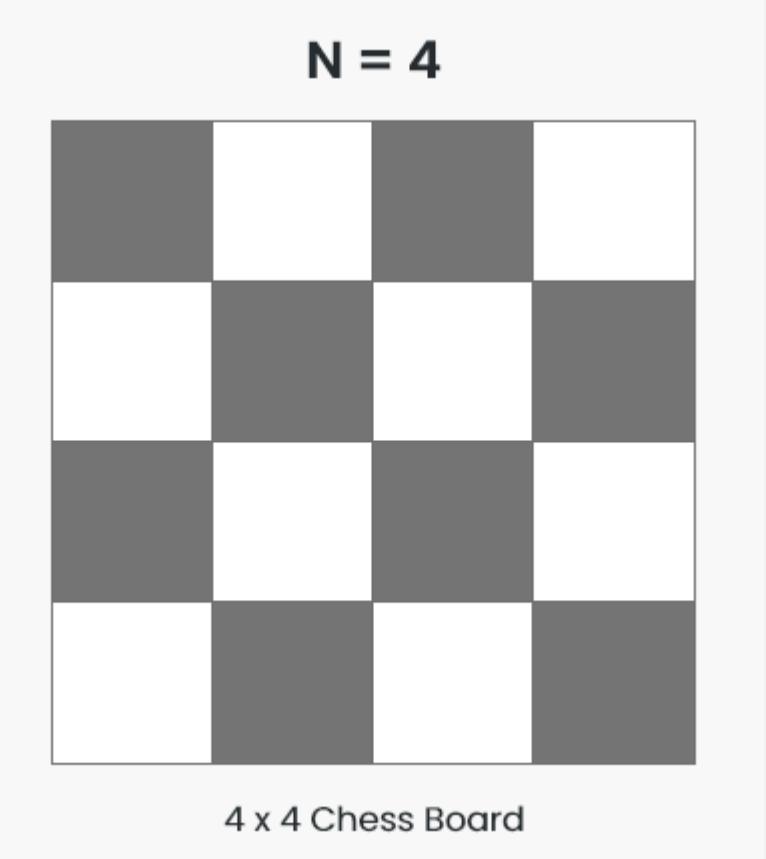
All queens safe

(no one threatens another)

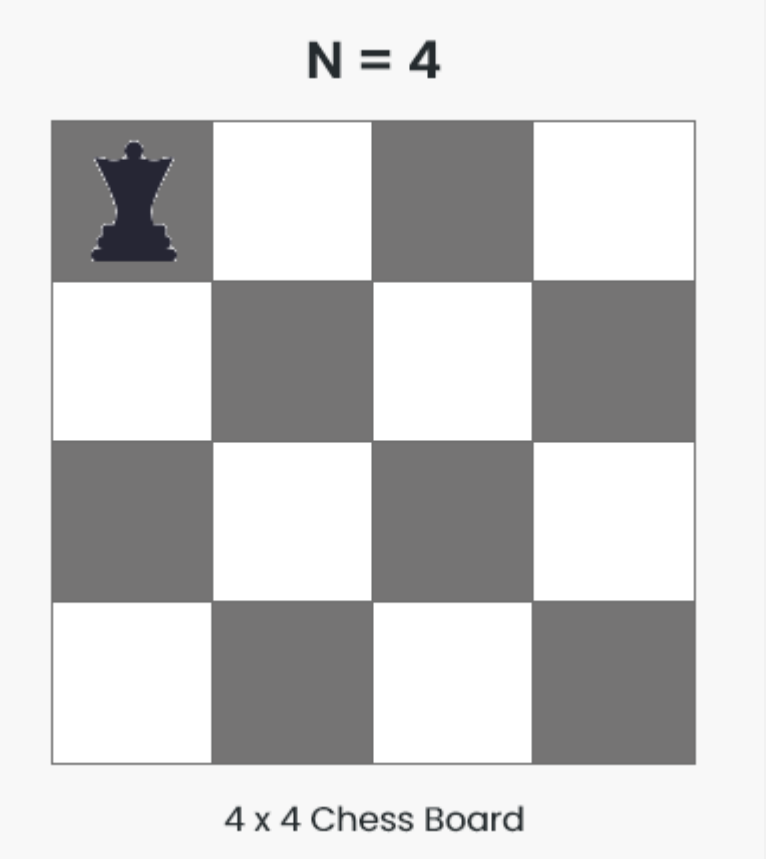
N-Queens problem – solving …

- Place n queens on an n x 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

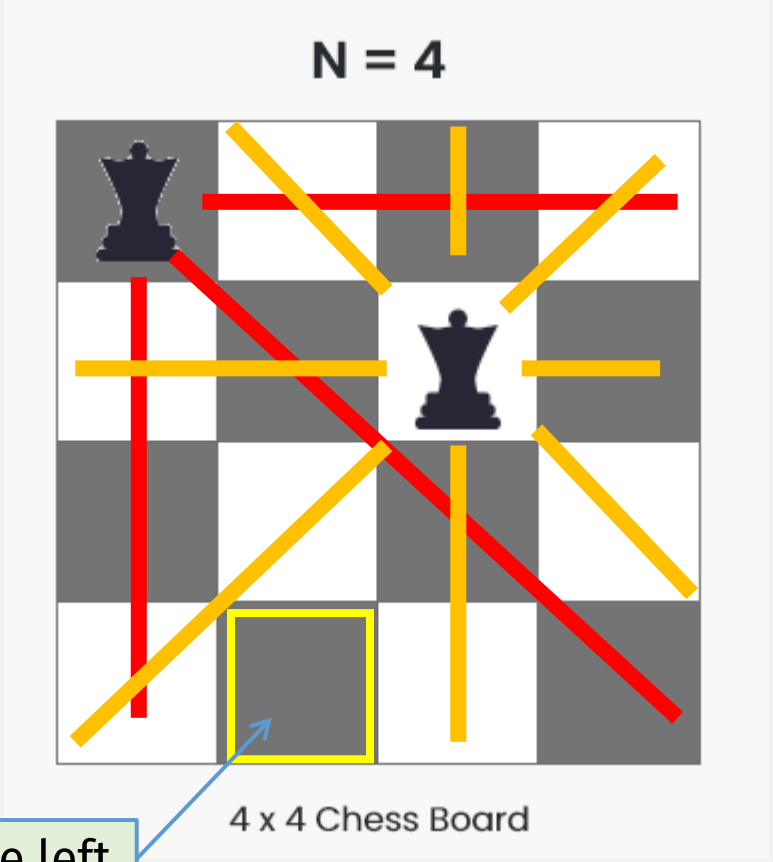


Example: 4-queens



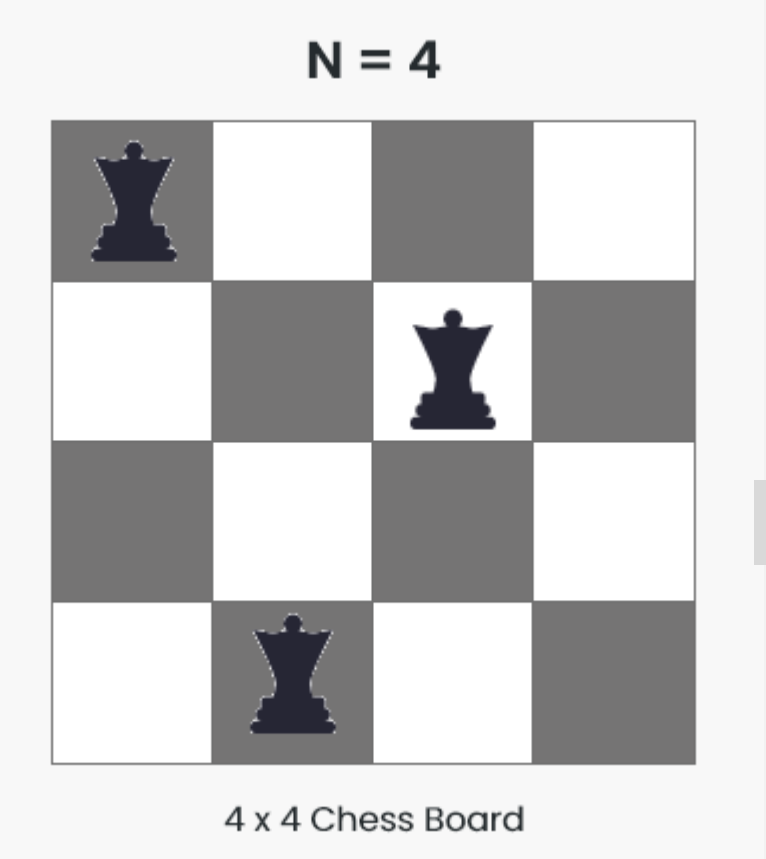
Example: 4-queens

Example: 4-queens



Only 1 non-threatened square left

Example: 4-queens



But … need to place 4 queens!

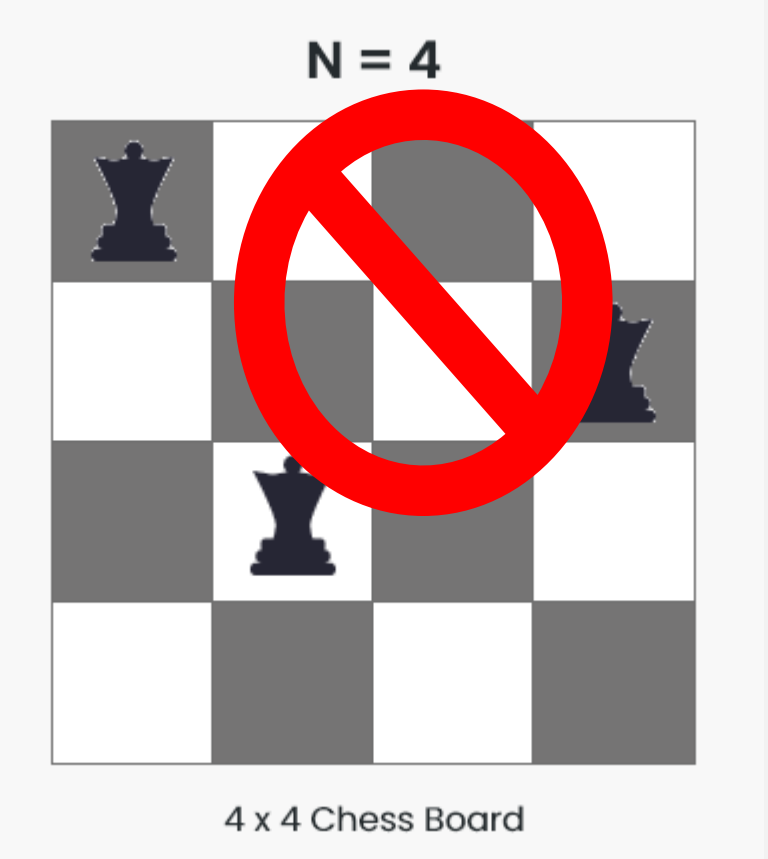
FAIL???

No! we havent tried all solutions …

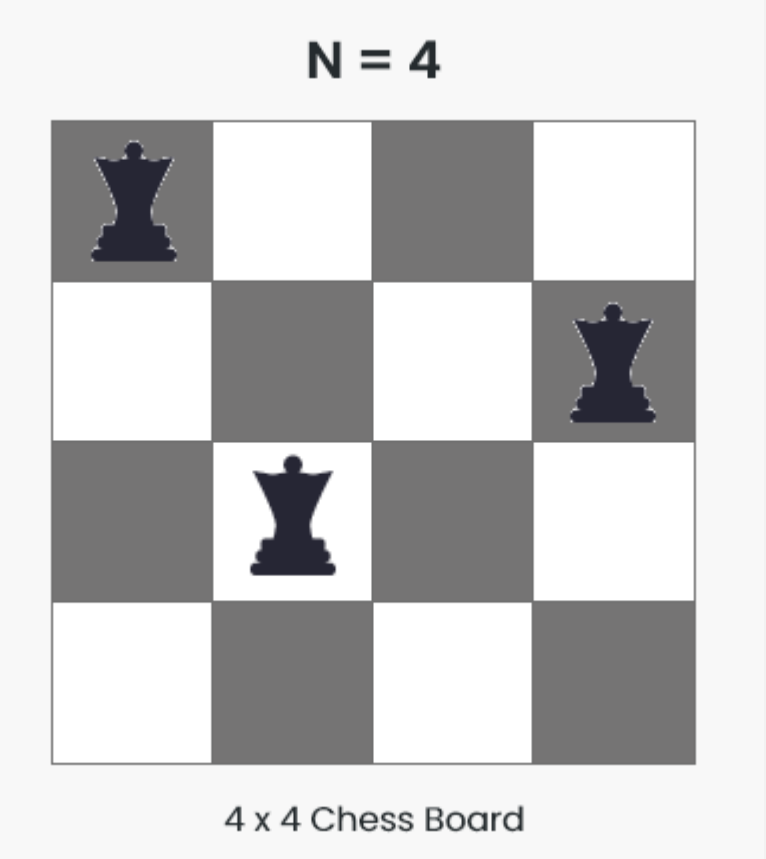
… need to go backwards

Example: 4-queens - Backtracking

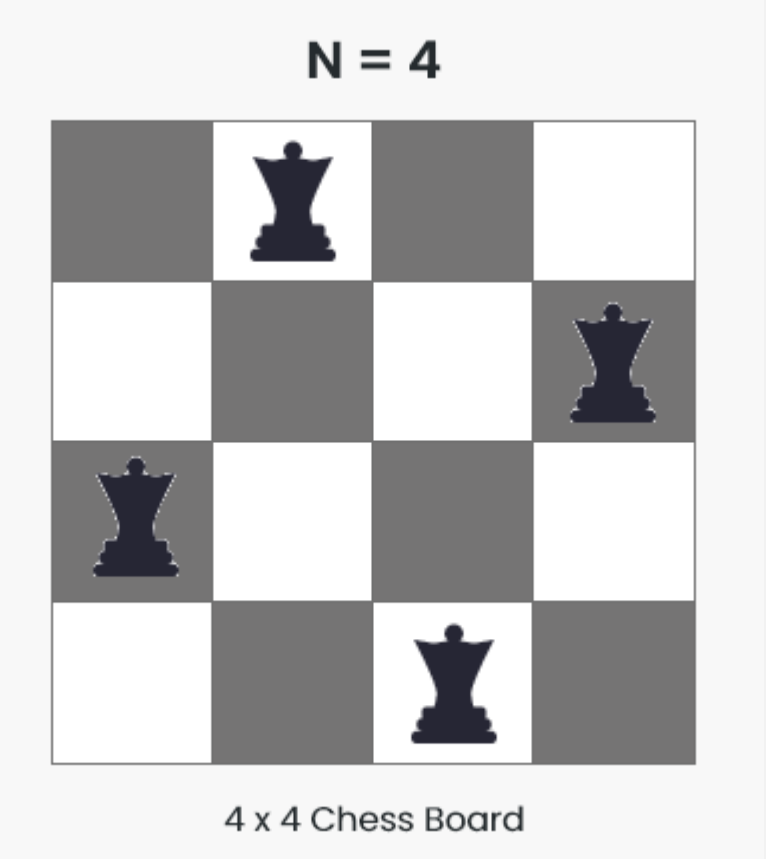
Anywhere to put 4th queen?



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 cu
  (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))]))
```

```
;; A Queen is a
;; … row and column …
```

Produce “false” value to indicate no 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 Maybe<X> is either:
;; - false
;; - X
```

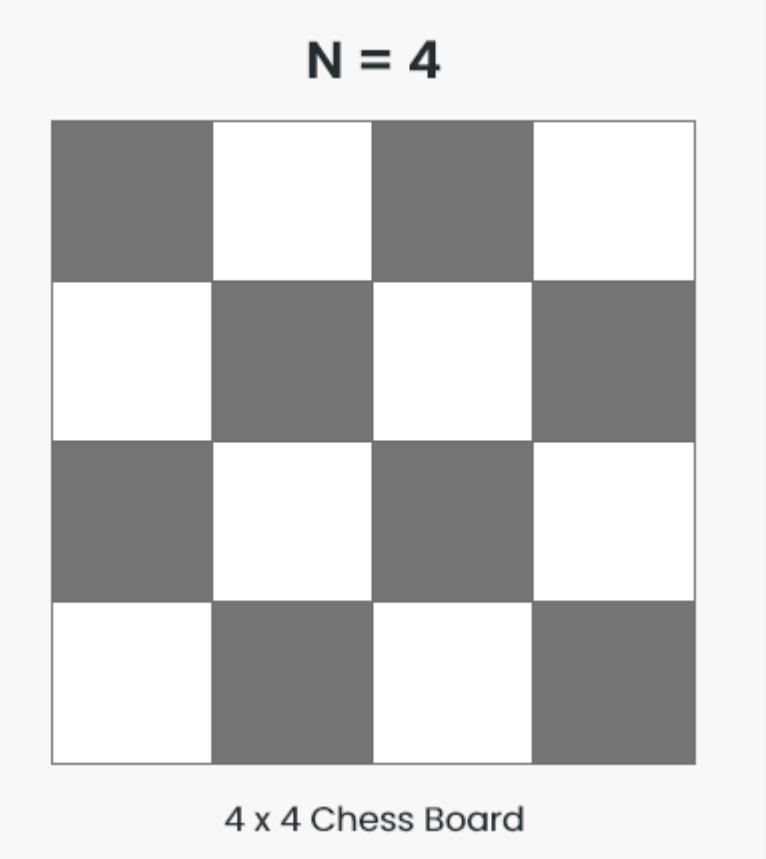
Parameterized Data Def

N-queens Solution Validation

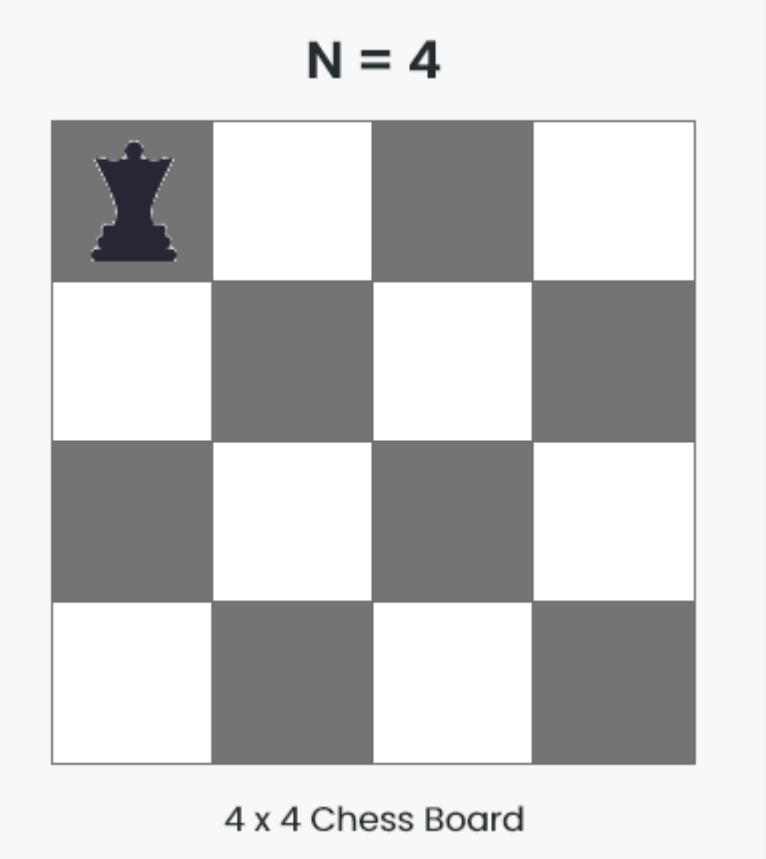
- Still useful to write a **`valid?`** predicate, i.e., for testing
- A “valid” n-queens solution has
  - n (unique) queens
  - No queens threaten any other

```
(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 … ))
```

Example: 4-queens

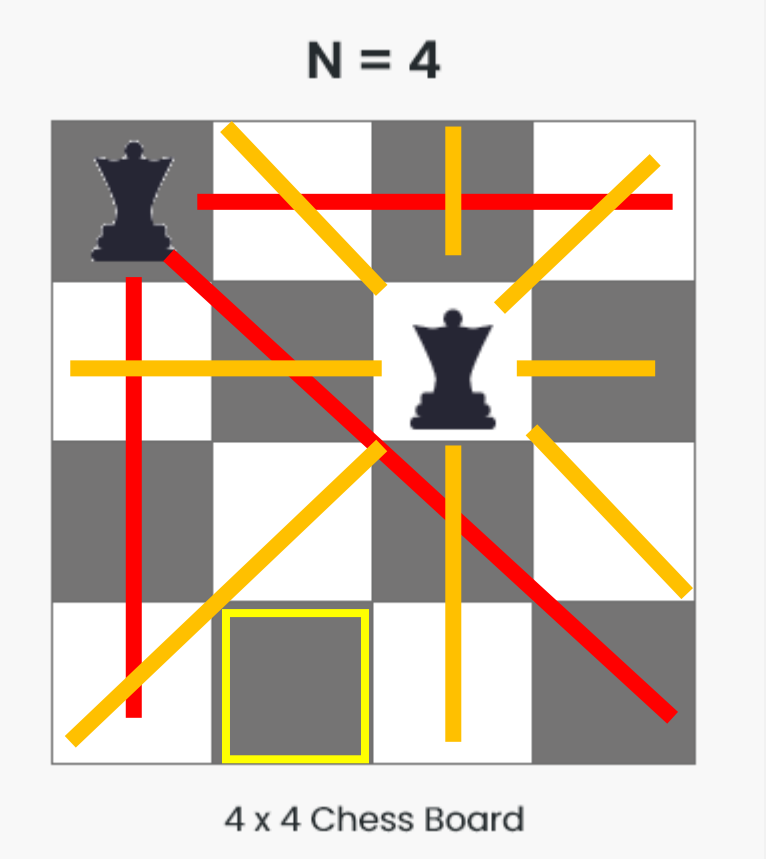


Example: 4-queens

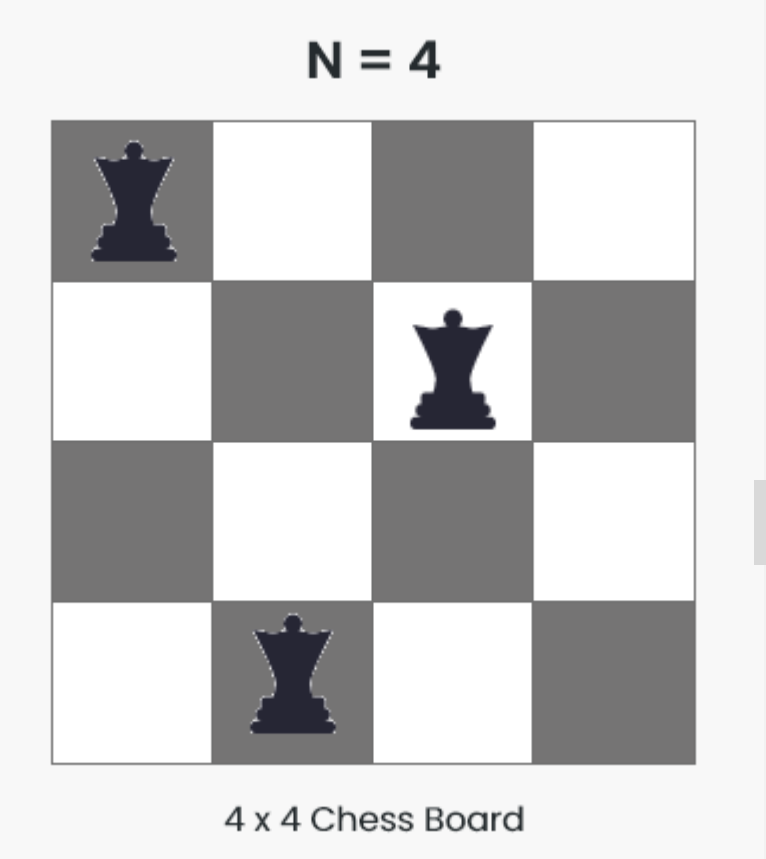


Example: 4-queens

Example: 4-queens



Example: 4-queens



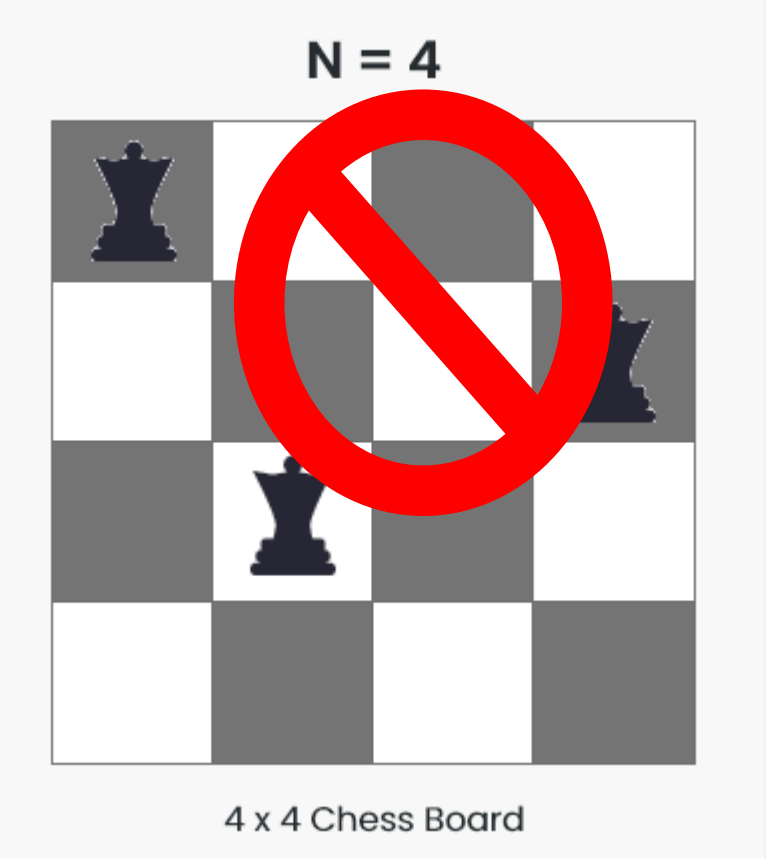
But … need to place 4 queens!

FAIL???

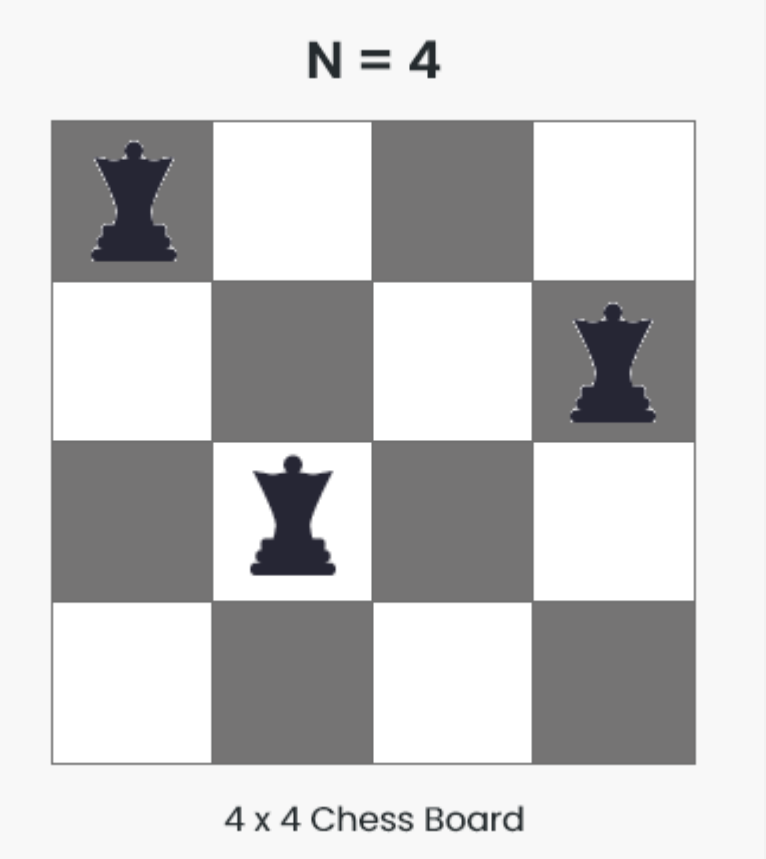
No, we havent tried all solutions …

… need to go backwards

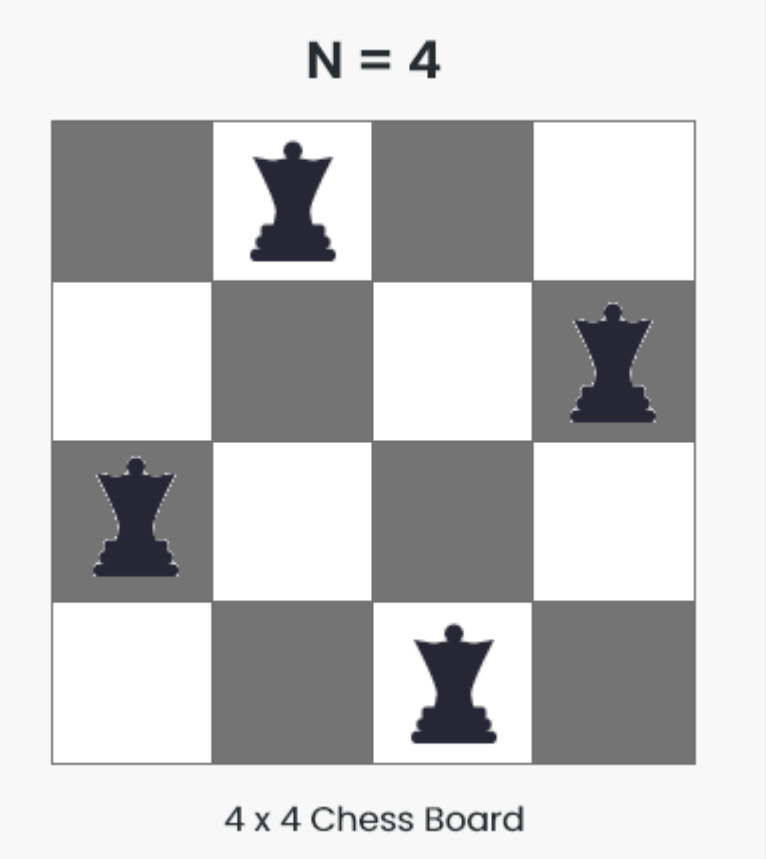
Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



☑

Interlude: Recursion vs Iteration

- **Recursive** functions have a self-reference

```
def factorialUsingRecursion(n):
    if (n == 0):
        return 1;

    # recursion call
    return n * factorialUsingRecursion(n - 1);
```

- **Iterative** code typically use a loop

```
def factorialUsingIteration(n):
    res = 1;

    # using iteration
    for i in range(2, n + 1):
        res *= i;

    return res;
```

Recursion vs Iteration: Which is “Better”?

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least O(n) memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the tradeoffs with your interviewer.

Cracking the Coding Interview, Ch8

r/learnprogramming • 11 yr. ago

[Best Practices] Recursion. Why is it generally avoided and when is it acceptable?

stackoverflow

Are recursive methods always better than iterative methods in Java?

Recursion vs Iteration: Conventional Wisdom

Iteration

Strengths:

- Iteration can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory.
- Iteration is faster and more efficient than recursion.
- It's easier to optimize iterative codes, and they generally have polynomial time complexity.
- They are used to iterate over the elements present in data structures like an array, set, map, etc.
- If the iteration count is known, we can use *for* loops; else, we can use *while* loops, which terminate when the controlling condition becomes false.

Weaknesses:

- In loops, we can go only in one direction, i.e., we can't go or transfer data from the current state to the previous state that has already been executed.
- It's difficult to traverse trees/graphs using loops.
- Only limited information can be passed from one iteration to another, while in recursion, we can pass as many parameters as we need.

Iteration is <u>good</u> with *non-recursive data*

Iteration is <u>bad</u> with *recursive data*!

Recursion better when **accumulators** are needed

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Recursion vs Iteration: Conventional Wisdom

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand. ☑
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- It requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- s difficult to optimize a recursive code, and they generally have higher time complexity an iterative codes due to overlapping subproblems.

Recursion better when **accumulators** are needed

Use recursion with *recursive data*!

Recursion is slow
Recursion is slow
Recursion is slow
Recursion is slow

Investigate:
Is recursion is slower??

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Recursion vs Iteration: In Racket

Racket Recursion

```
;; sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (sum-to x)
  (if (zero? x)
      x
      (+ x (sum-to (sub1 x)))))
```

Conclusion?
Recursion is slower?

WAIT!
Racket does not have “for” loops

```
(define BIG-NUMBER 999999)

(time (sum-to BIG-NUMBER))
; cpu time: 202 real time: 201 gc time: 156
```

Racket “Iteration”

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Recursion vs Iteration: In Racket

Conclusion?
Recursion is not slower than iteration?

equivalent

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in Racket is just a macro (i.e., "syntactic sugar") for a recursive function

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Tail Calls

From Wikipedia, the free encyclopedia

In computer science, a **tail call** is a subroutine call performed as the final action of a procedure. If the target of a tail is the same subroutine, the subroutine is said to be **tail recursive**, which is a special case of direct recursion. **Tail recursion** (or **tail-end recursion**) is particularly useful, and is often easy to optimize in implementations.

Tail calls can be implemented without adding a new stack frame to the call stack.

Fixed?

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least O(n) memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can

Recursion vs Iteration: In Racket

Conclusion?
Recursion is not slower than iteration?

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

Tail-recursive function

Tail-call (does not add to stack)

(Tail) recursion is iteration!

Recursion vs Iteration: Under the Hood

- It makes sense that recursion and iteration are equivalent …
  - Recursive call compiles to:
    - **JUMP** instruction
  - Loop compiles to:
    - **JUMP** instruction!

- … except in languages that make them not equivalent!
  - i.e., languages that push extra stack frames that are not needed

Tail-Calls in Other Languages

- Most functional languages (Racket, Haskell, Erlang, F#) implement proper tail calls
  (no extra stack frame)

- Some languages require an explicit annotation
  - Clojure: `recur`
  - Scala: `@tailrec`

- Some languages (JavaScript) have it (ECMAScript 6), but don't have it

- Most imperative languages don't properly implement tail calls
  (they add an unnecessary stack frame)
  - Python, Java, C#, Go

Guido Got It Backwards

Wednesday, April 22, 2009

Tail Recursion Elimination

I recently posted an entry in my Python History blog on the origins of Python's functional features. A side remark about not supporting tail recursion elimination (TRE) immediately sparked several comments about what a pity it is that Python doesn't do this, including links to recent blog entries by others trying to "prove" that TRE can be added to Python easily. So let me defend my position (which is that I don't *want* TRE in the language). If you want a short answer, it's simply unpythonic. Here's the long answer:

First, as one commenter remarked, TRE is incompatible with nice stack traces: when a tail recursion is eliminated, there's no stack frame left to use to print a traceback when something goes wrong later. This will confuse
cently wrote something recursive (the recursion isn't obviou
ce printed), and makes debugging hard. Providing an optio
eems wrong to me: Python's default is and should always b
be maximally helpful for debugging. This also brings me to the next issu

About Me

Guido van Rossum

Python's BDFL

View my complete profile

Blog Archive

► 2022 (2)

► 2019 (1)

Wrong!

Equivalent to saying:
"every **for** loop iteration should push a stack frame!"

Proper tail calls is about eliminating stack frames that shouldn't be there in the first place! (because it's just iteration!)

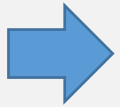
Non Tail Call

C

```c
// Program to find factorial of a number n modulo prime
int factorial(int n, int prime)
{
    if (n <= 1) {
        // base case
        return 1;
    }

    return (n * factorial(n - 1, prime) % prime) % prime;
}
```

Non-tail-call

Slower, more memory



Compiler output: **godbolt.org**

x86-64 gcc 14.2 -O3

```asm
1   factorial:
2     cmp edi, 1
3     jle .L16
4     push r15
5     mov eax, 2
6     push r14
7     push r13
8     push r12
9     lea r12d, [rdi-1]
10    push rbx
11    mov ebx, edi
12    sub rsp, 16
13    cmp edi, 2
14    je .L3
15    lea r13d, [rdi-2]
16    cmp edi, 3
17    je .L4
18    lea r15d, [rdi-3]
19    cmp edi, 4
20    je .L5
21    lea r14d, [rdi-4]
22    cmp edi, 5
23    je .L6
24    lea edi, [rdi-5]
25    mov DWORD PTR [rsp+12], esi
26    call factorial
27    mov esi, DWORD PTR [rsp+12]
28    imul eax, r14d
29    cdq
30    idiv esi
31    mov eax, edx
32    imul eax, r15d
33  .L6:
34    cdq
35    idiv esi
36    mov eax, edx
37    imul eax, r13d
38  .L5:
39    cdq
40    idiv esi
41    mov eax, edx
42    imul eax, r12d
43  .L4:
44    cdq
45    idiv esi
```

Stack push

https://www.geeksforgeeks.org/tail-call-optimisation-in-c/

Tail Calls as Loops

Compiler output: **godbolt.org**

x86-64 gcc 14.2 | -O3

c

```c
// C program to illustrate Tail Call Optimisation
int factorial(int store, int num, int prime) {
  if (num < 1) {
    // Base case
    return store;
  }

  return factorial((store%prime * num%prime)%prime, num - 1, prime);
}
```

Tail call – compiled directly to **for** loop!

Faster, less memory

```asm
factorial(int, int, int):
  mov ecx, edx
  test esi, esi
  jle .L5
.L2:
  mov eax, edi
  cdq
  idiv ecx
  mov eax, edx
  imul eax, esi
  cdq
  idiv ecx
  mov edi, edx
  sub esi, 1
  jne .L2
.L5:
  mov eax, edi
  ret
```

No stack push!

```c
// this function calculates factorial modulo prime
int factorial(int num, int prime) {
    int store = 1;
    for (int i = num; i > 0; i--) {
        store = (store%prime * i%prime)%prime;
    }
  return store;
}
```

for loop

```asm
factorial(int, int):
  mov edx, 1
  test edi, edi
  jle .L1
.L3:
  mov eax, edx
  cdq
  idiv esi
  mov eax, edx
  imul eax, edi
  cdq
  idiv esi
  sub edi, 1
  jne .L3
.L1:
  mov eax, edx
  ret
```

Recursion is same as **for** loop!

Some languages (with –O3 optimization) **directly compile recursion to a loop!** (because they are equivalent!)

Proper Tail Calls in JavaScript

Proper Tail Calls (PTC) is a new feature in the ECMAScript 6 language. This feature was added to facilitate recursive programming patterns, both for direct and indirect recursion. Various other design patterns can benefit from PTC as well, such as code that wraps some functionality where the wrapping code directly returns the result of what it wraps. Through the use of PTC, the amount of memory needed to run code is reduced. In deeply recursive code, PTC enables code to run that would otherwise throw a stack overflow exception.

https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/

Feature name

proper tail calls (tail call optimisation)

Not supported in V8 (Chrome) or SpiderMonkey (Firefox) or NodeJS!

https://compat-table.github.io/compat-table/es6/

There's hope!

WebAssembly tail calls

Published 06 April 2023 · Tagged with WebAssembly

We are shipping WebAssembly tail calls in V8 v11.2! In this post we give a brief overview of this proposal, demonstrate an interesting use case for C++ coroutines with Emscripten, and show how V8 handles tail

https://v8.dev/blog/wasm-tail-call

Recursion vs Iteration: Conclusion

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand.
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- t requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- t is difficult to optimize a recursive code, and they generally have higher time complexity than iterative codes due to overlapping subproblems.

Recursion is (usually) easier to read

Recursion better when **accumulators** are needed

Use recursion with *recursive data*!

Recursion is slower …

… in languages that choose to make it slower!

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Interlude

Recursion vs Iteration: In Racket

Conclusion?
Recursion is <u>not</u> slower than iteration?

equivalent

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in Racket is just
a macro (i.e., "syntactic sugar")
for a (tail) recursive function

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Racket **for** expressions

Generic "sequence"
(number, most data structures ...)

```
(for/list ([x lst]) (add1 x))
```

↓

```
(map add1 lst)
```

```
(for/list ([x n]) (add1 x))
```

↓

```
(build-list n add1)
```

```
(for/list ([x lst] #:when (odd? x)) (add1 x))
```

↓

```
(filter odd? (map add1 lst))
```

```
(for/sum ([x lst] #:when (odd? x)) (add1 x))
```

↓

```
(foldl + 0 (filter odd? (map add1 lst)))
```

Note:
These are still expressions!

Lots of variations!
(see docs)

Racket **for*** expressions

"nested" `for` loops

```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```

```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

```
(for*/list (for
(for*/lists (id
  body-or-break
(for*/vector ma
(for*/hash (for
(for*/hasheq (f
(for*/hasheqv (
(for*/hashalw (
(for*/and (for-
(for*/or (for-c
(for*/sum (for-
(for*/product (
(for*/first (fo
(for*/last (for
(for*/fold ([ac
  body-or-break
(for*/foldr ([a
            (for
```

Lots of variations! (see docs)