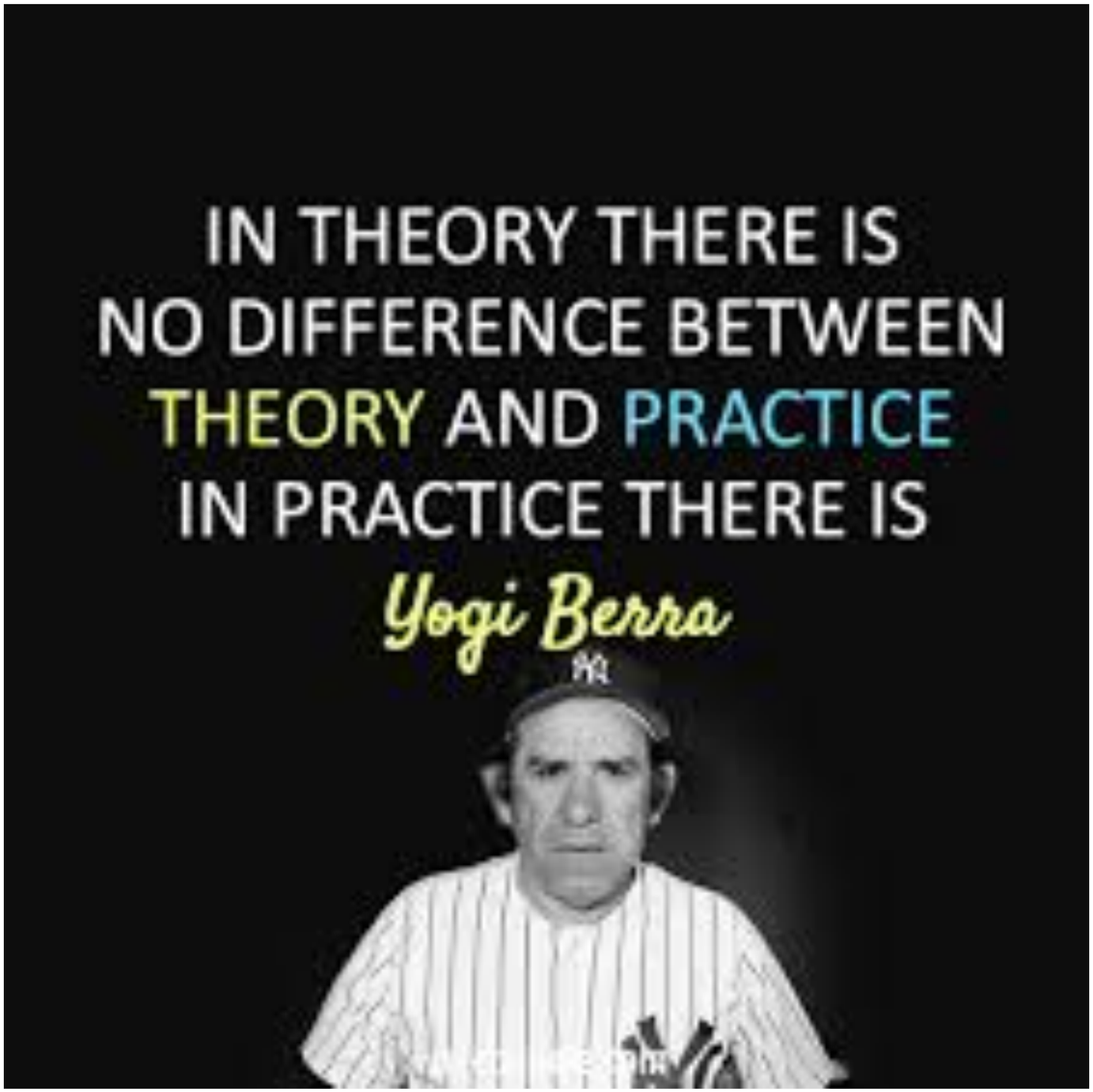
15-150 Fall 2020

Stephen Brookes

Lecture 13

Backtracking with continuations



correct **code may not be** ***fast*****…**
Yogi wants both!

case study

- **Direct**- and **continuation**-style programming
    - two different ways to solve the *same* problem
- Benefits and disadvantages…
    - taking advantage of math and logic
      to ensure ***correctness*** and assess ***efficiency***

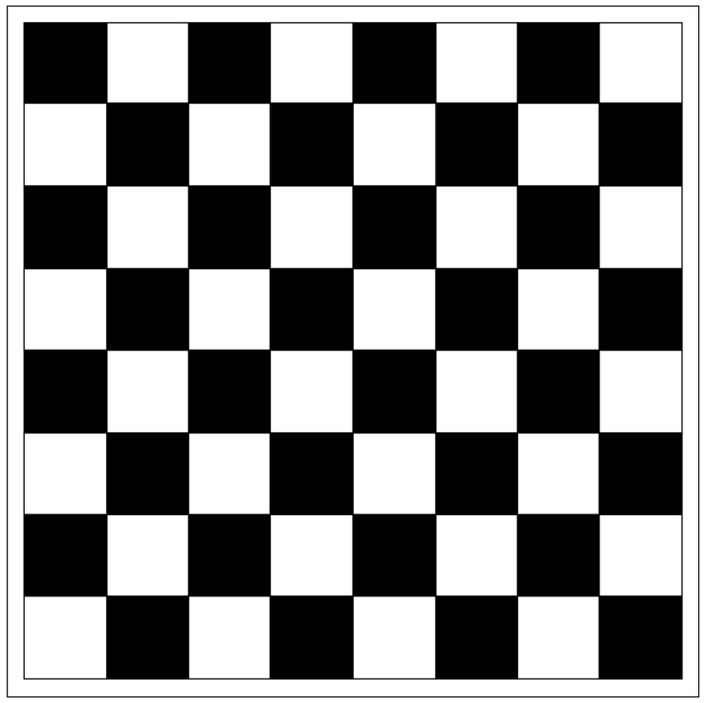
the bishops problem

the bishops problem

the bishops problem

the bishops problem

- Put m bishops onto an n-by-n chessboard *safely*.
    - bishops *attack* along diagonals
    - *safe* = each bishop is **attacked by at most one other**



the bishops problem

- Find a way to put m bishops
  onto an n-by-n chessboard *safely*, if possible.

bishops : int -> int -> (int * int) list option

bishops n m = SOME L
where L is a safe placement for m bishops
on an n-by-n chessboard, if possible

bishops n m = NONE
otherwise

the most bishops

- What's the ***largest*** number of bishops
  that can be safely placed on an n-by-n chessboard?

most_bishops : int -> int

most_bishops n = m,

where m is
the largest number of bishops
that can safely be placed
on the n-by-n chessboard

basic types

```
type pos = int * int

type sol = pos list

type state = sol * pos list

type ans = sol option
```

A position
is a cell or grid square
(x,y)

A (partial) solution
is a list of positions

A *state* is
a (partial) solution
and a list of remaining positions

An *answer* is
SOME solution
or
NONE

Warning: safety not built in!

board

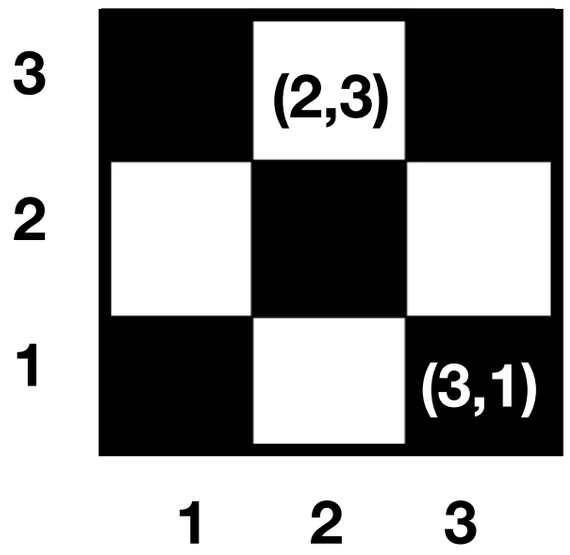
upto : int -> int -> int list
cart : int list * int list -> pos list
board : int -> pos list

```
fun upto i j = if i>j then [ ] else i :: upto (i+1) j

fun cart ([ ], B) = [ ]
|     cart (a::A, B) = map (fn b => (a, b)) B @ cart (A, B)

fun board n = let val xs = upto 1 n in cart (xs, xs) end
```

board 8 represents the 8-by-8 chessboard

example

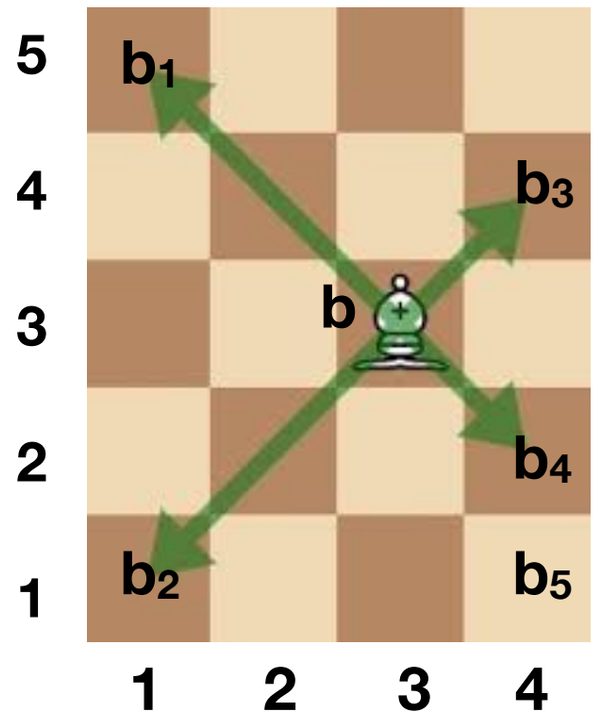
```
- board 3;
val it = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
  : (int * int) list
```



counting threats

threats : pos * pos -> int
attacks : pos list -> pos -> int

```
fun threats ((x, y), (i, j)) = if abs(x-i) = abs(y-j) then 1 else 0

fun attacks bs b = foldr (op +) 0 (map (fn p => threats (p, b)) bs)
```

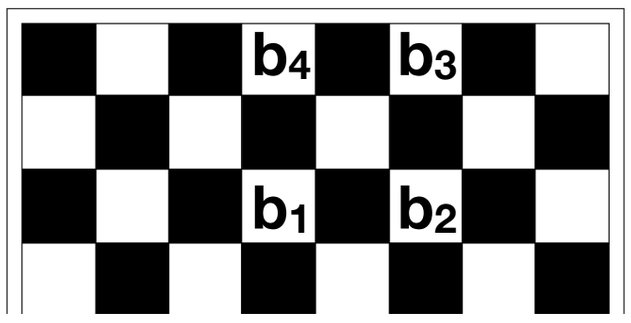


attacks [b_1,…,b_5] b = 4

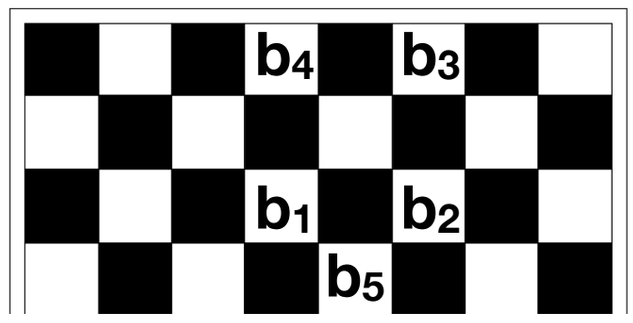
safe

forall : (pos -> bool) -> pos list -> bool
safe : pos list -> bool

```
fun forall p = foldr (fn (x, t) => (p x) andalso t) true

fun safe bs = forall (fn b => (attacks bs b <= 2)) bs
```



safe



unsafe:

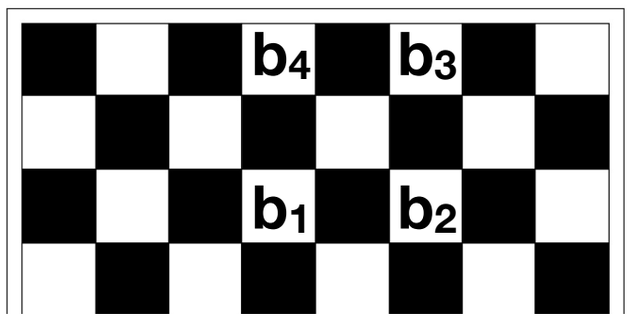
b_5 threatened by b_1 and b_2

safe

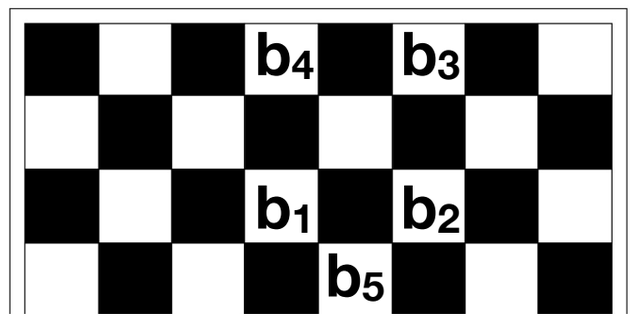
forall : (pos -> bool) -> pos list -> bool
safe : pos list -> bool

```
fun forall p = foldr (fn (x, t) => (p x) andalso t) true

fun safe bs = forall (fn b => (attacks bs b <= 2)) bs
```

why 2?



safe



unsafe:

b_5 threatened by b_1 and b_2

safe spec

safe : pos list -> bool

ENSURES

safe bs = true,

if each cell in bs is attacked by *at most one other*

safe bs = false,

otherwise

direct-style design

- Start with the obvious initial state
  (empty partial solution, all squares are candidates)

- Use a helper function that finds a list of
  the safe ways to *extend* a state with one more bishop

- A *searching* function that
  maintains a list of candidate states to be explored,
  and looks for a *satisfactory* solution
  reachable from one of these states

“depth-first search”

steps

init : int -> state
del : pos -> pos list -> pos list
steps : state -> state list

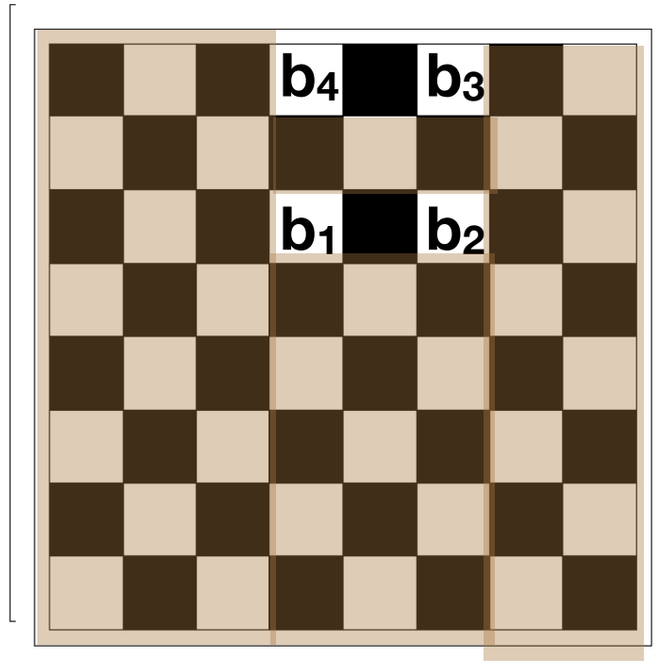
```
fun init n = ([ ], board n)


fun del b [ ]     = [ ]
|     del b (c::cs) = if b=c then cs else c::(del b cs)


fun steps (bs, rest) =
let
   val R = List.filter (fn b => safe(b::bs)) rest
in
   map (fn b => (b::bs, del b rest)) R
end
```

steps : state -> state list

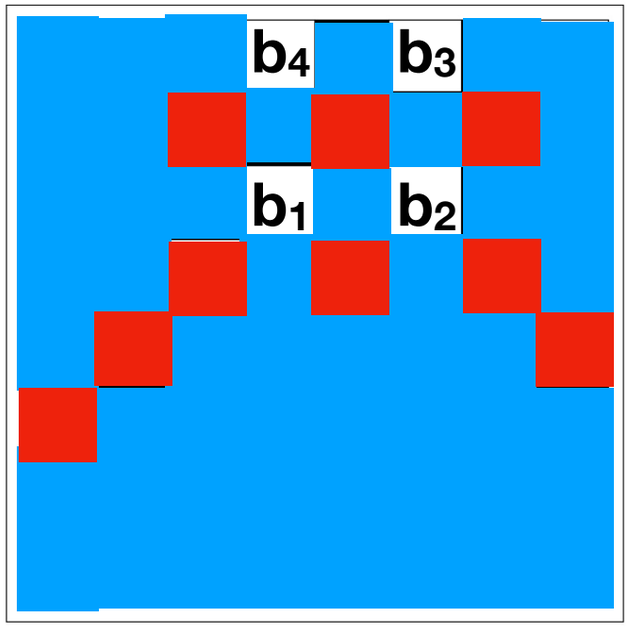
steps (bs, rest) =
a list of all safe ways to extend bs with a bishop from rest.

Each element of this list is a state (b::bs, del b rest),
where b is in rest and safe (b::bs) = true.

([b_1,b_2,b_3,b_4], rest)



steps ([b_1,b_2,b_3,b_4], rest) has length **51**



ok

not ok

valid states

- A state (bs, rest) is ***valid*** iff safe(bs) = **true**

- All states generated from init n using steps are valid

  - init n is a valid state

  - When (bs, rest) is valid,
    steps (bs, rest) returns a list of valid states.

search

: (sol -> bool) -> state list -> sol option

```
fun search p [ ] = NONE
|     search p ((bs, rest)::states) =
   if (p bs) then SOME bs
             else search p (steps (bs, rest) @ states)
```

"depth-first search"

REQUIRES L is a list of valid states

search p L = SOME bs
where bs is a safe list satisfying p
reachable from a state in L,
if there is one

search p L = NONE
otherwise

```
type sol = pos list
```

bishops

bishops : int -> int -> sol option

```
fun bishops n m =
    search (fn bs => (length bs = m)) [init n]
```

length m, reachable from ([ ], board n)

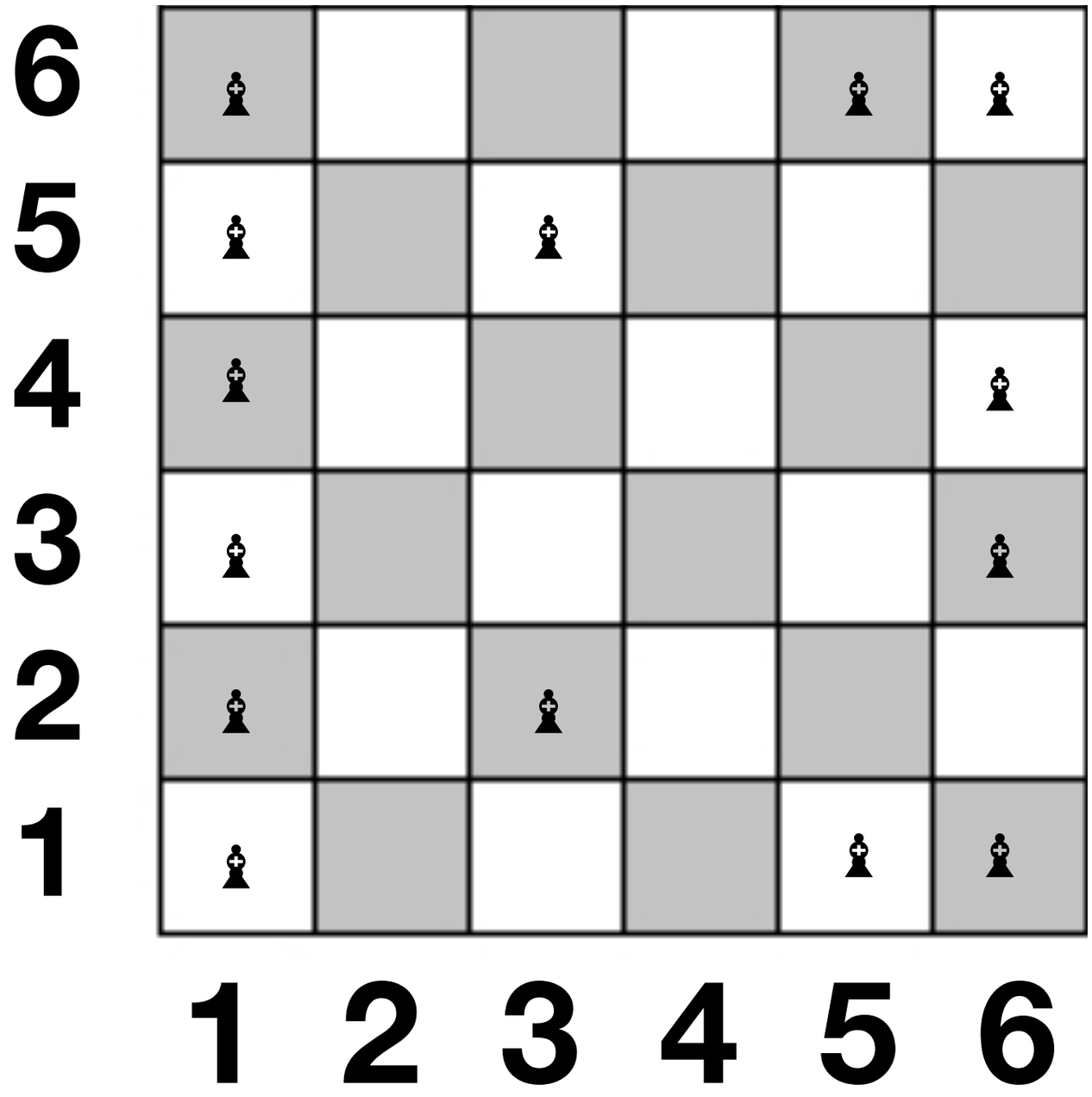
bishops n m = SOME bs,
where bs is a safe placement of m bishops
on the n-by-n board,
if there is one

bishops n m = NONE,
if there is no safe placement of m bishops
on the n-by-n board

```
- bishops 6 14;
val it =
  SOME
    [(6,6),(6,4),(6,3),(6,1),(5,6),(5,1),(3,5),
     (3,2),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
```

14 bishops
safely
on 6-by-6 board

```
- bishops 6 14;
val it =
  SOME
    [(6,6),(6,4),(6,3),(6,1),(5,6),(5,1),(3,5),
     (3,2),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
```



14 bishops safely on 6-by-6 board

more examples

```
-bishops 6 15;
    VERY SLOW…

-bishops 8 20;
    VERY SLOW….. (I gave up!)
```

the most bishops

most_bishops : int -> int

```
fun most_bishops n =
  let
     fun loop m =
         (print ( "Trying " ^ Int.toString m ^ "\n");
          case (bishops n m) of
                SOME _ => loop (m+1)
              | NONE   => m-1)
in
   loop 1
end
```

example

```
- most_bishops 6;
Trying 1
Trying 2
Trying 3
Trying 4
Trying 5
Trying 6
Trying 7
Trying 8
Trying 9
Trying 10
Trying 11
Trying 12
Trying 13
Trying 14
Trying 15

TAKING FOREVER…
```

The direct-style searcher is VERY SLOW

diagnosis

The search function does a lot of list-building and safety checking….

search p ((bs, rest)::states)

may call search p (steps (bs, rest) @ states)

- The number of candidate states grows
- steps (bs, rest) calls safe (b::bs) for each b in rest

In example, **steps ([b_1,b_2,b_3,b_4], rest)** has length **51**

The work for safe L is O ((length L)^2)

a cps design

- Avoid enumerating lists of states

- Work with a single “current” safe state,
  look for ***any*** safe solution *extending* that state
  - only check for safety of the *current* state
  - on *success*, call a ***success continuation***
    with the (complete) solution
  - on failure, call a ***failure continuation***
    to *backtrack* and try ***other*** extensions

solver

solver : (sol -> bool) -> state -> (sol -> ans) -> (unit -> ans) -> ans

solver p (bs, rest) s k

- p : sol -> bool criterion for “success"
- bs : sol current (partial) solution
- rest : pos list remaining board cells
- s : sol -> ans to be applied on “success”
- k : unit -> ans to be used on “failure”

type sol = pos list
type ans = sol option

solver spec

REQUIRES p total**,** bs safe

ENSURES

solver p (bs, rest) s k

= s(L),
where L is a solution, satisfying p,
extending bs with bishops from rest,
if there is one

= k( ), otherwise

a safe placement

In each case, we get a result of type ans

solver

solver : (sol -> bool) -> state -> (sol -> ans) -> (unit -> ans) -> ans

```
fun solver p (bs, rest) s k =
  if p(bs) then s(bs) else
     case rest of
       [ ]   => k( )
  | b::cs  => if     safe (b::bs)
                then solver p (b::bs, cs) s (fn ( ) => solver p (bs, cs) s k)
                else   solver p (bs, cs) s k
```

backtracking

```
solver p (bs, b::cs) s k =>*
        if safe (b::bs) then
            solver p (b::bs, cs) s (fn ( ) => solver p (bs, cs) s k)
                                …
```

- If b::bs is safe,
  but cannot be extended to success using cs,
  the failure continuation triggers solver p (bs, cs) s k

bishops

bishops : int -> int -> sol option

```
fun bishops n m =
    solver (fn bs => length bs = m) (init n) SOME (fn ( ) => NONE)
```

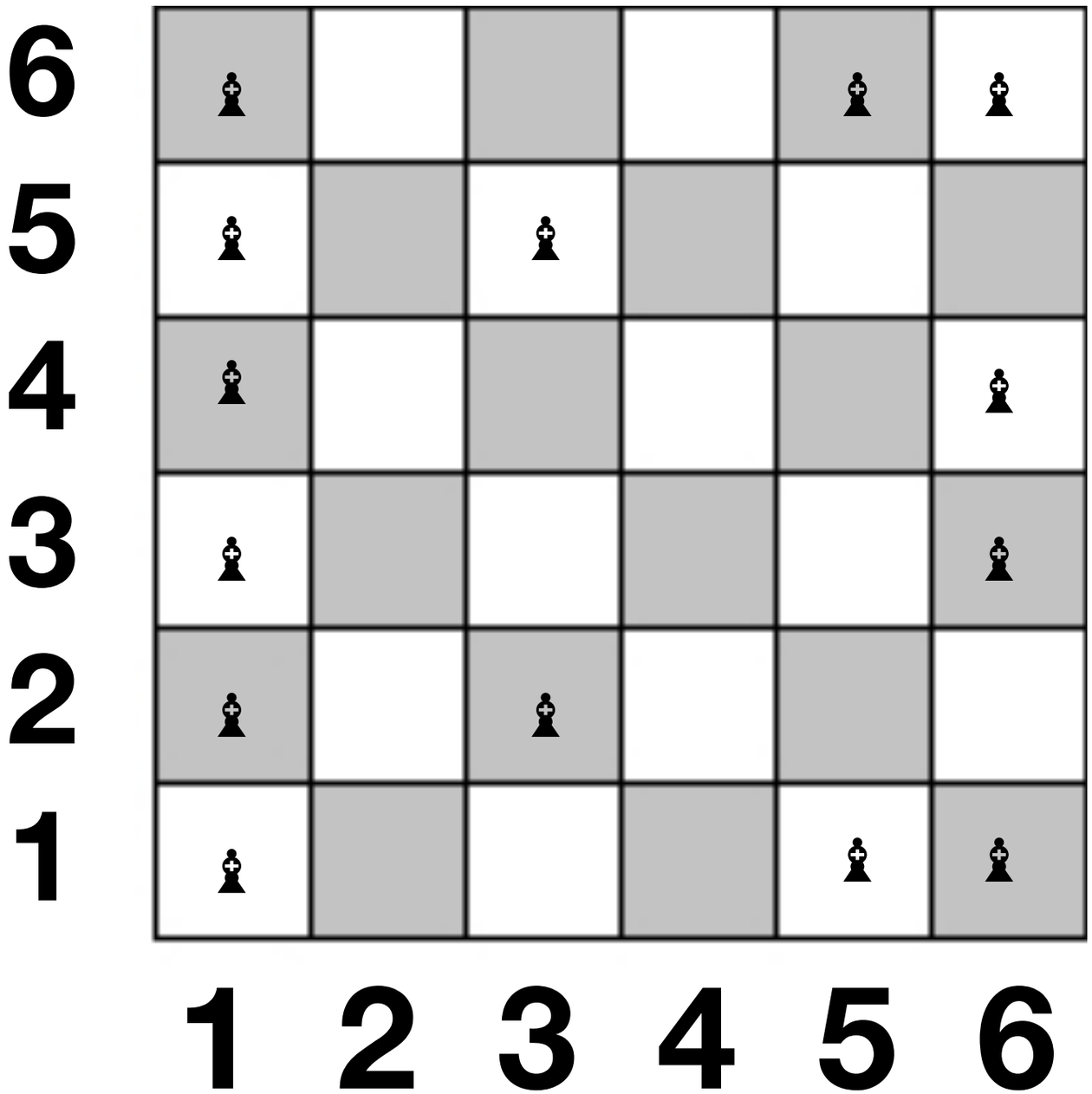
length m,
reachable from ([ ], board n)

bishops n m = SOME bs,
where bs is a safe placement of m bishops
on the n-by-n board,
if there is one

bishops n m = NONE,
if there is no safe placement of m bishops
on the n-by-n board

```
- bishops 6 14;
val it =
  SOME
    [(6,6),(6,4),(6,3),(6,1),(5,6),(5,1),(3,5),
     (3,2),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
```

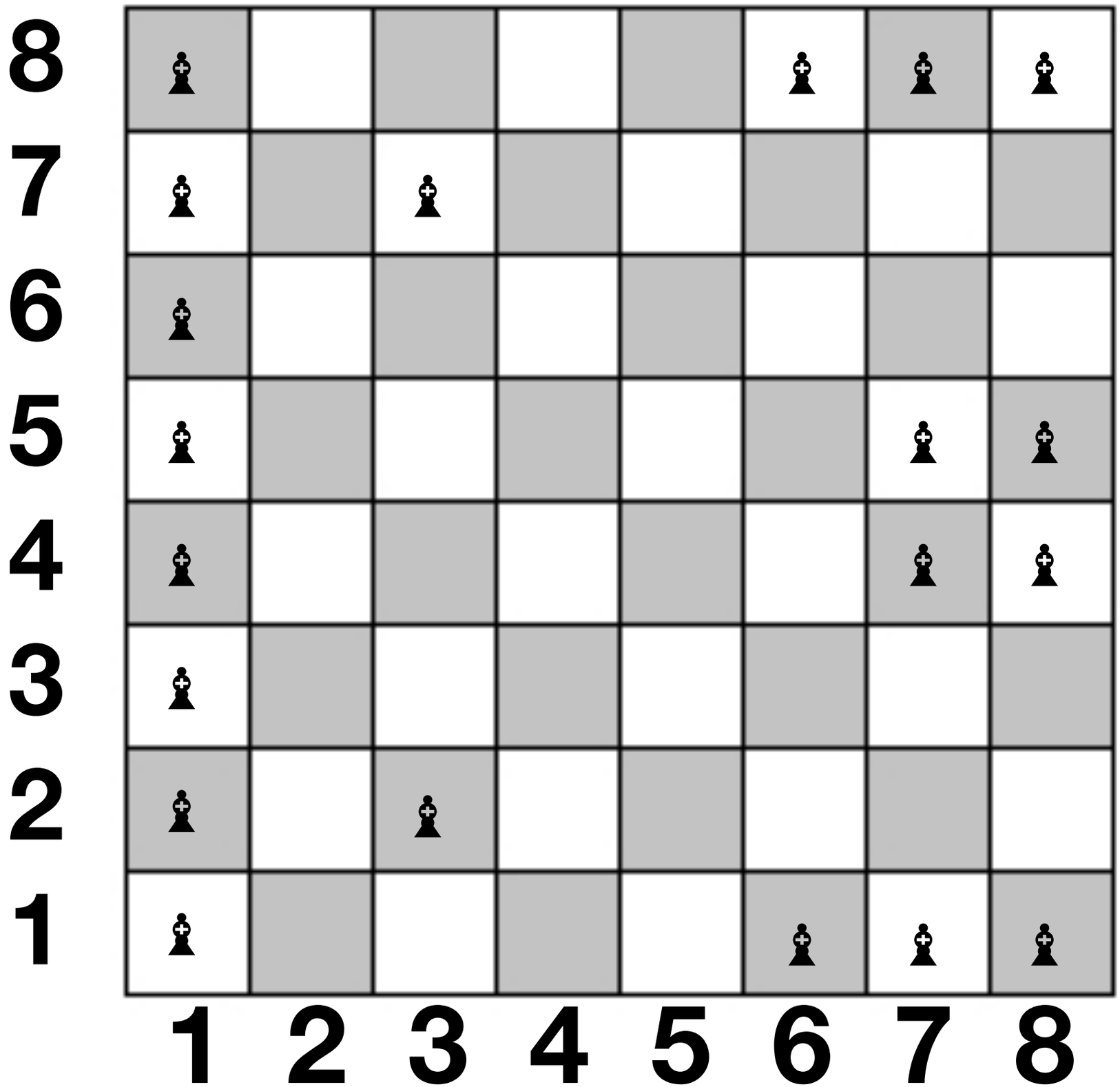
FAST!



14 bishops safely on 6-by-6 board

```
- bishops 8 20;
val it =
  SOME
    [(8,8),(8,5),(8,4),(8,1),(7,8),(7,5),(7,4),(7,1),(6,8),(6,1),
     (3,7),(3,2),(1,8),(1,7),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
```

FAST!



20 bishops safely on 8-by-8 board

n=10

10	X	X					X	X		X
9	X			X						
8	X									
7	X									
6	X								X	X
5	X								X	X
4	X									
3	X									
2	X			X						
1	X	X					X	X		X
	1	2	3	4	5	6	7	8	9	10

24 bishops safely on 10-by-10 board

n=12

	1	2	3	4	5	6	7	8	9	10	11	12
12	X					X				X	X	X
11	X		X	X								
10	X											
9	X											X
8	X										X	X
7	X											
6	X											
5	X										X	X
4	X											X
3	X											
2	X		X	X								
1	X					X				X	X	X

30 bishops safely on 12-by-12 board

most_bishops

most_bishops : int -> int

```
fun most_bishops n =
  let
      fun loop m =
          (print ( "Trying " ^ Int.toString m ^ "\n");
           case (bishops n m) of
                SOME _ => loop(m+1)
              | NONE   => m-1)
in
   loop 1
end
```

results

```
- most_bishops 6;
Trying 1
Trying 2
Trying 3
Trying 4
Trying 5
Trying 6
Trying 7
Trying 8
Trying 9
Trying 10
Trying 11
Trying 12
Trying 13
Trying 14
Trying 15
val it = 14 : int
```

FAST!!!!

```
- most_bishops 8;
Trying 1
Trying 2
Trying 3
Trying 4
Trying 5
Trying 6
Trying 7
Trying 8
Trying 9
Trying 10
Trying 11
Trying 12
Trying 13
Trying 14
Trying 15
Trying 16
Trying 17
Trying 18
Trying 19
Trying 20
Trying 21
```

still room for improvement!

SLOW!!!!

generality

- The *most general type* for solver is polymorphic!

solver : (sol -> bool) -> state -> (sol -> 'a) -> (unit -> 'a) -> 'a

REASON?

The specification says why…

For all types t, and all s : sol -> t and k : unit -> t,
solver p (bs, rest) s k = s L,
where L is a solution extending bs…
if there is one
= k( ),
otherwise

Behavior is *oblivious* of what s and k are.
The "answer" type can be chosen later!

cps benefits

- Polymorphic type, very general spec
    - highly adaptable
    - write-once/use-many
- Better efficiency — runtime is visibly faster!
    - code design implements *backtracking*, and *avoids* list-building/safe-checking

improvements

- Can get faster performance by exploiting ***symmetry***

    - **black** squares and **white** squares are *independent*
      so we can — in *parallel* —

        - find a way to place (white) bishops on *white* squares

        - find a way to place (black) bishops on *black* squares

- The *safe* check is **quadratic** in list length.
  This strategy works with shorter lists
  and avoids lots of unnecessary checking!

improvements

to the improvements

- Can get faster performance by exploiting ***symmetry***
    - **black** squares and **white** squares are *independent*
        - find a way to place bishops on *white* squares 
        - ***deduce*** a way to put bishops on *black* squares 
- When n is even, it's easy to *blacken* a white solution!

(we don't even need parallel evaluation!)

exercises

- Implement a smart solver (for n even)
  based on black/white independence

- Find some other solutions
  - e.g. non-symmetric placements

- Write a function that displays a solution as a picture

draw : int -> sol -> string

print (draw n L)

challenge

- Find a list of ***all*** safe placements
  for m bishops on an n-by-n board.

- You can assume n is even, again.

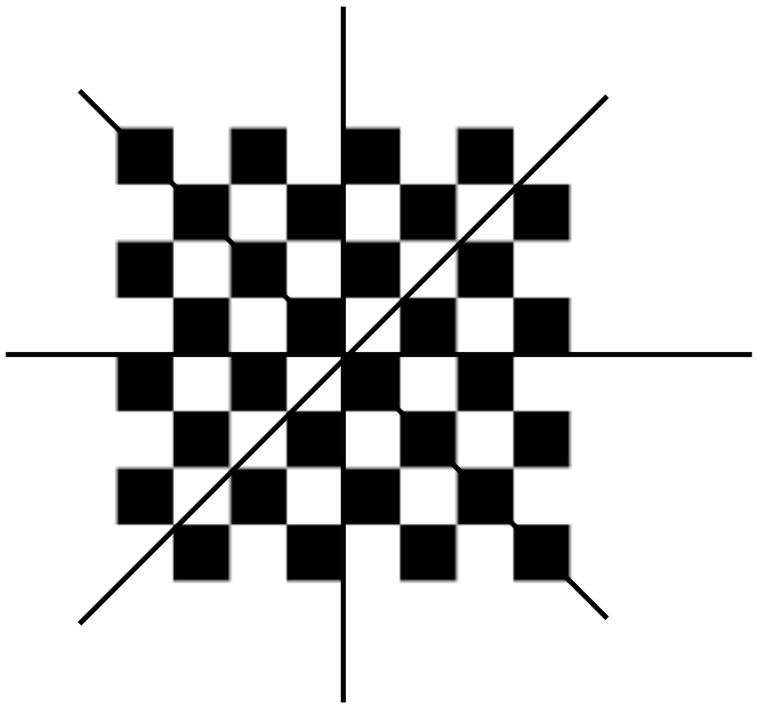
- And you can use ***solver*** from before, as a helper!

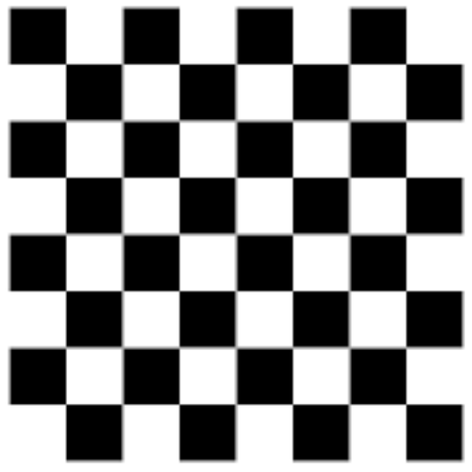
challenge

- Find a ***reduced*** list of safe placements for m bishops on an n-by-n board

- You can assume n is even, again.

- Don’t include solutions that are obtainable by *symmetry*.



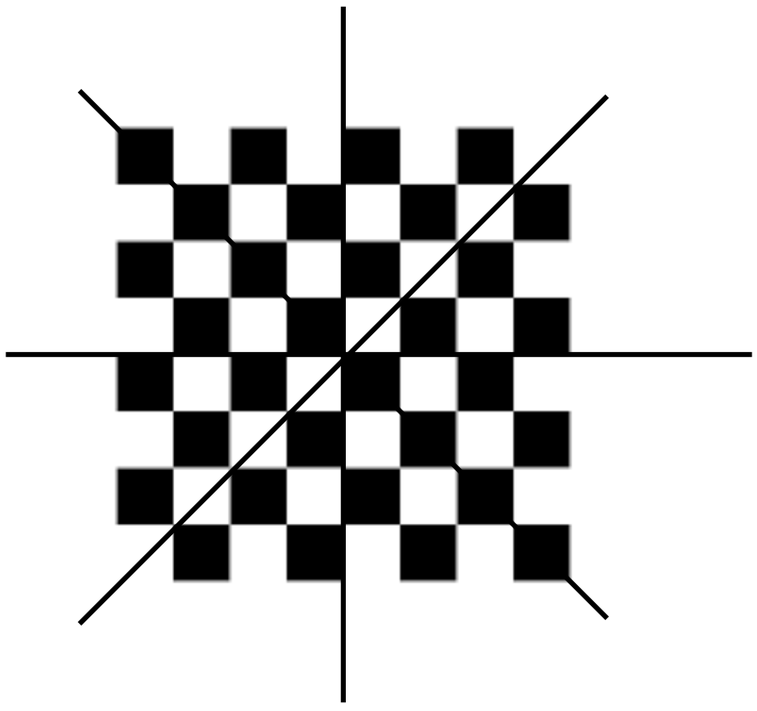


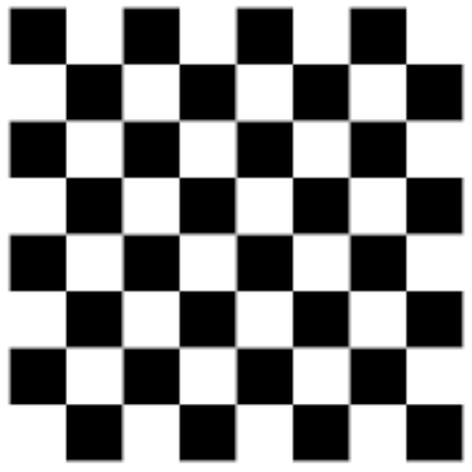
challenge

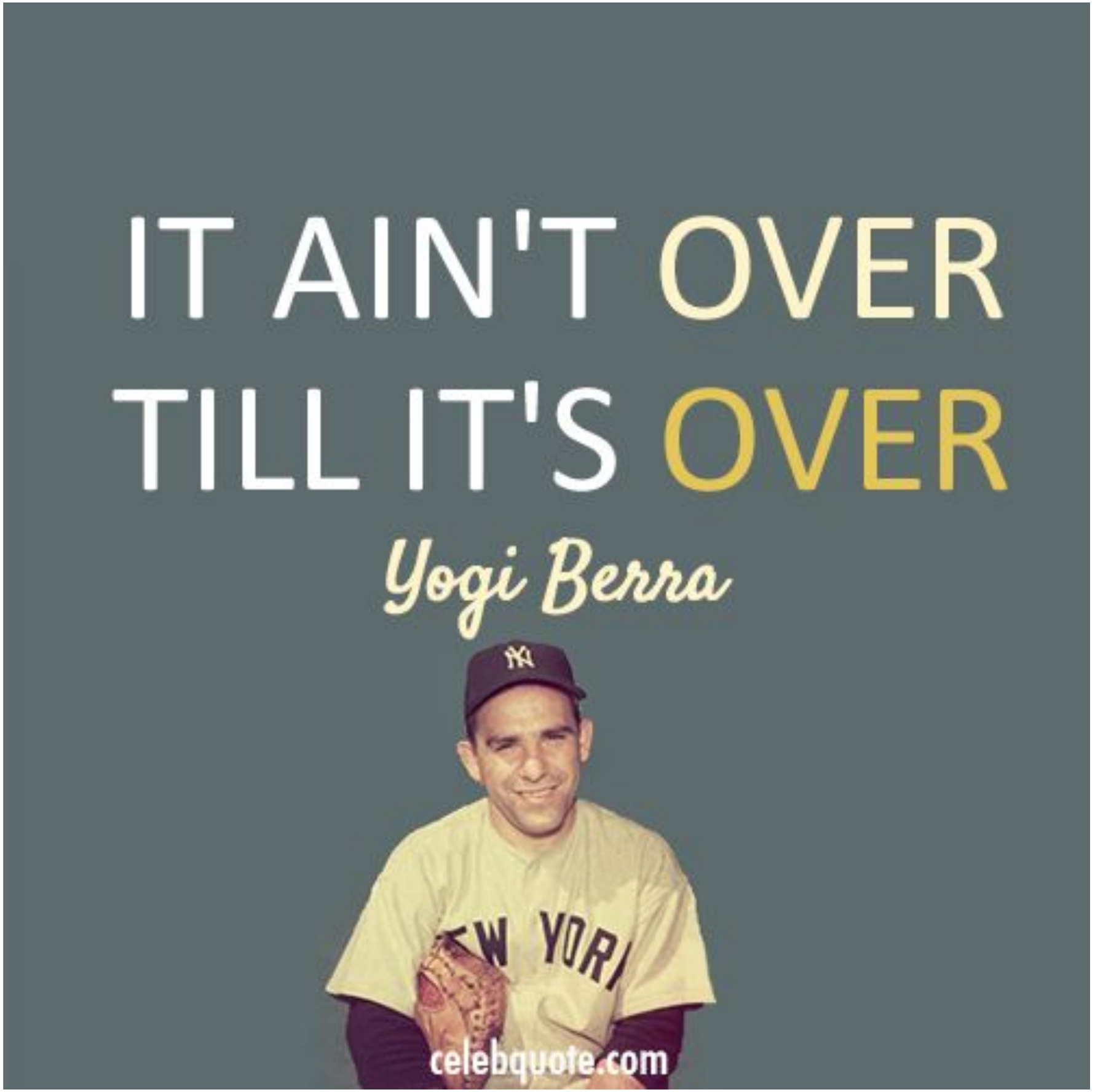
- Find a ***reduced*** list of safe placements for m bishops on an n-by-n board

- You can assume n is even, again.

- Don’t include solutions that are obtainable by *symmetry*.






IT AIN'T OVER
TILL IT'S OVER
Yogi Berra
celebquote.com


I NEVER SAID
MOST OF THE THINGS
I SAID
Yogi Berra
13
celebquote.com