Being Lazy

Peter Marks and Ben Moseley

Goals

- Understand the functional paradigm, lazy evaluation and Monadic IO
- Learn some key Haskell idioms and style tips
- Experience software development in Haskell

Not…

- Haskell tutorial
- Demo of maths functions
- Sales pitch… though we are passionate

Overview

- Introduction
- Code scenarios
- Issues and pitfalls

- Break

- Live coding
- Wrap up

The power of Haskell

- Purity
- Type system
- Laziness

- Rich syntax
- Sophisticated optimizer
- Extensible

- Extensive abstract libraries

Barriers to learning Haskell

- Purity
- Type system
- Laziness

- Rich syntax
- Sophisticated optimizer
- Extensible

- Extensive abstract libraries

Haskell

A lazily evaluated, pure functional language

Lazy evaluation

```
max(x + 5, y + 5)

(x + 5) * (y + 5)
```

(Not Haskell)

Lazy evaluation

```
(x != 0) && (y / x > 0)
```

(Not Haskell)

Lazy evaluation

```
foo x z = if x /= 0
             then (z > 0)
             else False
```

Lazy evaluation

```
foo x z = if x /= 0
             then (z > 0)
             else False
```

Lazy evaluation

```
foo x z = if x /= 0
             then (z > 0)
             else False
```

```
foo x (y `div` x)
```

Lazy evaluation

```
foo x z = if x /= 0
             then (z > 0)
             else False
```

```
foo x (y `div` x)
```

```
foo x (div y x)
```

Lazy evaluation

Can define your own control structures:

```
bool :: a -> a -> Bool -> a
bool t _ True  = t
bool _ f False = f
```

...replaces some uses of macros

```
contents <- bool
               readFile
               (throwError . ("Can't read:"++))
               isAdministrator file
```

Lazy evaluation

```
[42, 27, head [], 3]
```

Lazy evaluation

```
[42, 27, head [], 3] !! 3
```

Lazy evaluation

```
[42, 27, head [], 3] !! 3
```

```
allSame xs = all (== head xs) xs
```

Lazy evaluation

```
doubling a =
  a : doubling (a * 2)

take 9 $ doubling 3

[3,6,12,24,48,96,192,384,
768]
```

Code scenarios

Opportunities to be lazy

Names and numbers

Given a list of names print each one with its index in the list.

Q: How would you do this imperatively?
Q: How would you do this functionally?

Names and numbers

```
names = ["Fred","Jim","Bob"]

report ns = rep 1 ns
  where
    rep _ [] = []
    rep i (n:ns) = (printf "Name %d is %s." i n) :
                   (rep (i+1) ns)
```

Infinite data structure

```
names = ["Fred","Jim","Bob"]

zipWith (printf "Name %d is %s.")
  [1..length names]
  names
```

```
["Name 1 is Fred.", "Name 2 is Jim.", "Name 3 is
  Bob."]
```

Infinite data structure

```
names = ["Fred","Jim","Bob"]

zipWith (printf "Name %d is %s.")
   [1..length names]
   names


["Name 1 is Fred.", "Name 2 is Jim.", "Name 3 is
  Bob."]


zipWith (printf "Name %d is %s.") [1..] names
```

Infinite data structure

```
names = ["Fred","Jim","Bob"]

zipWith (printf "Name %d is %s.")
  [1..length names]
  names
```

```
["Name 1 is Fred.", "Name 2 is Jim.", "Name 3 is
  Bob."]
```

```
zipWith (printf "Name %d is %s.") [1..] names
```

Q: Why is using an infinite list better?

Top 50

Find the top 50 elements of a 50000 list.

Q: What is the obvious way to do this?
Q: Would there be any issues with that approach?

Avoid unnecessary work

```
qsort []     = []
qsort (x:xs) = qsort (filter (> x) xs) ++
               [x] ++
               qsort (filter (<= x) xs)
```

Avoid unnecessary work

```
qsort []     = []
qsort (x:xs) = qsort (filter (> x) xs) ++
               [x] ++
               qsort (filter (<= x) xs)

top50 = take 50 . qsort
```

Avoid unnecessary work

```
qsort []     = []
qsort (x:xs) = qsort (filter (> x) xs) ++
               [x] ++
               qsort (filter (<= x) xs)

top50 = take 50 . qsort

vals <- take 50000 . randoms <$> newStdGen
```

Avoid unnecessary work

```
qsort []     = []
qsort (x:xs) = qsort (filter (> x) xs) ++
               [x] ++
               qsort (filter (<= x) xs)

top50 = take 50 . qsort

vals <- take 50000 . randoms <$> newStdGen

qsort vals -- Takes 0.50s in GHCi
```

Avoid unnecessary work

```
qsort []     = []
qsort (x:xs) = qsort (filter (> x) xs) ++
               [x] ++
               qsort (filter (<= x) xs)

top50 = take 50 . qsort

vals <- take 50000 . randoms <$> newStdGen

qsort vals -- Takes 0.50s in GHCi
top50 vals -- Takes 0.14s in GHCi
```

Recursive definitions

- Recursive function:

```
len xs = case xs of
           [] -> 0
           _  -> 1 + len (tail xs)
```

Recursive definitions

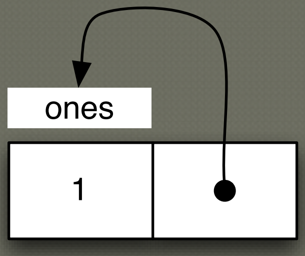
- Cyclic values:

```
ones         = 1 : ones
```

Recursive definitions

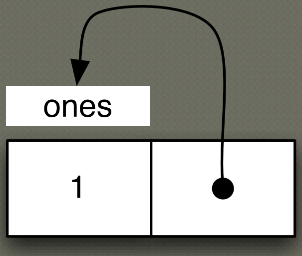
- Cyclic values:

```
ones            = 1 : ones
```



Recursive definitions

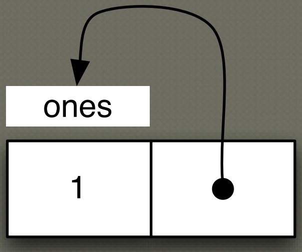
- Cyclic values:



```
ones       = 1 : ones

alternates = 1 : 0 : alternates
```

Recursive definitions

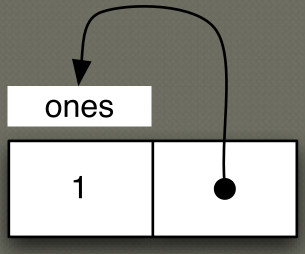
- Cyclic values:



```
ones        = 1 : ones

alternates = 1 : 0 : alternates

months = "Jan" : "Feb" : "Mar" : "Apr"
       : "May" : "Jun" : "Jul" : "Aug"
       : "Sep" : "Oct" : "Nov" : "Dec"
       :  months
```

Recursive definitions

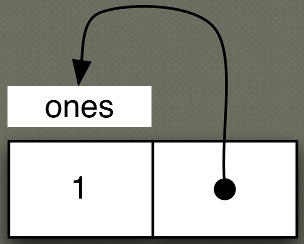
- Cyclic values:



```
ones       = 1 : ones

alternates = 1 : 0 : alternates

months = "Jan" : "Feb" : "Mar" : "Apr"
       : "May" : "Jun" : "Jul" : "Aug"
       : "Sep" : "Oct" : "Nov" : "Dec"
       :  months

nMonthsAfter m n =
  dropWhile (/=m) months !! n
```

Recursive definitions

- Cyclic values:



```
ones       = 1 : ones

alternates = 1 : 0 : alternates

months = "Jan" : "Feb" : "Mar" : "Apr"
       : "May" : "Jun" : "Jul" : "Aug"
       : "Sep" : "Oct" : "Nov" : "Dec"
       :  months

nMonthsAfter m n =
  dropWhile (/=m) months !! n

nMonthsAfter "May" 25 ----> "Jun"
```

Tom and Jerry

Define two types:
– Cat has a name and a victim (Mouse).
– Mouse has a name and a tormentor (Cat).

Create instances:
– Cat: Tom whose victim is Jerry.
– Mouse: Jerry whose tormentor is Tom.

Q: How would you do this imperatively?
Q: Could laziness help?

Cyclic graph

```
data Cat = Cat {
        cname  :: String
        victim :: Mouse
      }
      deriving Show

data Mouse = Mouse {
        mname     :: String
        tormentor :: Cat
      }
      deriving Show
```

Cyclic graph

```
data Cat = Cat {
          cname  :: String
          victim :: Mouse
        }
        deriving Show

data Mouse = Mouse {
          mname     :: String
          tormentor :: Cat
        }
        deriving Show

tom   = Cat   {cname = "Tom",   victim    = jerry}
jerry = Mouse {mname = "Jerry", tormentor = tom}
```

Powers of 2

Define an infinite list of the powers of 2 using a cyclic definition

Q: How would you do this?

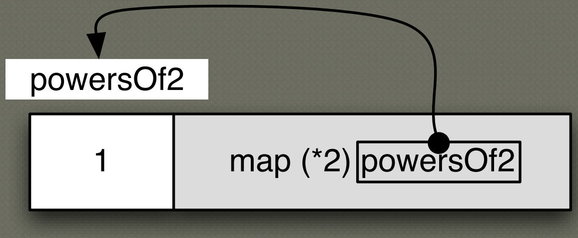
Cyclic definition

```
powersOf2 = 1 : map (* 2) powersOf2
```

```
[1,2,4,8,16,32,64,128,256,512,1024,20
48,4096,8192,16384,32768,65536,131072
,262144,524288...
```

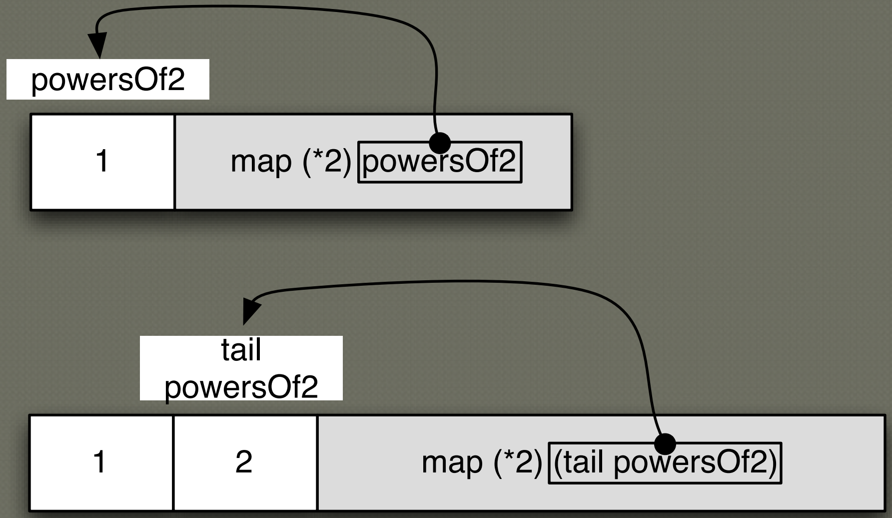
Cyclic definition

```
powersOf2 = 1 : map (* 2) powersOf2
```



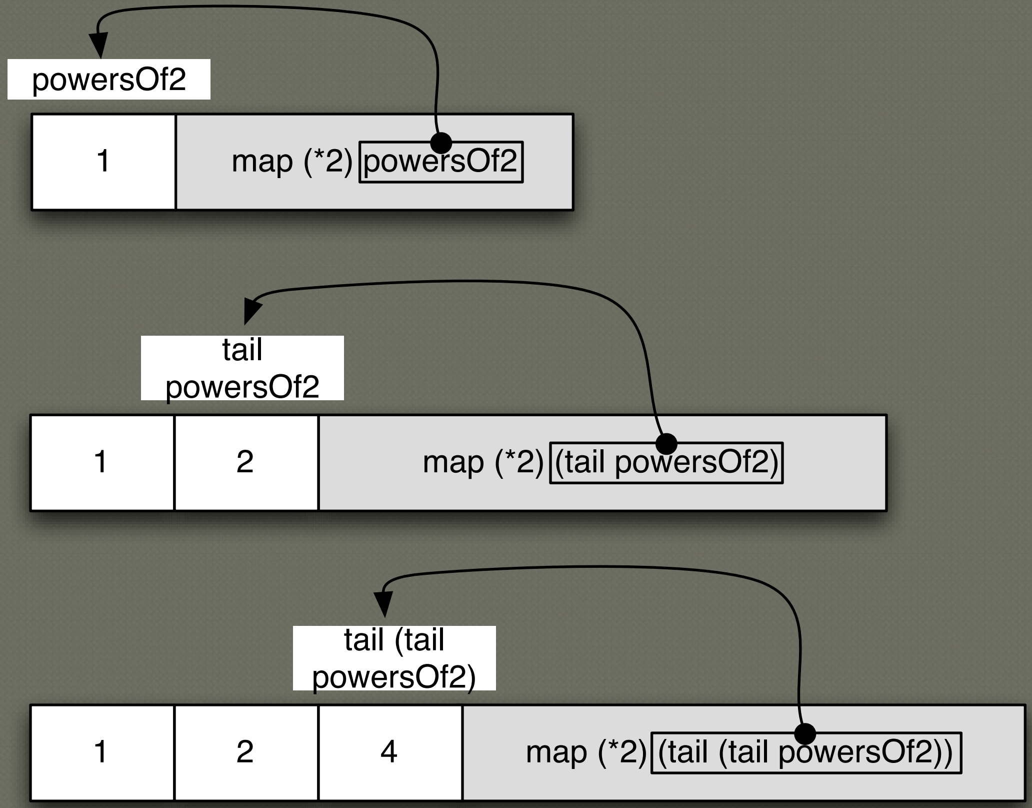
Cyclic definition

```
powersOf2 = 1 : map (* 2) powersOf2
```



Cyclic definition

```
powersOf2 = 1 : map (* 2) powersOf2
```



Memoization

Fibonnaci numbers are:

```
[1,1,2,3,5,8,13,21,34,55,89,14
 4,233,377,610,987,1597,2584,4
 181,6765...
```

Q: How would find the ‘nth’ one?

Memoization

Q: How would you do this in Haskell?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Memoization

Q: How would you do this in Haskell?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

fib 30 takes about 4.3s
fib 40 will probably take about an hour...

Memoization

Q: How would you do this in Haskell?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

fib 30 takes about 4.3s
fib 40 will probably take about an hour...

```
fib' n = fibs !! n
  where
    fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Memoization

Q: How would you do this in Haskell?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

fib 30 takes about 4.3s
fib 40 will probably take about an hour...

```
fib' n = fibs !! n
  where
    fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

...fib’ 4000 takes 0.01s according to GHCi:

```
645748844909481735313769490153695956444139006401513427084075775981772103
590340889144494778072872417437607415237838188974992270097421831524820190
627635507987437042751068564702163075936230573885067767672020696704775060
888952943005092911660239478668417638539538139822817039366653699227090953
080068213995247807210499558291914070299436220877792964591740126101486595
203811704525911413319493360805771417086457836066360819419152173551158109
939739457834939838445927496726613615480616157565958189443176199220973699
176769740582063418920881445493379744229521401326215683407010162734227278
277627261530663030930529820517574447424280331075224194662196557804131017
595052316172225782924860810023912187851892996757577669202694023487336446
627257747177409240688283001864394259217610825454631646288077026537526196
161573244340403420573366832792840985908015O1
```

Reverse lines

Read lines from a large file, reverse the characters of each line and write the result to a new file.

Q: How would you do this imperatively?

IO loop

```
main = do
  i <- openFile "input"  ReadMode
  o <- openFile "output" WriteMode
  untilM_ (hIsEOF i) $ do
    l <- hGetLine i
    hPutStrLn o (reverse l)
  hClose i
  hClose o
```

IO loop

```
main = do
  i <- openFile "input"  ReadMode
  o <- openFile "output" WriteMode
  untilM_ (hIsEOF i) $ do
    l <- hGetLine i
    hPutStrLn o (reverse l)
  hClose i
  hClose o
```

Q: How could you do it more elegantly?

Lazy IO

```
main = do
  i <- readFile  "input"
  let o = unlines . map reverse . lines $ i
  writeFile "output" o
```

Code is simpler...

Lazy IO

```
main = do
  i <- readFile  "input"
  let o = unlines . map reverse . lines $ i
  writeFile "output" o
```

Code is simpler... and still scalable.

Lazy IO

```
main = do
  i <- readFile  "input"
  let o = unlines . map reverse . lines $ i
  writeFile "output" o
```

Code is simpler... and still scalable.

```
main = interact
          (unlines . map reverse . lines)
```

...if just want stdin / stdout, the above is even simpler

Pitfalls

- Debugging
- Lazy IO gotchas – eg file handles
- Performance
  - Harder to understand / predict
  - Benchmarking
  - Performance cost to creating thunks
  - Space Leaks
  - Too much laziness
  - Too little laziness

Pitfalls

```
foldl (+) 0 [1..100000]
5000050000
       8,216,376 bytes copied during GC
       1,706,916 bytes maximum residency (4 sample(s))
  %GC time      64.3%  (61.7% elapsed)
```

Pitfalls

```
foldl (+) 0 [1..100000]
5000050000
       8,216,376 bytes copied during GC
       1,706,916 bytes maximum residency (4 sample(s))
  %GC time      64.3%  (61.7% elapsed)


foldl' (+) 0 [1..100000]
5000050000
           4,180 bytes copied during GC
           3,732 bytes maximum residency (1 sample(s))
  %GC time       2.2%  (2.8% elapsed)
```

Pitfalls

`foldl (+) 0 [1..100000]`
```
5000050000
       8,216,376 bytes copied during GC
       1,706,916 bytes maximum residency (4 sample(s))
  %GC time       64.3%  (61.7% elapsed)
```

`foldl' (+) 0 [1..100000]`
```
5000050000
           4,180 bytes copied during GC
           3,732 bytes maximum residency (1 sample(s))
  %GC time        2.2%  (2.8% elapsed)
```

`foldl (+) 0 [1..100000]` **-O2** *(this is strictness analysis)*
```
5000050000
           4,152 bytes copied during GC
           3,716 bytes maximum residency (1 sample(s))
  %GC time        2.3%  (3.1% elapsed)
```

Pitfalls

- `f = ("Report\n"++) . unlines . map show`

Pitfalls

- `f = (“Report\n”++) . unlines . map show`

- `if length xs >= 0 then f (tail xs) else f xs`

Pitfalls

- `f = ("Report\n"++) . unlines . map show`
- `if length xs >= 0 then f (tail xs) else f xs`
- `if not (null xs) then f (tail xs) else f xs`

Pitfalls

- `f = (“Report\n”++) . unlines . map show`
- `if length xs >= 0 then f (tail xs) else f xs`
- `if not (null xs) then f (tail xs) else f xs`
- `f (if not (null xs) then tail xs else xs)`

Break

Live coding

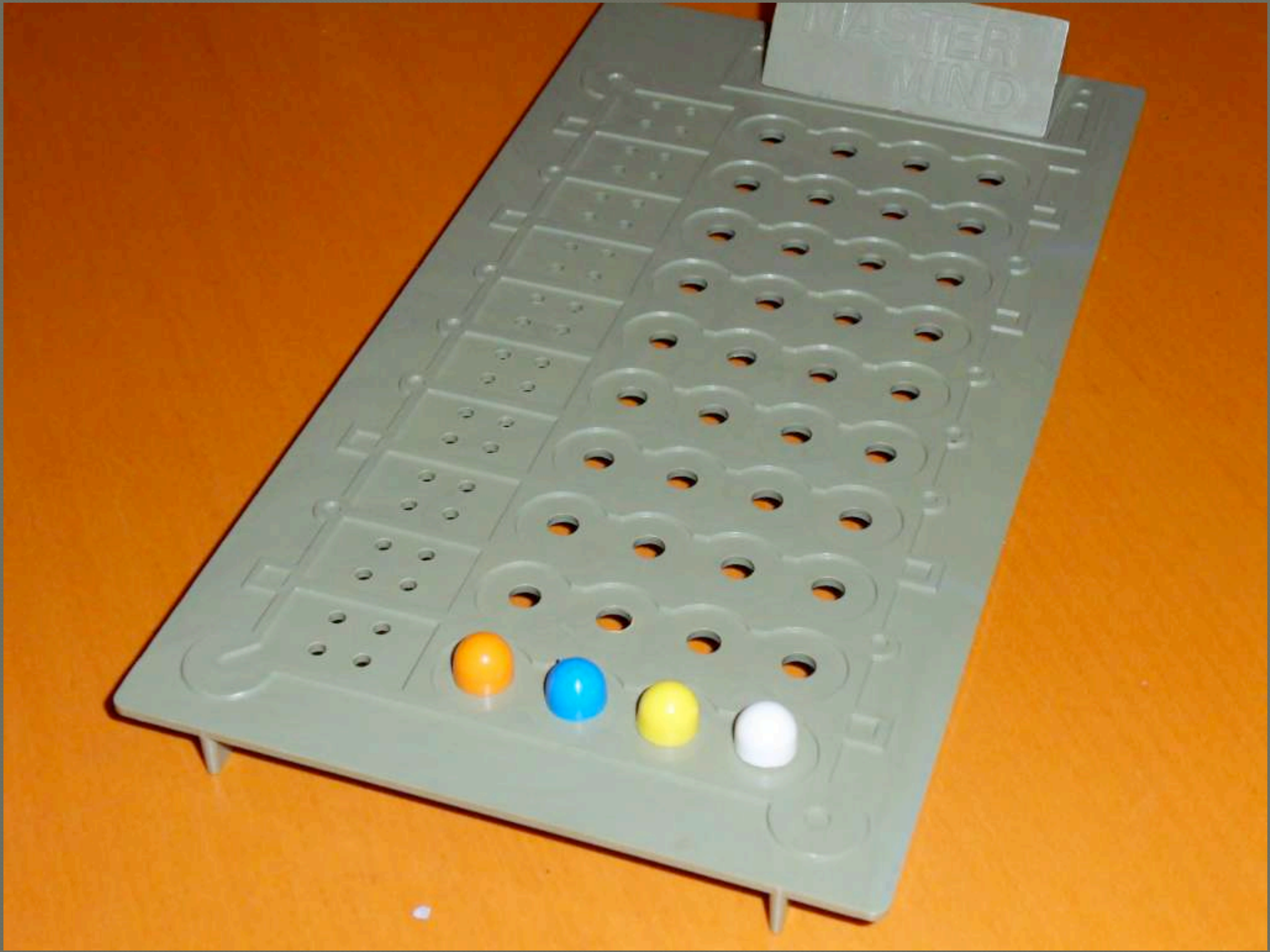
A larger example of laziness

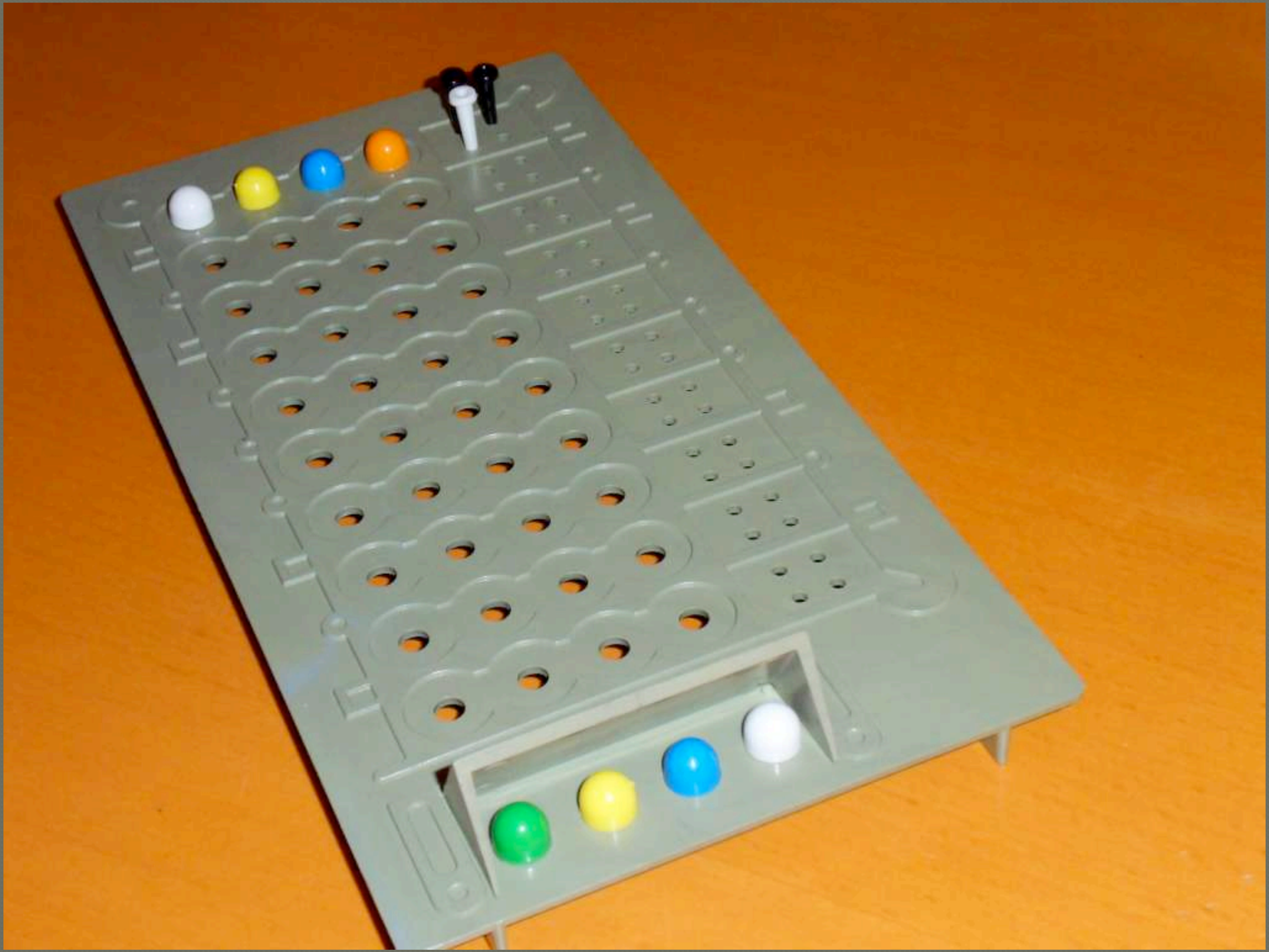
Mastermind

- Player 1 chooses a secret sequence of 4 tokens from a pool of 6 (no repeats).
- Player 2 makes a guess.
- Player 1 scores the guess indicating:
  - How many are the correct token in the correct position
  - How many are the correct token in the wrong position
- Play continues until the secret is guessed or player 2 gives up.


MASTER MIND
A GAME OF CUNNING AND LOGIC FOR TWO PLAYERS
BREAK THE HIDDEN CODE


STER MIND


MASTER
MIND

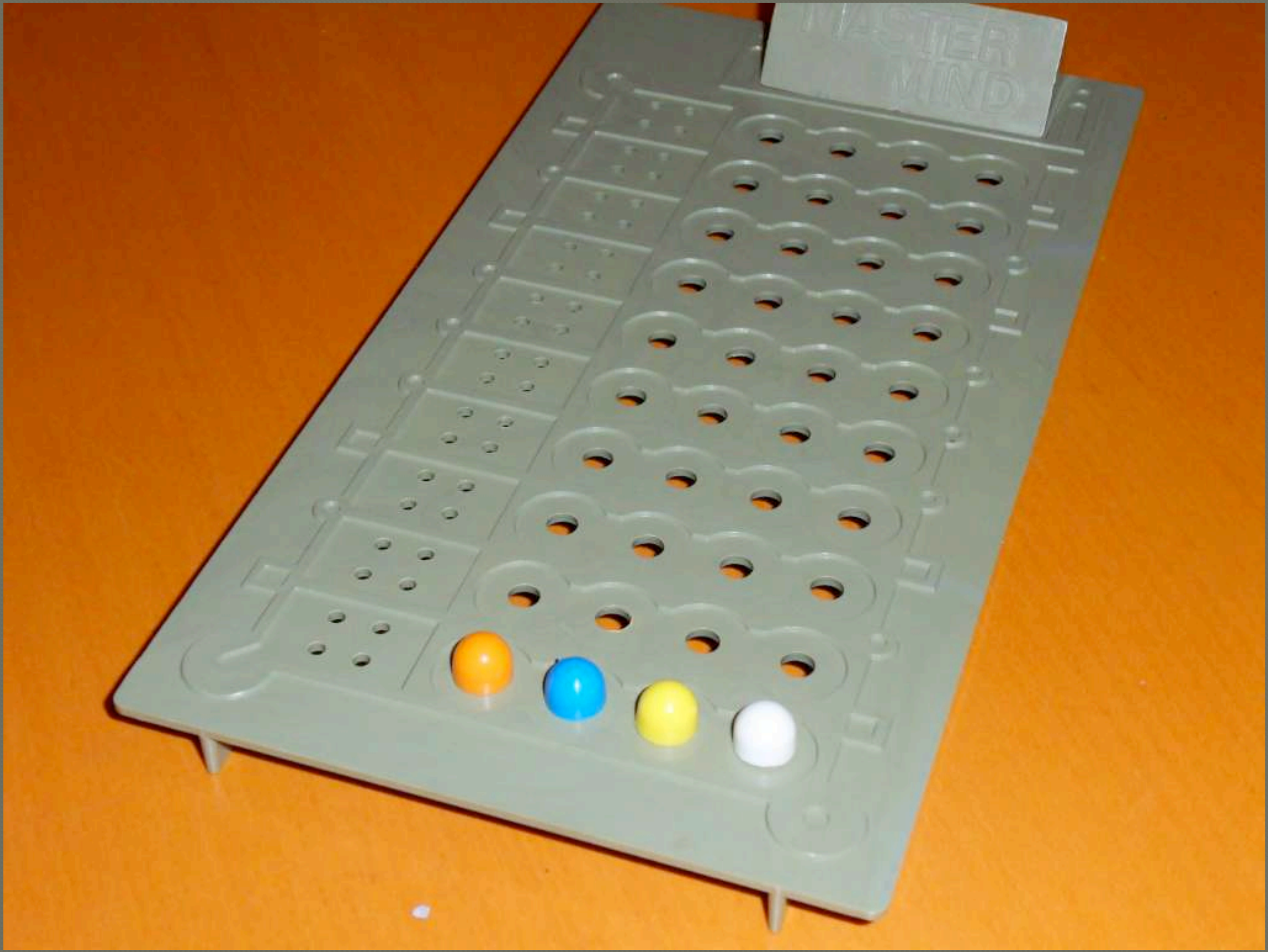



MASTER MIND

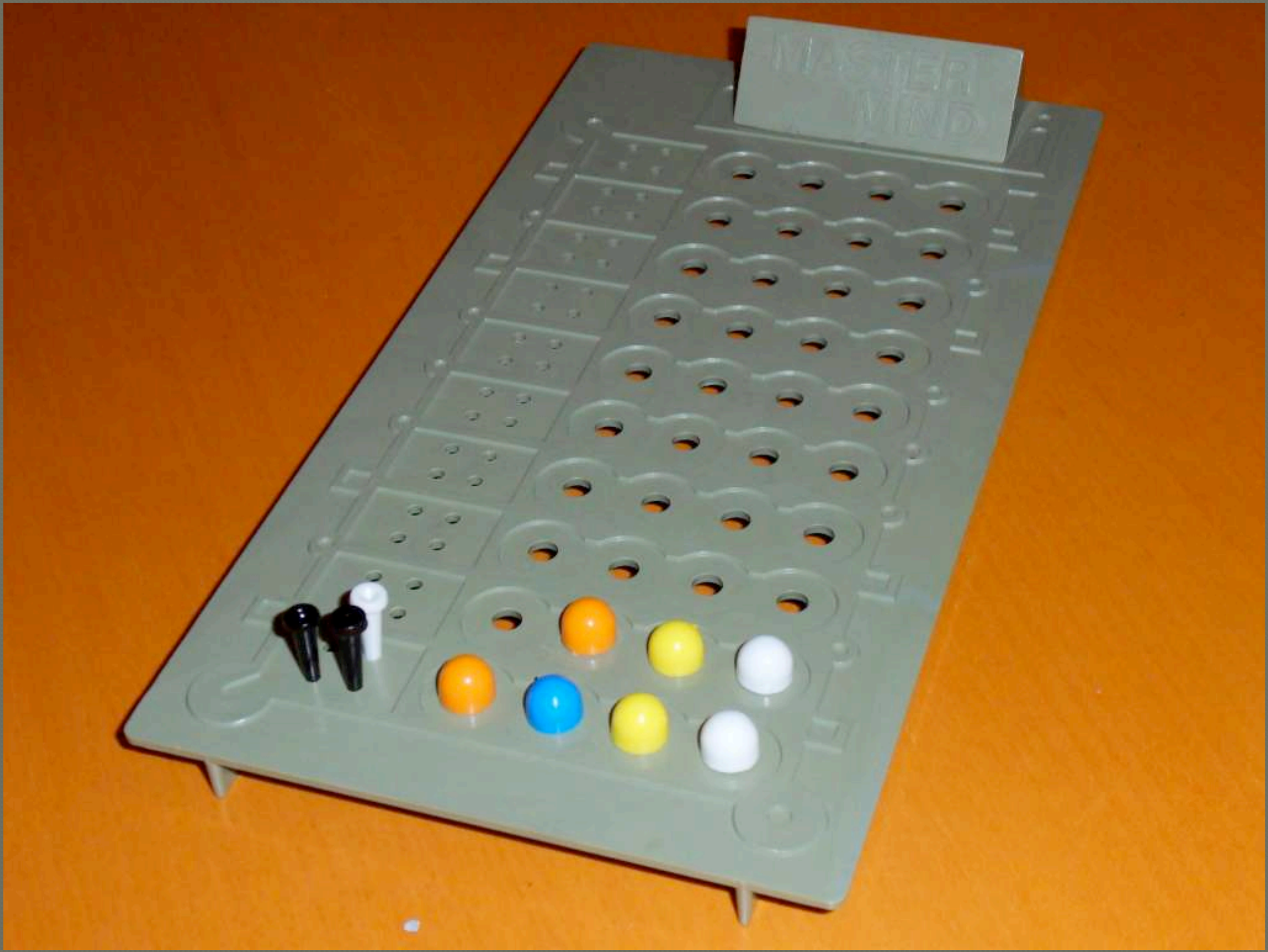

MASTER
MIND

MASTER MIND

Lets Code!

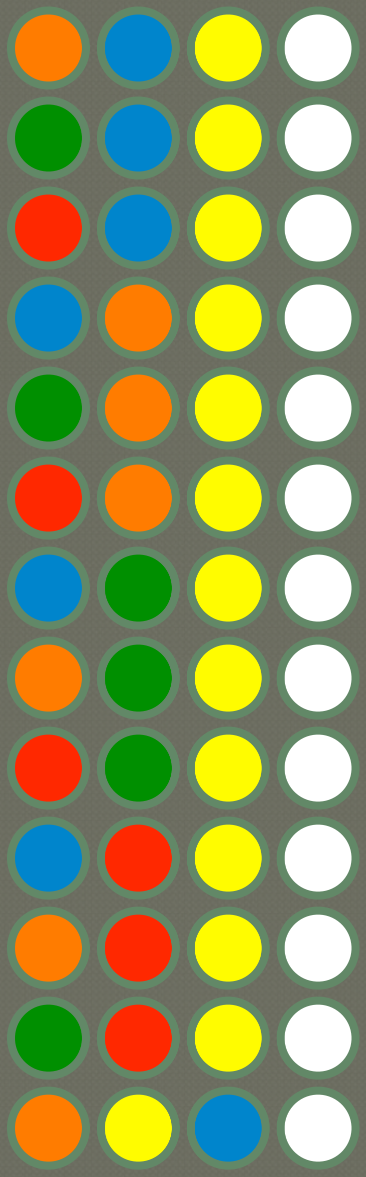

MASTER
MIND


MASTER
MIND

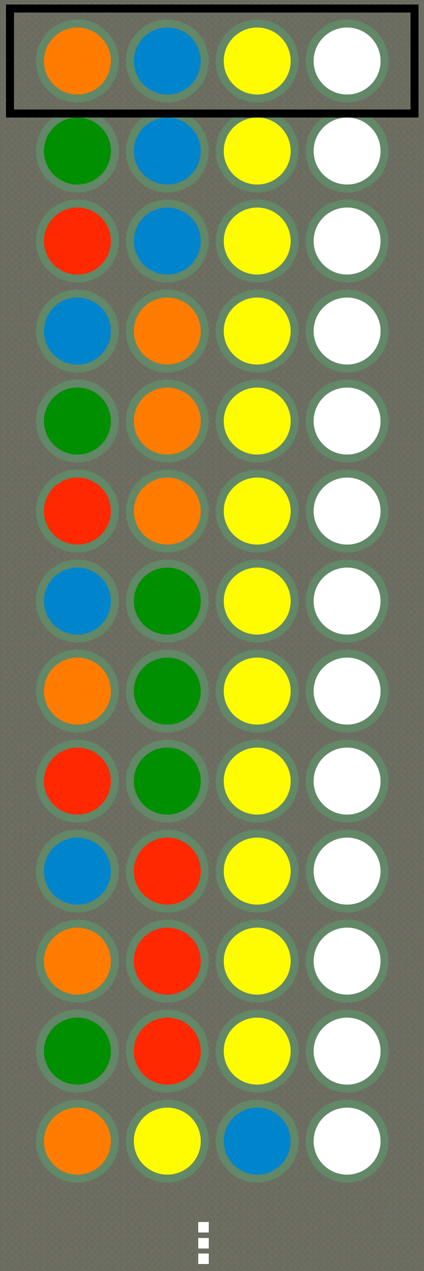

MASTER
MIND

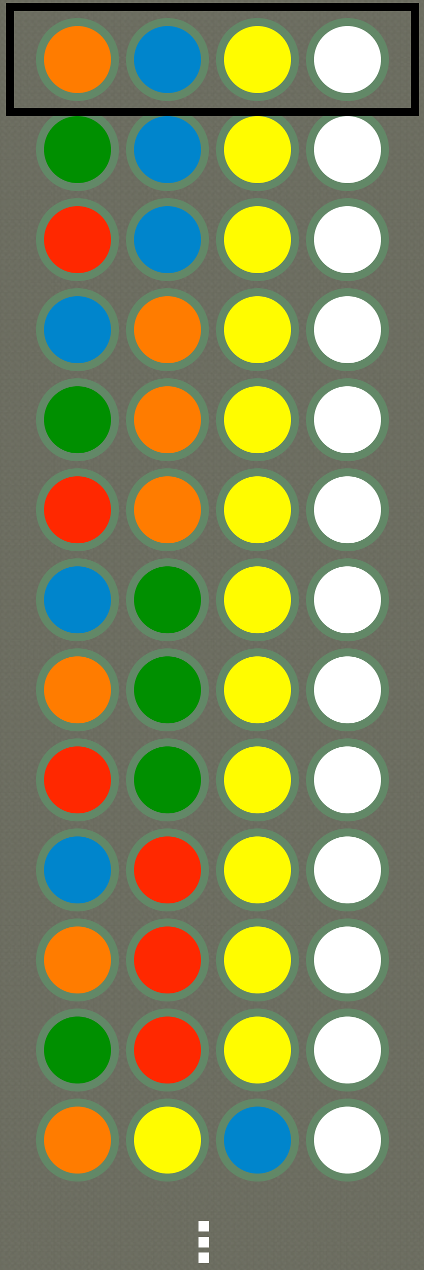

MASTER MIND

MASTER
MIND

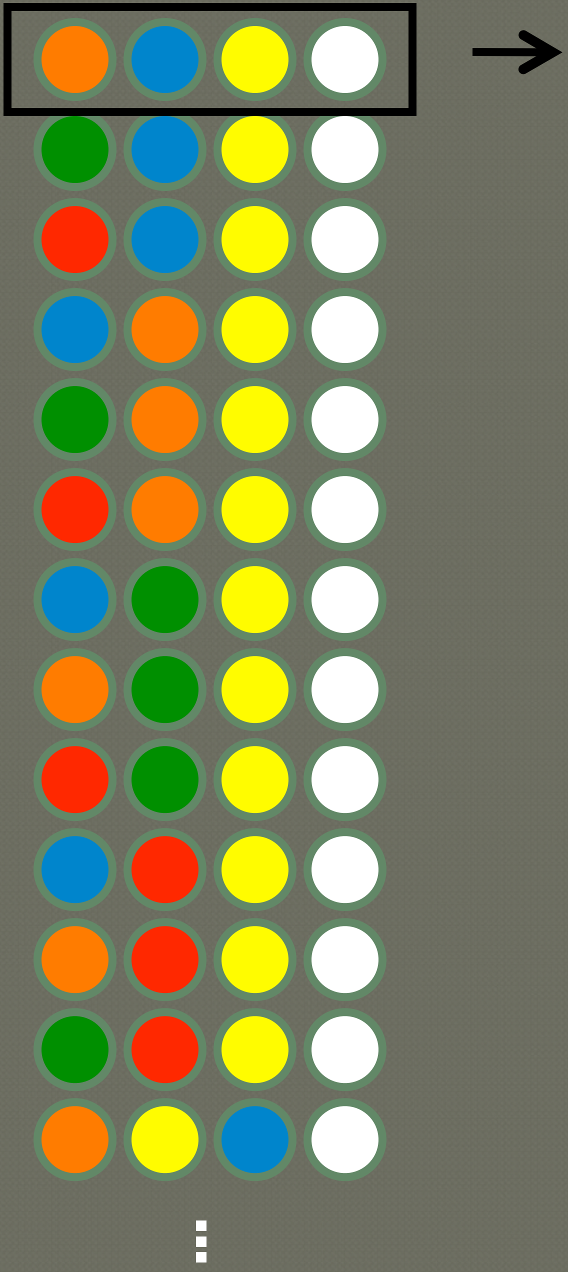


⋮

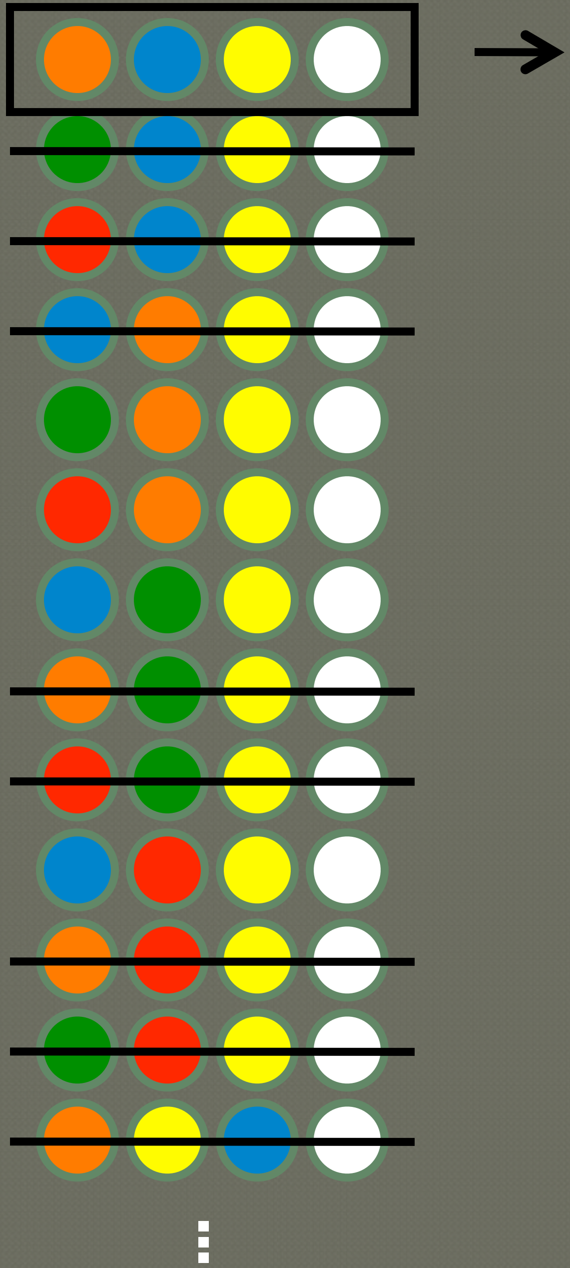




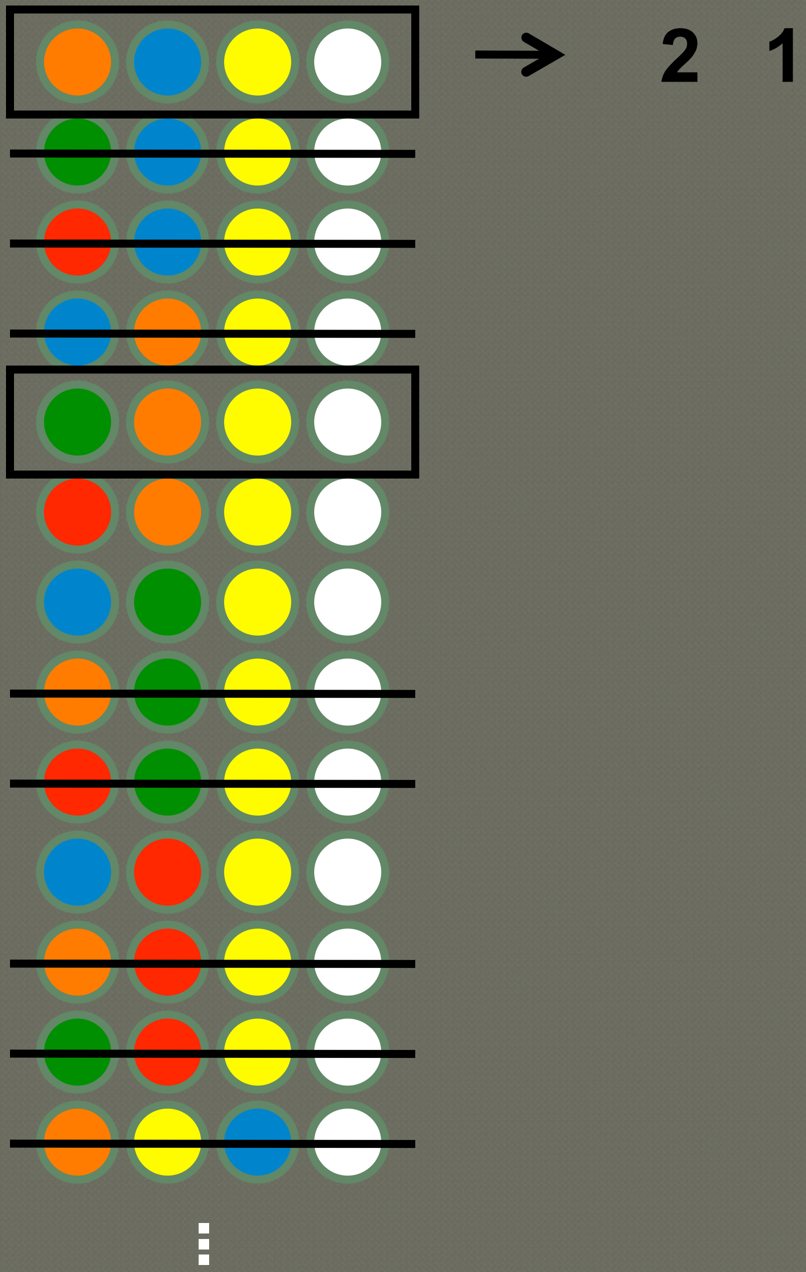
→ 2 1

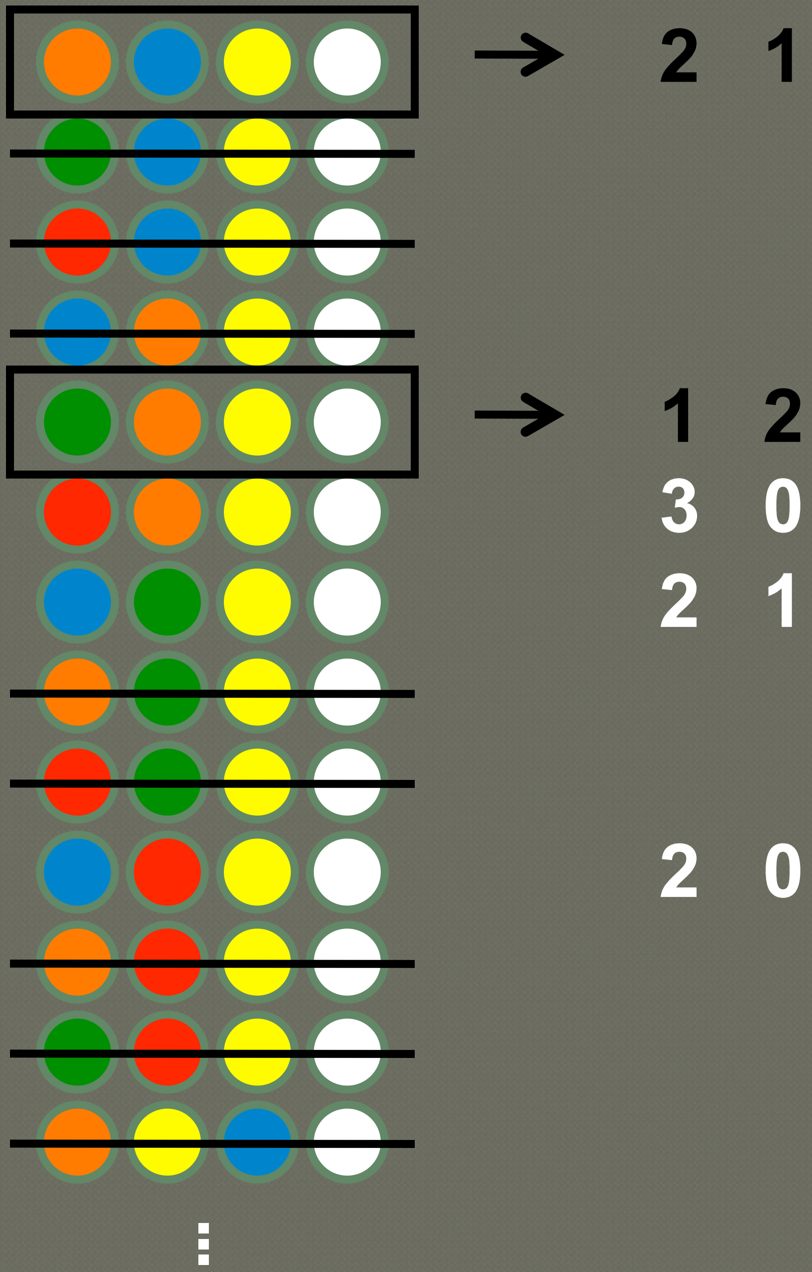


2 1
3 0
3 0
2 2
2 1
2 1
2 1
3 0
2 0
2 1
3 0
2 0
2 2



2 1
3 0
3 0
2 2
2 1
2 1
2 1
3 0
2 0
2 1
3 0
2 0
2 2


→ 2 1



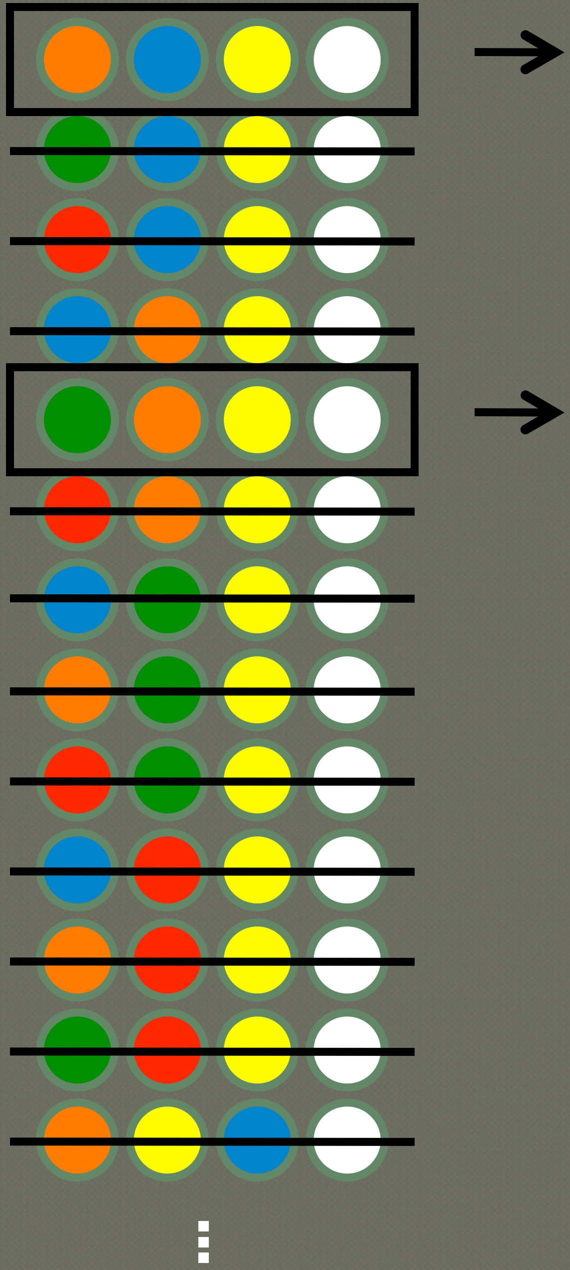
2 1

1 2

3 0

2 1

2 0



2 1

1 2
3 0
2 1

2 0

More Coding

Review

- Lazy evaluation
- Infinite structures
- Avoiding unnecessary work
- Cyclic definitions
- Memoization
- Lazy IO
- Circular programming