Programming in Haskell
Aug-Nov 2015

LECTURE 13

SEPTEMBER 29, 2015

S P Suresh
Chennai Mathematical Institute

Search problems

- No closed form for the solution
- Go through a cycle of
  - expanding out possible solutions
  - undoing partial solutions when we reach a dead end
- **Backtracking**

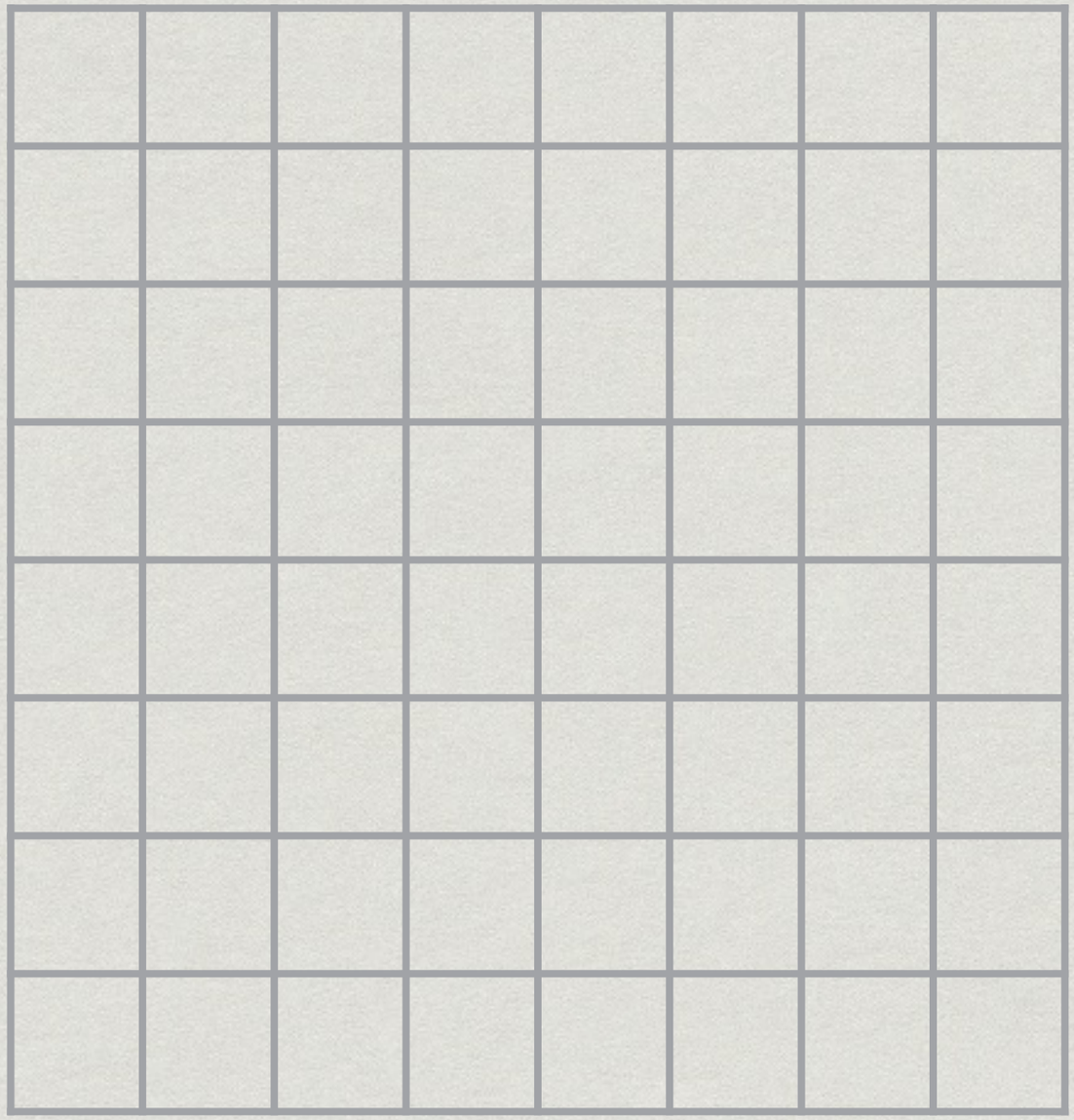
N queens

- Place N queens on N x N chessboard so that no two attack each other
  - Two queens attack each other if they are on the same row, column or diagonal
- N queens, N rows, N columns
  - Exactly one queen in each row, each column

Heuristic

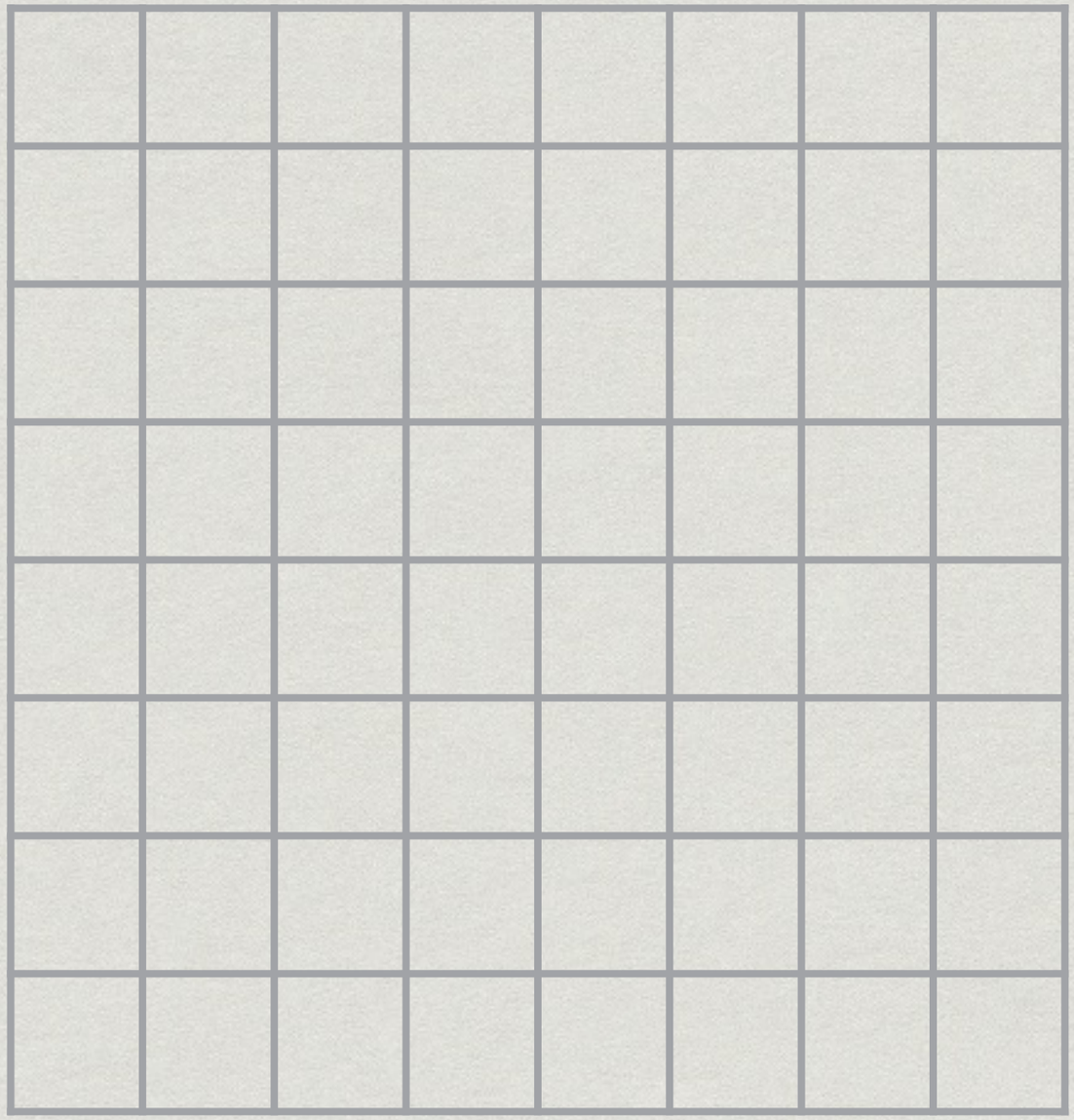
- Place the first queen somewhere in the first row
- In each succeeding row, place a queen at the leftmost square that is not attacked by any of the earlier queens

Backtracking ...



Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner

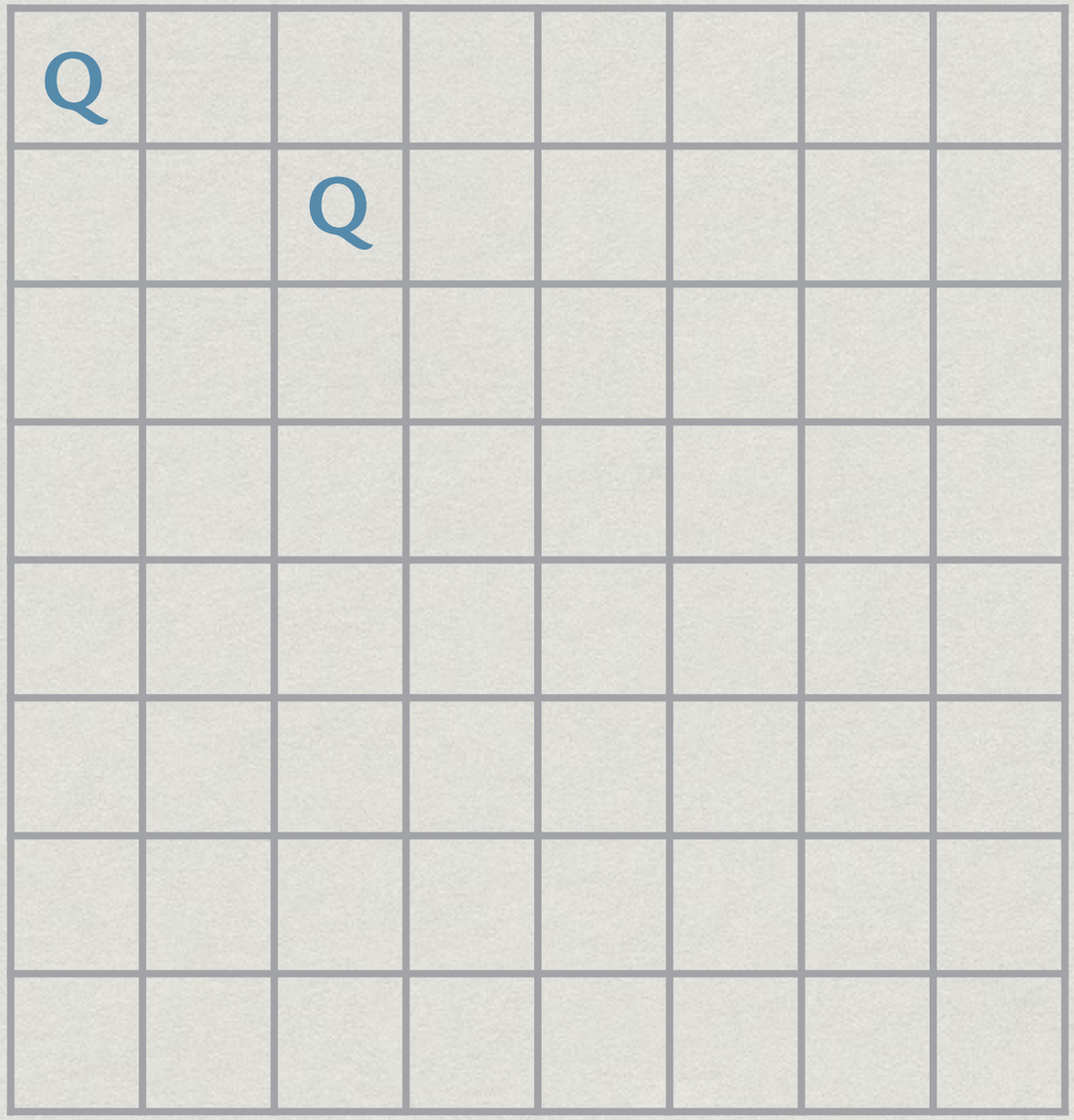


Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner

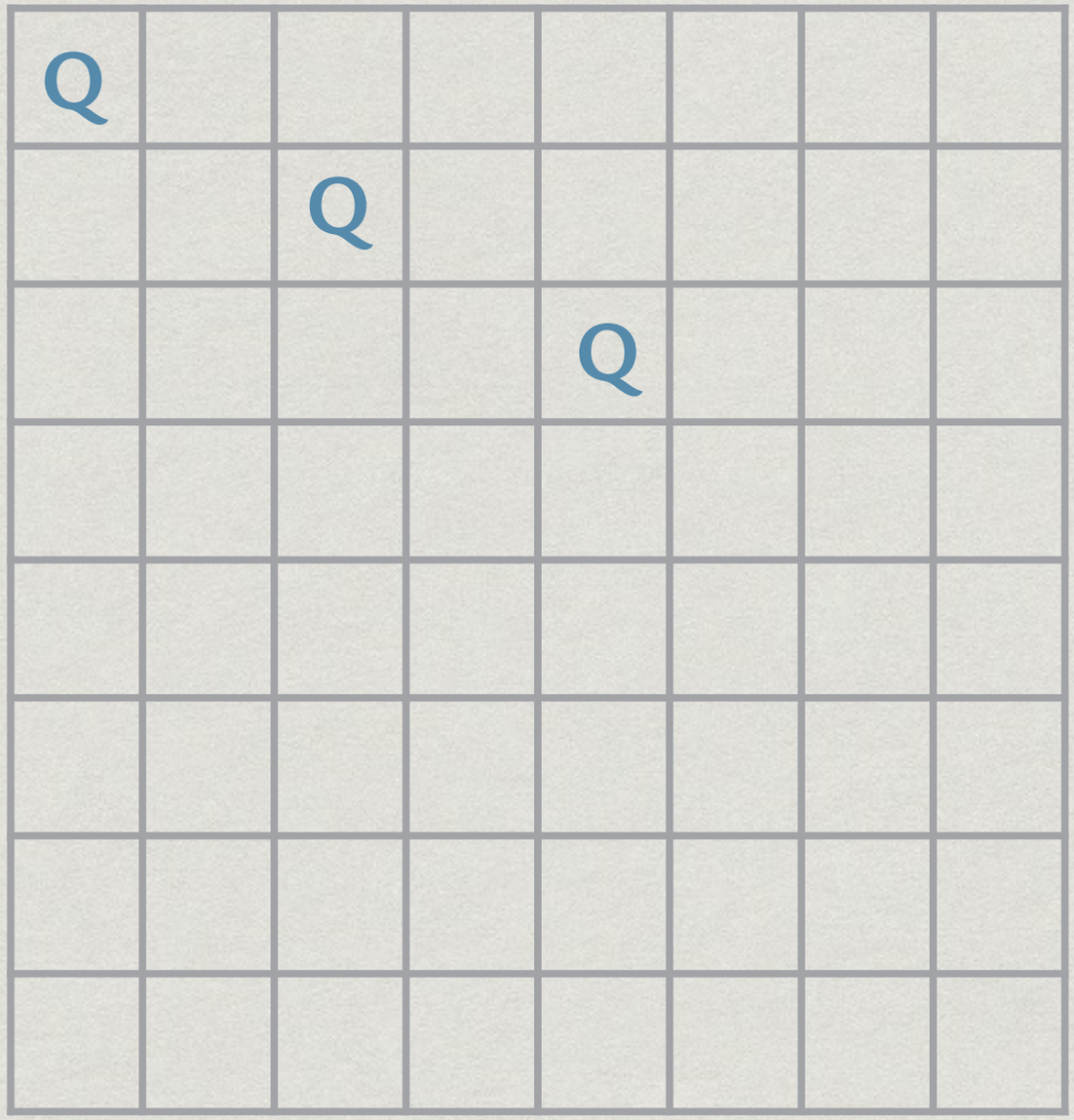
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



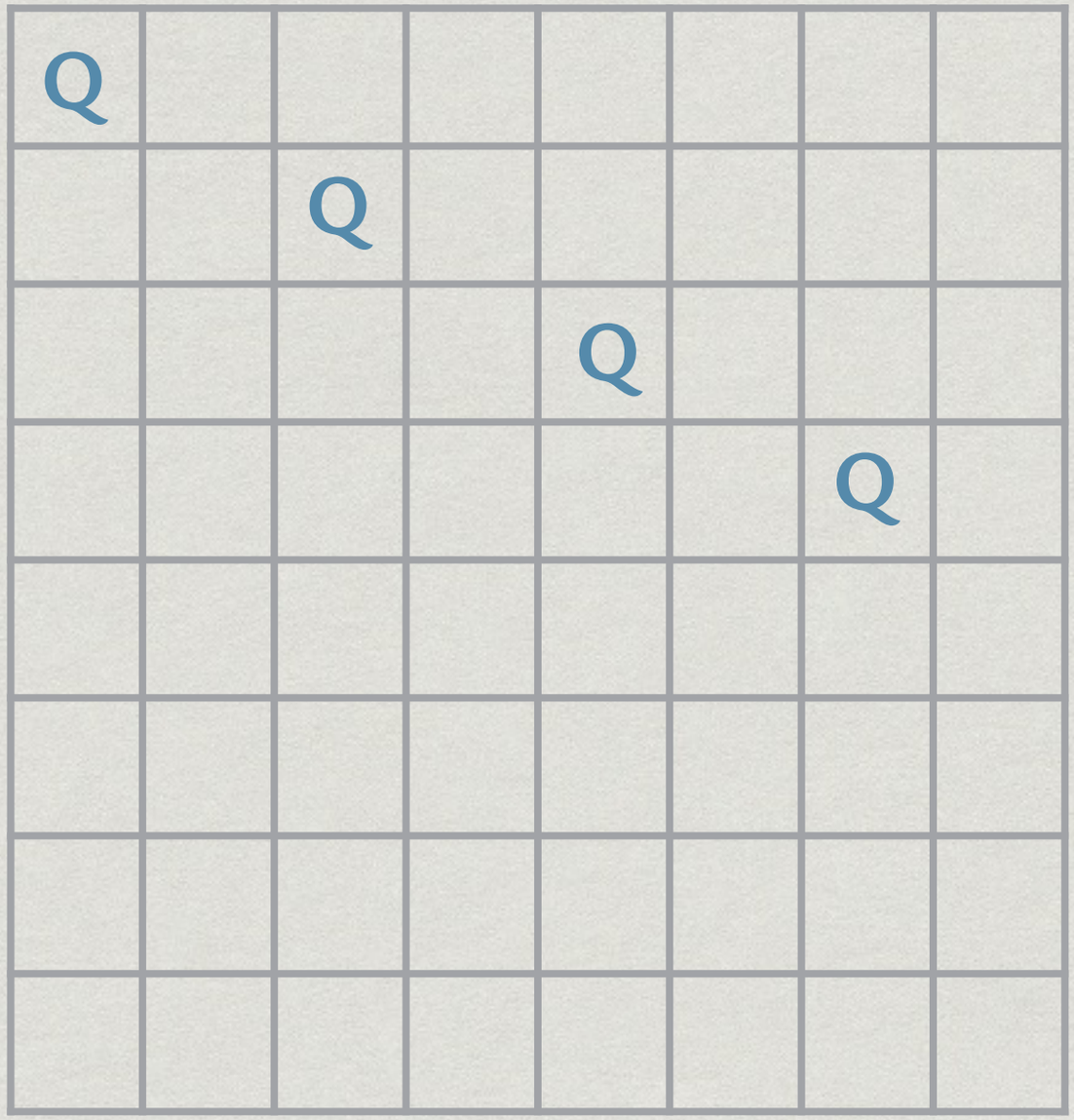
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



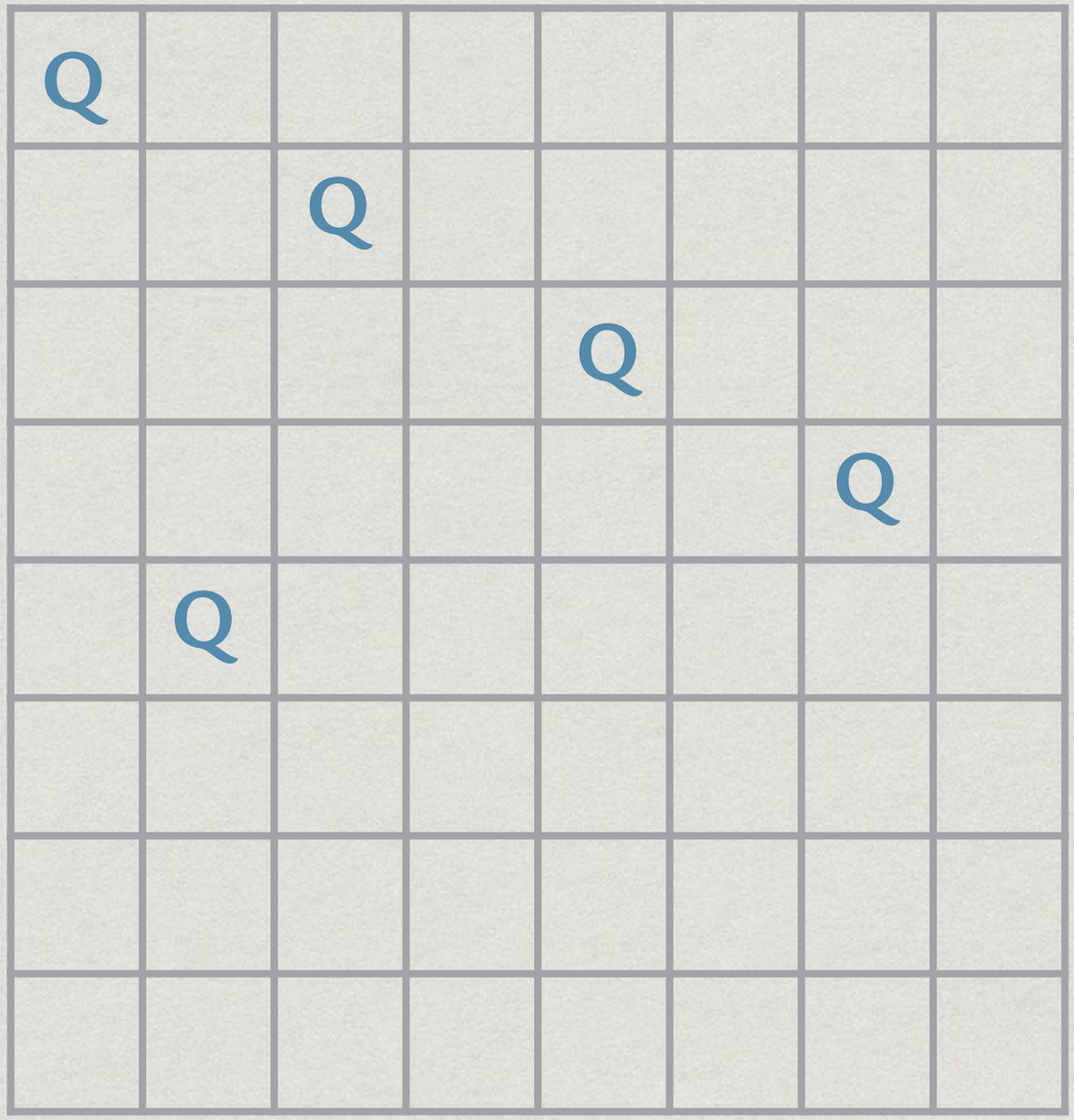
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



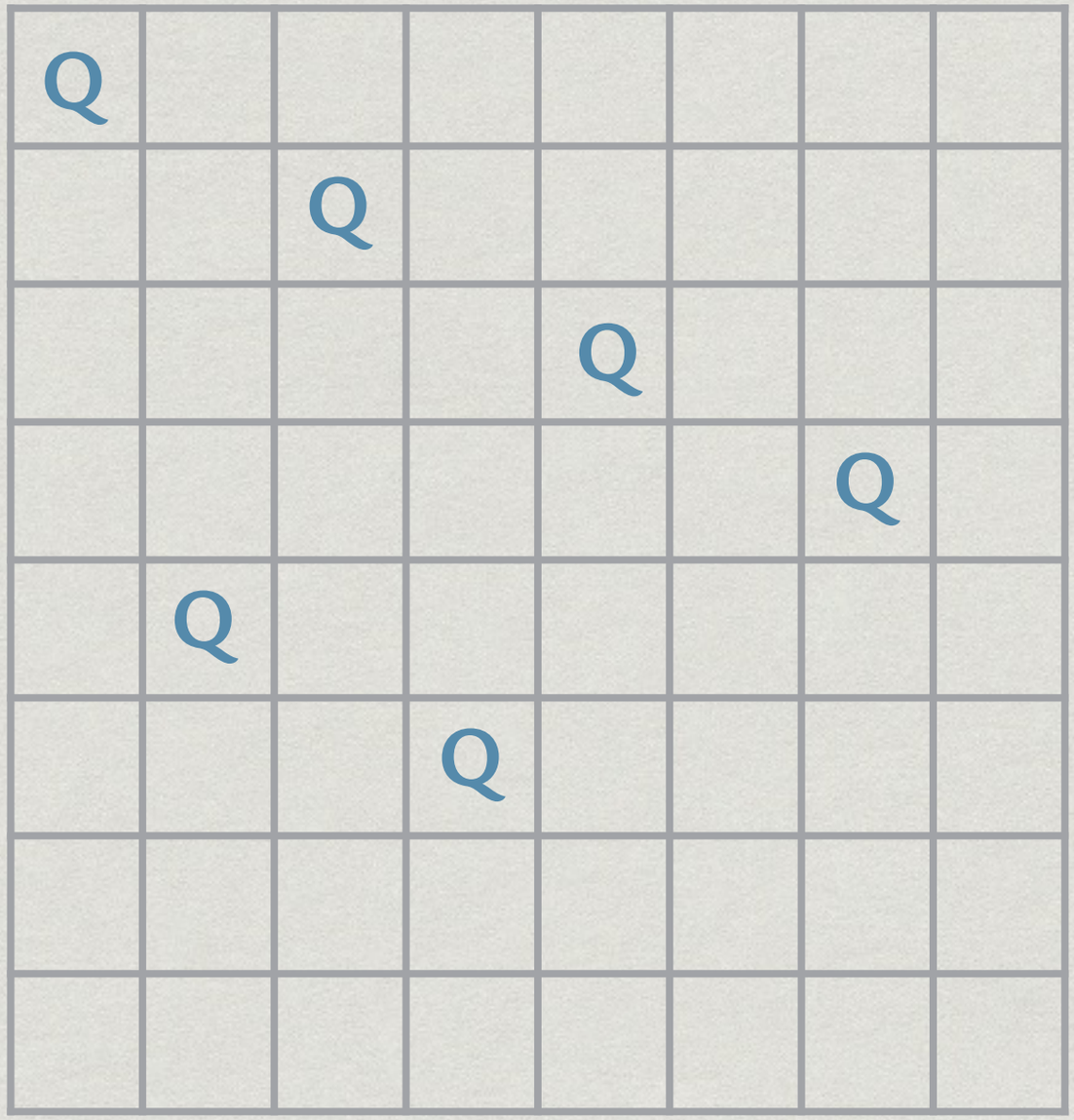
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



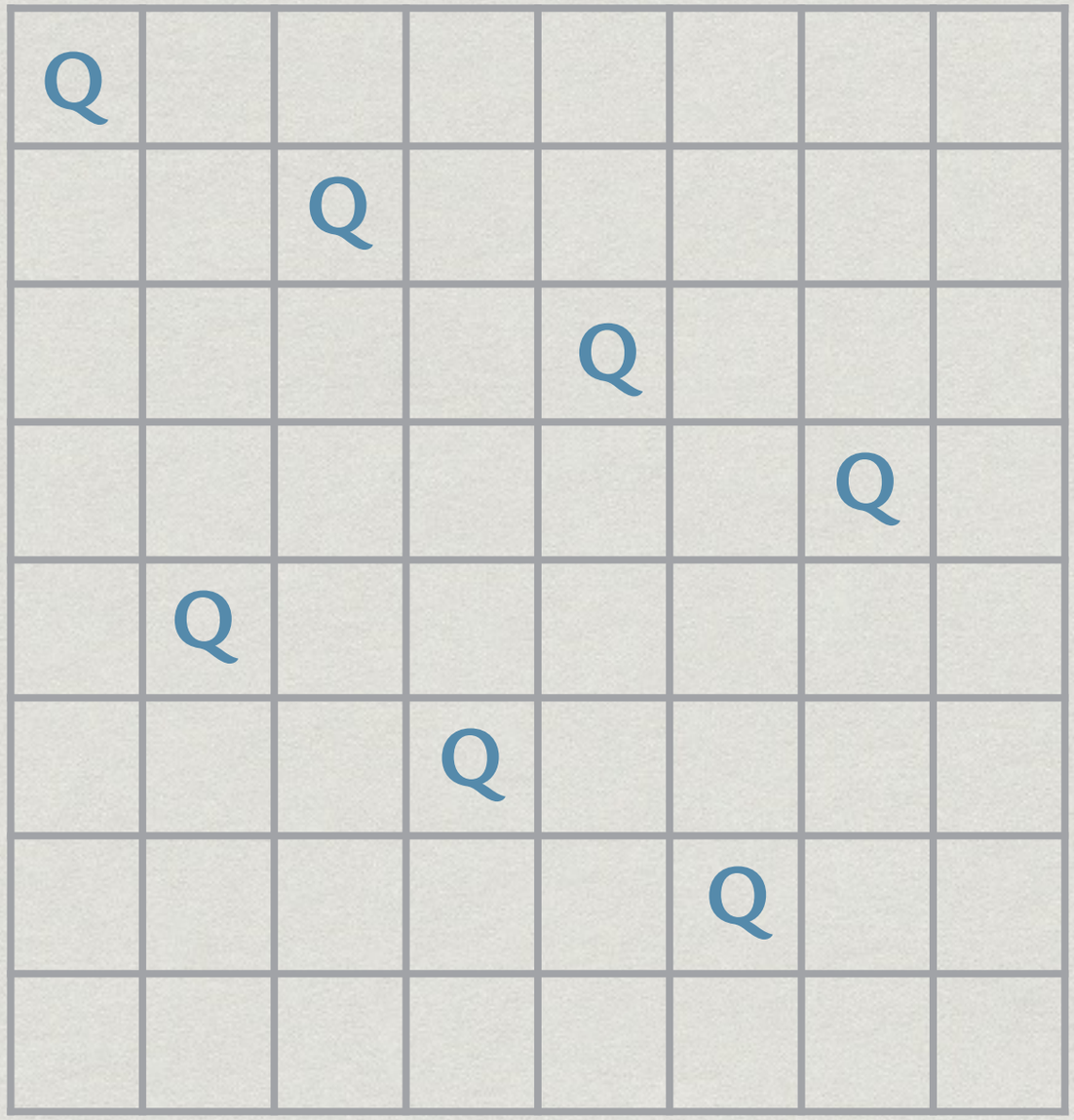
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



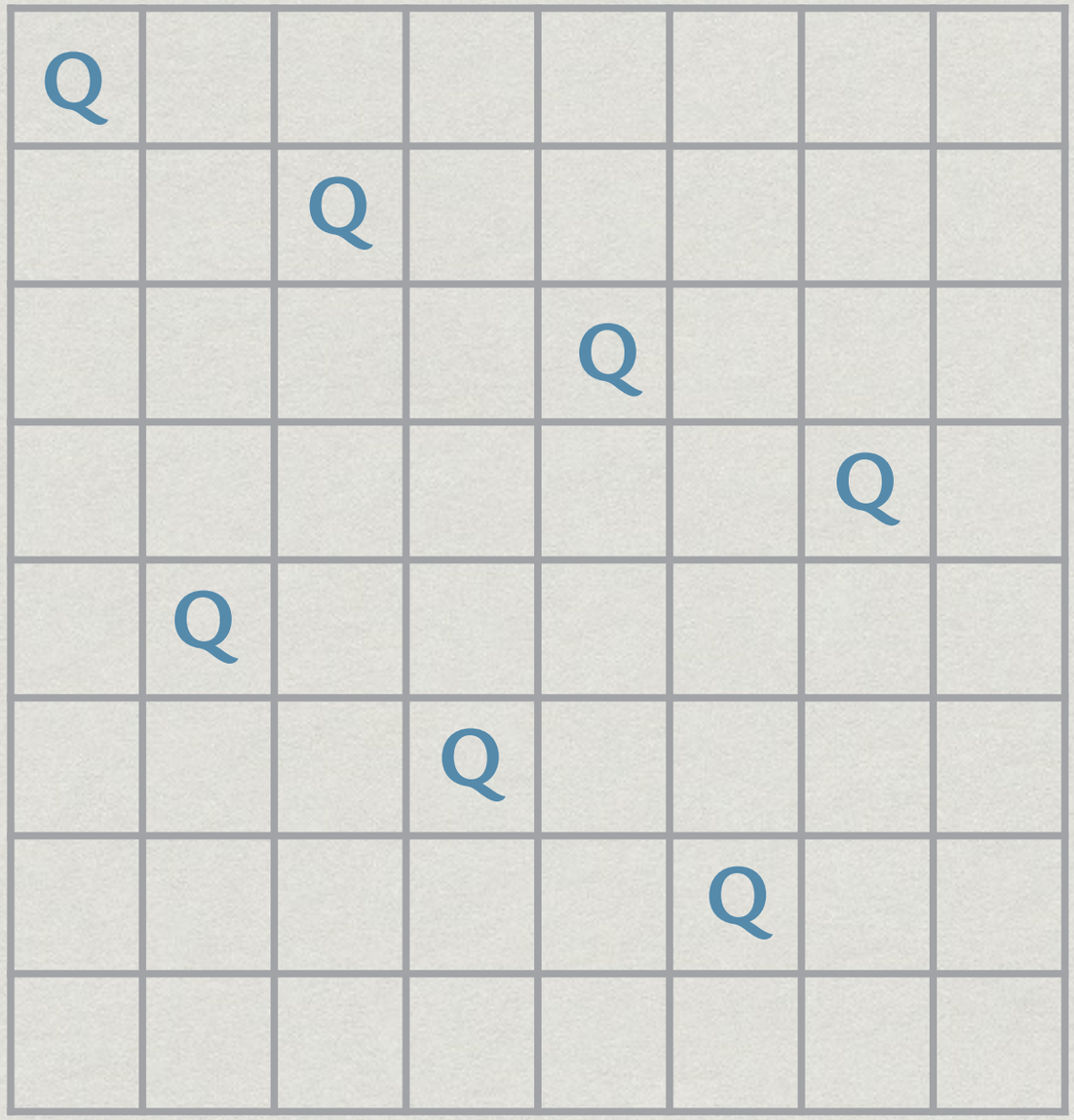
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner



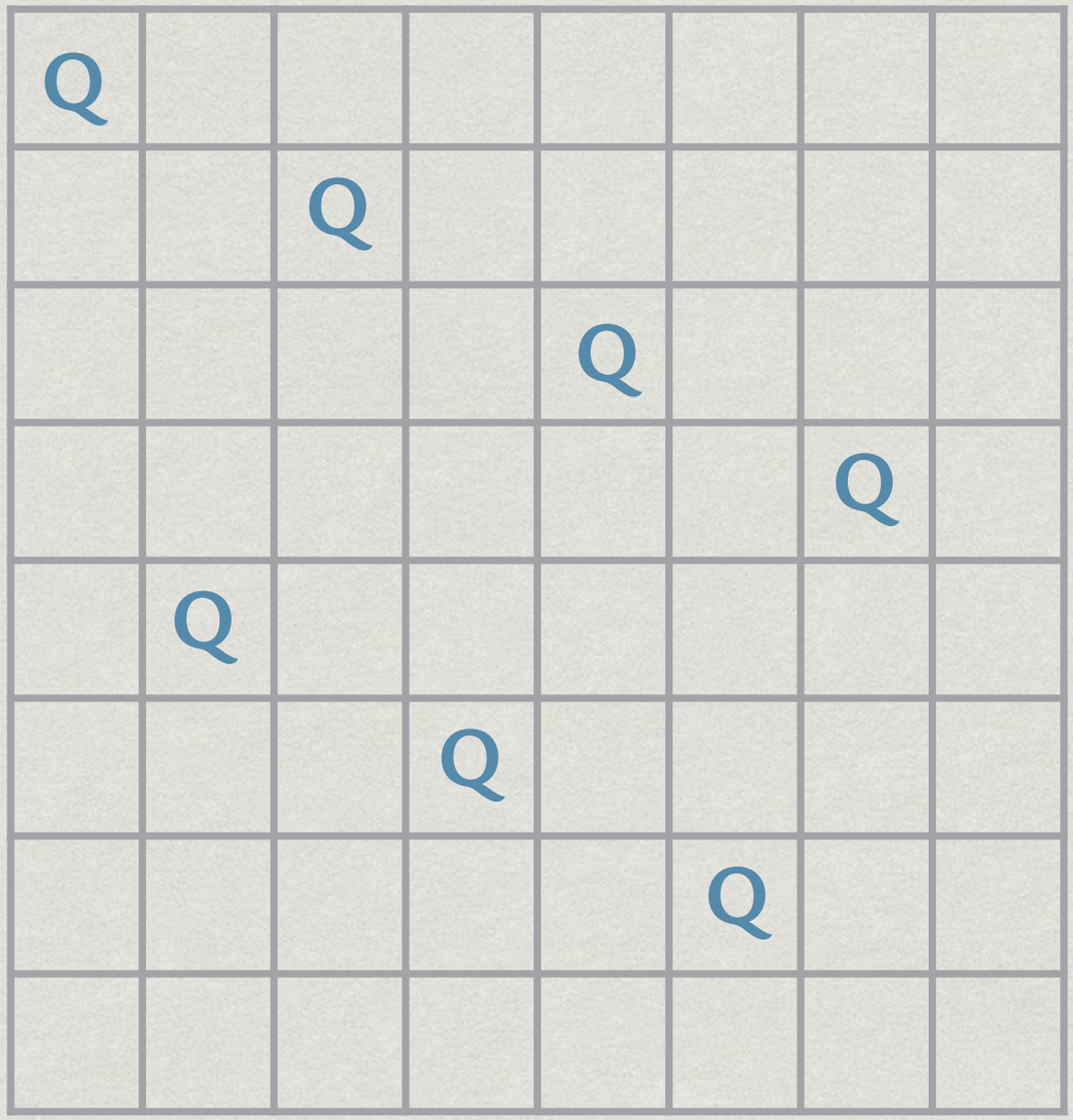
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner
- After 7 queens, we get stuck



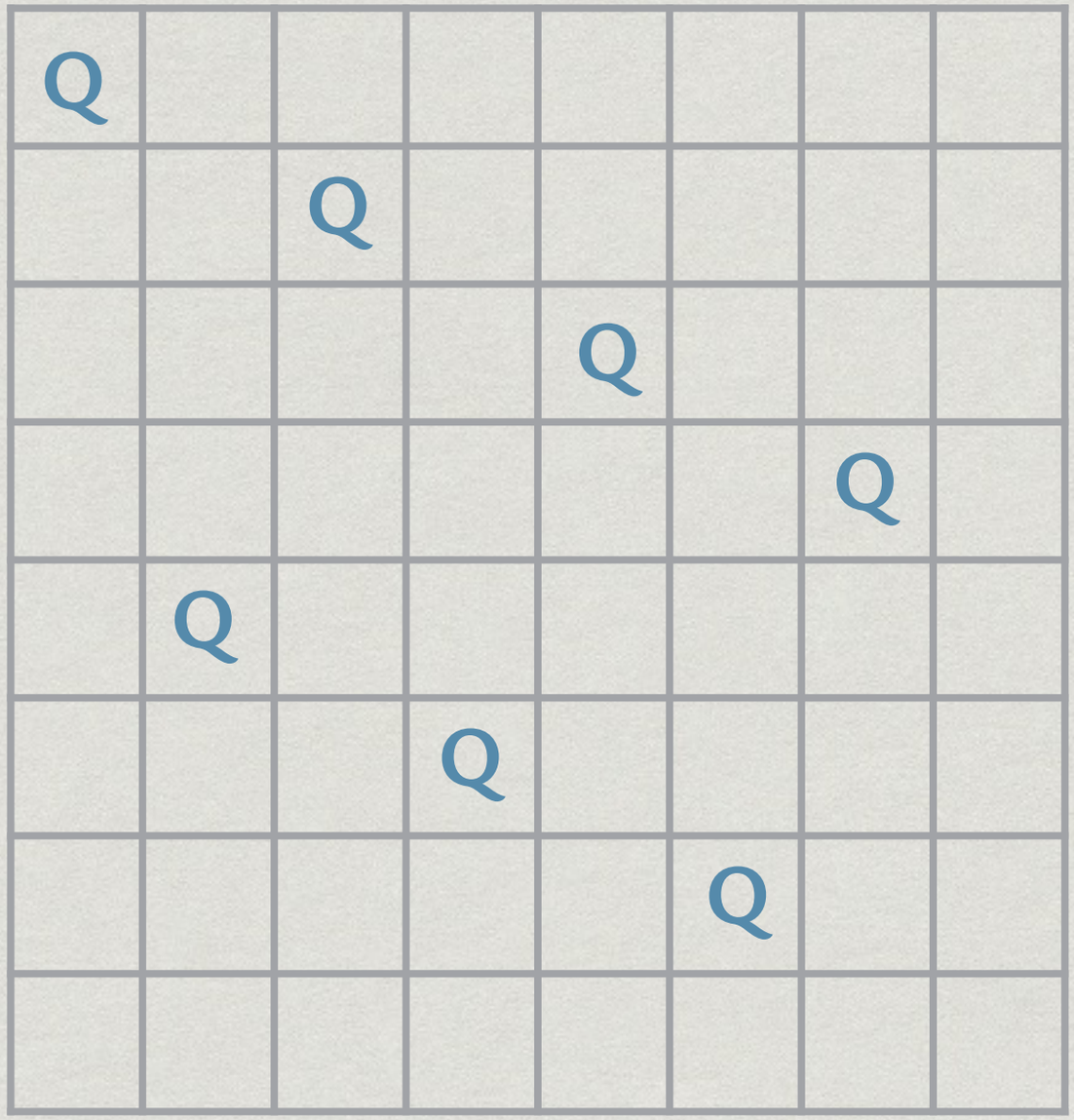
Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner
- After 7 queens, we get stuck
- Try other positions for queen 7



Backtracking …

- Suppose we have an 8 x 8 board and we start at the top left corner
- After 7 queens, we get stuck
- Try other positions for queen 7
- Go back and try other positions for queen 6 …



Searching for a solution

- Arrangement of queens is a list
  - Position `i` describes queen in row `i`
  - Value is column number
  - Previous arrangement is `[0,2,4,6,1,3,5]`

Searching for a solution ...

- `extend` — function to add queen `k+1` to current list of `k` queens
  - Should not be in same column as any earlier queen
  - Should not be on same diagonals — values on a diagonal agree on `r+c` or `r-c`

Searching for a solution …

Searching for a solution ...

```
queens n = (iterate extend [[]])!!(n+1)
```

Searching for a solution …

```
queens n = (iterate extend [[]])!!(n+1)
```

- All arrangements of n queens on n x n board

Searching for a solution …

```
queens n = (iterate extend [[]])!!(n+1)
```

- All arrangements of `n` queens on `n x n` board
- Extract the first such arrangement

Searching for a solution …

```
queens n = (iterate extend [[]])!!(n+1)
```

- All arrangements of n queens on n x n board
- Extract the first such arrangement

```
queensone n =
        head ((iterate extend [])!!(n+1))
```

Searching for a solution …

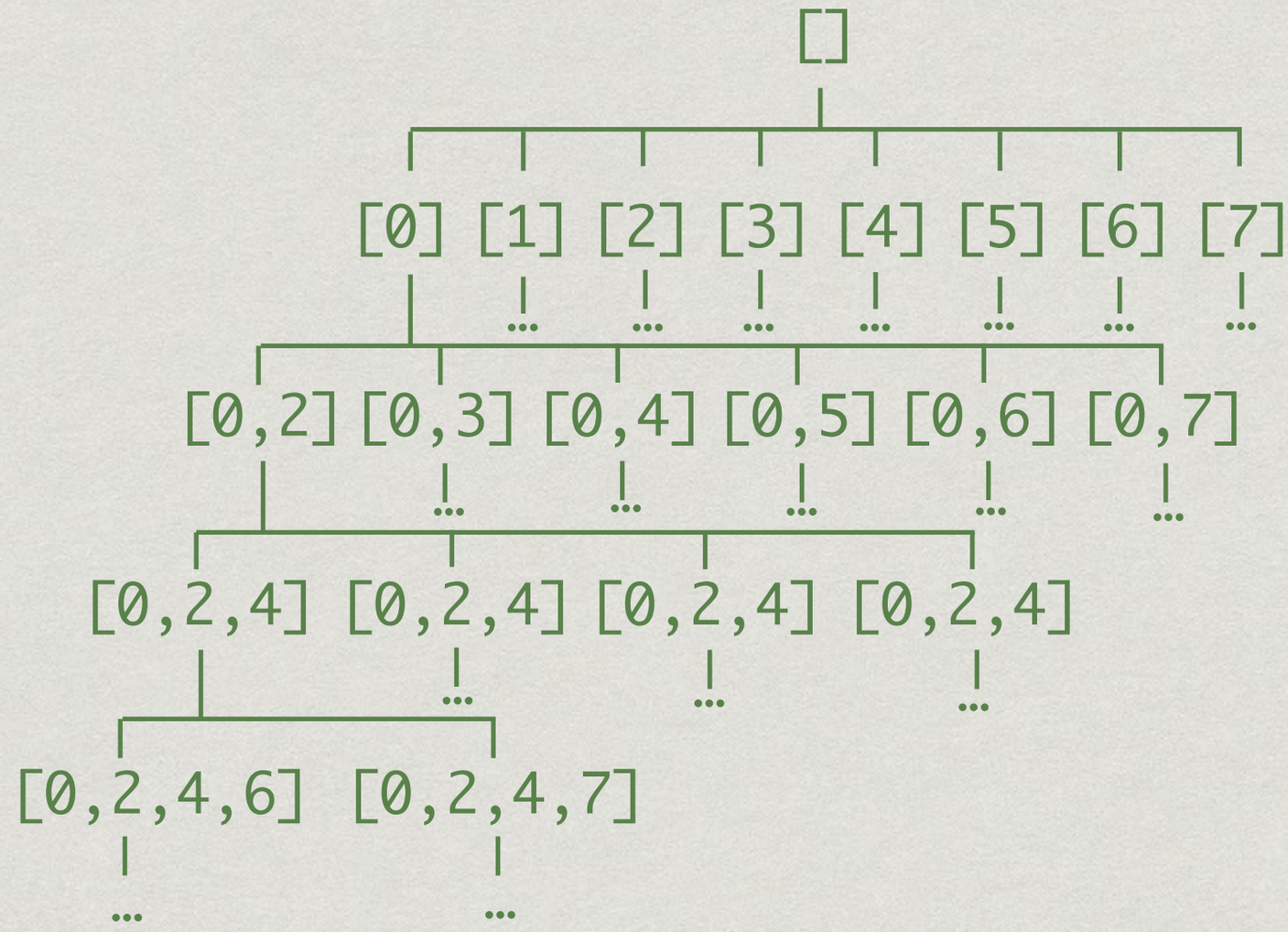
```
queens n = (iterate extend [[]])!!(n+1)
```

- All arrangements of n queens on n x n board
- Extract the first such arrangement

```
queensone n =
        head ((iterate extend [])!!(n+1))
```

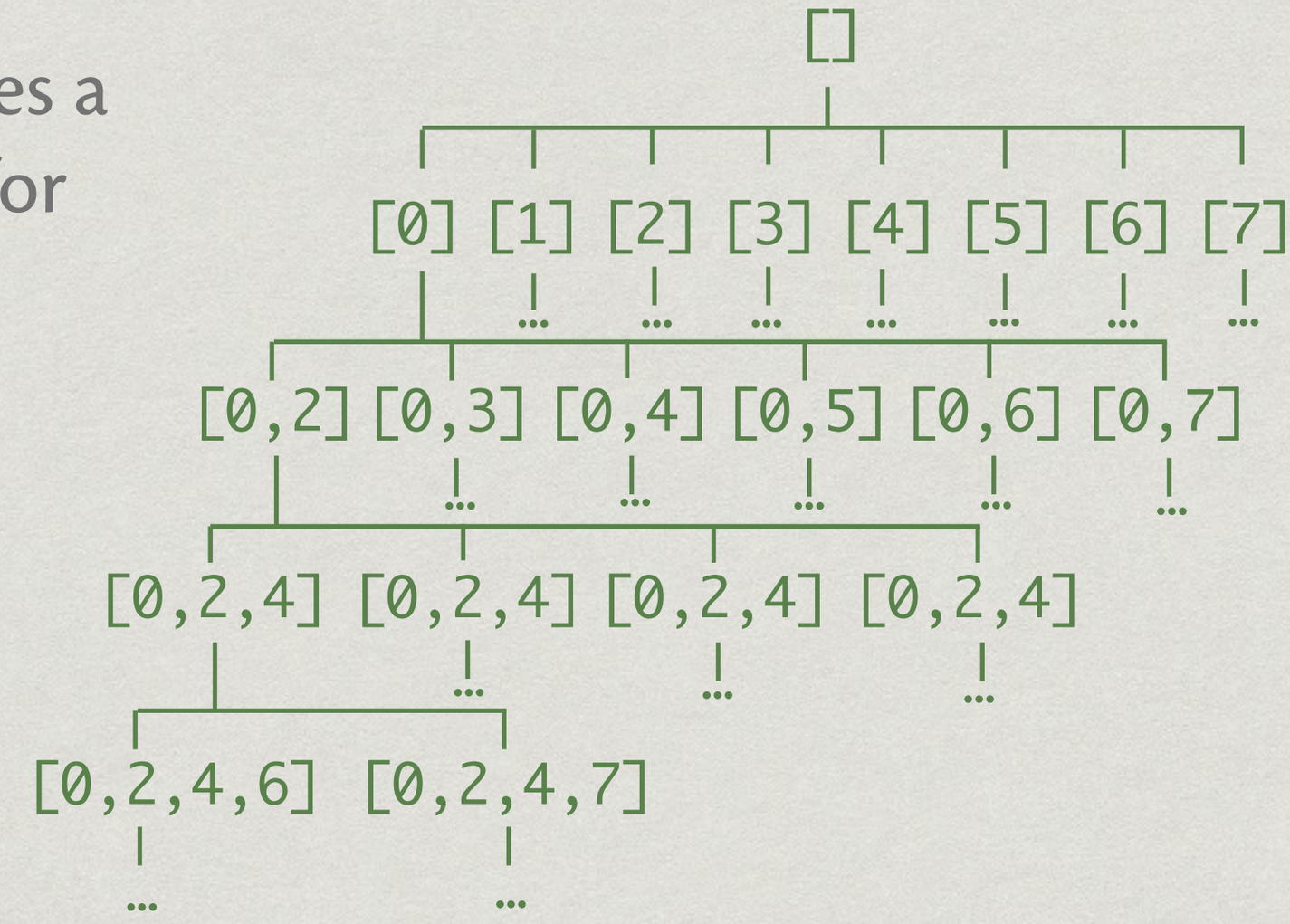
- Arrangements of k queens that have no extensions die out automatically

Laziness and efficiency



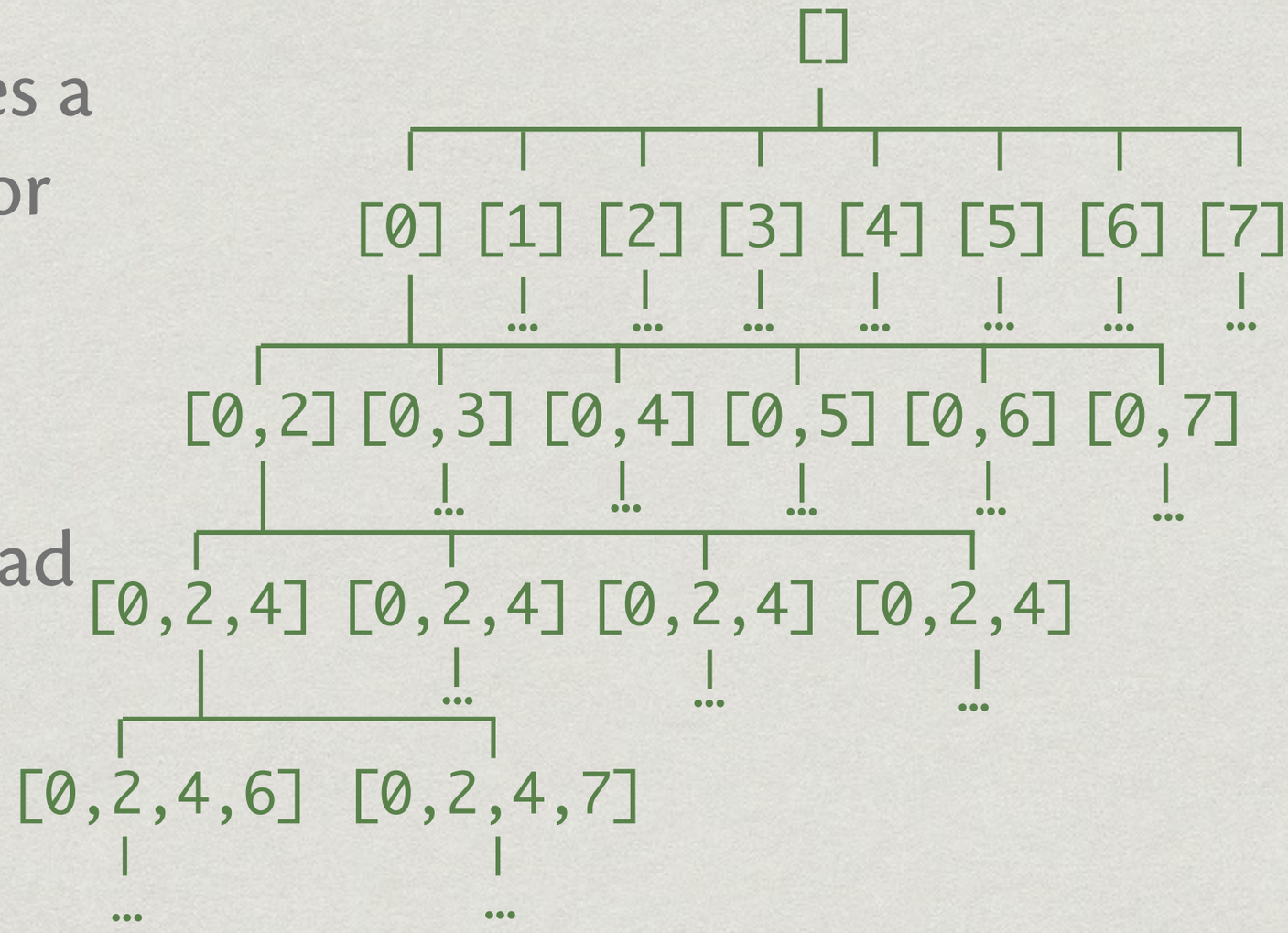
Laziness and efficiency

- Our strategy describes a breadth first search for solutions: inefficient



Laziness and efficiency

- Our strategy describes a breadth first search for solutions: inefficient
- Lazy evaluation of head stops with left most solution —depth first search!



Summary

- Infinite lists are a useful way to think about search problems
- Instead of backtracking, iteratively generate all valid solutions upto a desired depth
- Lazy evaluation converts inefficient breadth-first search to depth-first evaluation of leftmost solution

Polymorphism

- Functions that only look at the structure of a list work on lists of any type

```
head :: [a] -> a

length :: [a] -> Int

reverse :: [a] -> [a]

take :: Int -> [a] -> [a]
```

Sorting?

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?

- Can we sort any type of list?

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?

- Can we sort any type of list?

- What about a list of functions
  `[factorial, (+3), (*5)] :: [Int -> Int]`

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?
- Can we sort any type of list?
- What about a list of functions
  `[factorial, (+3), (*5)] :: [Int -> Int]`
- How do we compare `f < g` for functions?

Type classes

Type classes

- Want to assign a type as follows

Type classes

- Want to assign a type as follows

  `quicksort :: [a] -> [a]` provided we can compare values of type *a*

Type classes

- Want to assign a type as follows

  `quicksort :: [a] -> [a]` provided we can compare values of type *a*

- A type class is a collection of types with a required property

Type classes

- Want to assign a type as follows

  `quicksort :: [a] -> [a]` provided we can compare values of type `a`

- A type class is a collection of types with a required property

- The type class `Ord` contains all types whose values can be compared

Type classes …

Type classes …

- `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`

Type classes …

- `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`
  - If `Ord t`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`

Type classes …

- `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`
  - If `Ord t`, then <, <=, >, >=, ==, /= are defined for `t`
- We now write

Type classes …

- `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`
  - If `Ord t`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`
- We now write

```
quicksort :: (Ord a) => [a] -> [a]
```

Type classes …

- `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`
  - If `Ord t`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`
- We now write

```
quicksort :: (Ord a) => [a] -> [a]
```

  - If `a` is in `Ord`, `quicksort` is of type `[a] -> [a]`

What about elem?

What about elem?

```
elem x [] = False
elem x (y:ys)
  | x == y = True
  | otherwise = elem x ys
```

What about elem?

```
elem x [] = False
elem x (y:ys)
   | x == y = True
   | otherwise = elem x ys
```

- Consider the list

```
funclist =
    [factorial, (+3), (*5)] :: [Int -> Int]
```

What about elem?

```
elem x [] = False
elem x (y:ys)
   | x == y = True
   | otherwise = elem x ys
```

- Consider the list

```
funclist =
    [factorial, (+3), (*5)] :: [Int -> Int]
```

- How to evaluate `elem f funclist`?

Equality

Equality

- Can we check `f == g` for functions?

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?
  - Recall that `f x` may not terminate

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?
  - Recall that `f x` may not terminate

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

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?
  - Recall that `f x` may not terminate

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

`factorial (-1)`?

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?
  - Recall that `f x` may not terminate

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

```
factorial (-1)?
```

- `f == g` implies for all `x`, `f x` terminates iff `g x` does

Equality …

Equality …

- Can we write a function

Equality …

- Can we write a function

```
halting :: (a -> b) -> a -> Bool
```

Equality …

- Can we write a function

  ```
  halting :: (a -> b) -> a -> Bool
  ```

  such that `halting f x` is `True` iff `f x` terminates?

Equality …

- Can we write a function

  ```
  halting :: (a -> b) -> a -> Bool
  ```

  such that `halting f x` is `True` iff `f x` terminates?

- Alan Turing proved such a function cannot be effectively computed

Equality …

- Can we write a function

  ```
  halting :: (a -> b) -> a -> Bool
  ```

  such that `halting f x` is `True` iff `f x` terminates?

- Alan Turing proved such a function cannot be effectively computed

- Hence, equality over functions is not computable

The type class Eq

The type class Eq

- `Eq a` holds if `==`, `/=` are defined on `a`

The type class Eq

- `Eq a` holds if `==`, `/=` are defined on `a`

```
elem :: (Eq a) => a -> [a] -> Bool
```

The type class Eq

- `Eq a` holds if `==`, `/=` are defined on `a`

```
elem :: (Eq a) => a -> [a] -> Bool
```

- If `Eq a`, `Eq b` then `Eq [a]`, `Eq (a,b)`

The type class Eq

- `Eq a` holds if `==`, `/=` are defined on `a`

```
elem :: (Eq a) => a -> [a] -> Bool
```

- If `Eq a`, `Eq b` then `Eq [a]`, `Eq (a,b)`

- Cannot extend `Eq a`, `Eq b` to `a -> b`

The type class Ord

The type class Ord

- `Ord a` holds if `<`, `<=`, `>`, `>=`, `==`, `/=` are defined on `a`

The type class Ord

- `Ord a` holds if <, <=, >, >=, ==, /= are defined on `a`
- If `Ord a` then `Ord [a]` — lexicographic (dictionary) order

The type class Ord

- `Ord a` holds if <, <=, >, >=, ==, /= are defined on `a`
- If `Ord a` then `Ord [a]` — lexicographic (dictionary) order
- If `Ord a`, `Ord b` then `Ord (a,b)`

The type class Ord

- `Ord a` holds if `<`, `<=`, `>`, `>=`, `==`, `/=` are defined on `a`
- If `Ord a` then `Ord [a]` — lexicographic (dictionary) order
- If `Ord a`, `Ord b` then `Ord (a,b)`
- Cannot extend `Ord a`, `Ord b` to `a -> b`

The type class Num

The type class Num

- Recall the function `sum`

The type class Num

- Recall the function `sum`

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

The type class Num

- Recall the function `sum`

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

- `sum` requires `+` to be defined on list elements

The type class Num

- Recall the function `sum`

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

- `sum` requires `+` to be defined on list elements

- `Num a` says `a` is a number, supports basic arithmetic operations

The type class Num

- Recall the function `sum`

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

- `sum` requires `+` to be defined on list elements

- `Num a` says `a` is a number, supports basic arithmetic operations

```
sum :: (Num a) => [a] -> a
```

Some other type classes

Some other type classes

- `Integral`, `Frac` : subclasses of `Num`

Some other type classes

- `Integral`, `Frac` : subclasses of `Num`
- `Show` : values that can be displayed

Some other type classes

- `Integral`, `Frac` : subclasses of `Num`
- `Show` : values that can be displayed
  - Requires a function `show` to display a value

Some other type classes

- `Integral`, `Frac` : subclasses of `Num`
- Show : values that can be displayed
  - Requires a function `show` to display a value
  - For example, function types `a -> b` do not belong to Show

Signatures

Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support

Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support
  - `Ord` is defined by the signature `<`, `<=`, `>`, `>=`, `==`, `/=`

Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support
  - `Ord` is defined by the signature `<`, `<=`, `>`, `>=`, `==`, `/=`
  - `Eq` is defined by the signature `==`, `/=`

Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support
  - `Ord` is defined by the signature `<`, `<=`, `>`, `>=`, `==`, `/=`
  - `Eq` is defined by the signature `==`, `/=`
  - …

Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support
  - `Ord` is defined by the signature `<`, `<=`, `>`, `>=`, `==`, `/=`
  - `Eq` is defined by the signature `==`, `/=`
  - …
- Later we will see how to define our own type classes and add types to a type class

Summary

- A type class is a collection of types satisfying additional properties over their values
  - Check for equality, compare magnitude …
- Additional properties are defined in terms of signatures — additional functions that must be defined on all values of the type
- Can use type classes to specify conditionally polymorphic types for functions