Reminders

- Assignment 09 is due **Monday, December 4 at 9:00pm**, the last day of classes
- Office hours will be rescheduled after classes end

Clicker Question - History

Who invented the lambda calculus?

A John von Neumann  
B Alonzo Church  
C Kurt Gödel  
D Grace Hopper  
E John McCarthy
The n-queens puzzle

The n-queens puzzle is the problem of placing n queens on an n x n chessboard, such that no 2 queens attack each other. A queen can attack another queen if they are both on the same row, column or diagonal. In this tutorial, we will implement a program that can solve the n-queens problem for any n.

We can use graphs to represent such puzzles. Each node in the graph represents a particular state of the puzzle, and an edge from one state to the next represents a possible next step. Finding a solution is then equivalent to finding a path between the initial state of the puzzle and a valid final state.

We can thus adapt the same backtracking algorithm used in find-route from lectures. First, let's define a state for the n-queens problem:

```scheme
;; A Queen is a (make-posn Nat Nat)

;; A State is a (listof Queen)

Given a State, a new State results from adding a queen to the next empty column.

Here is the empty state for n=4, and its neighbours:
Here is another example state, for \( n=5 \), and its neighbours:

![Example state and neighbours](image)

On the assignment, you are provided with a higher-order function `search` that performs the back-tracking algorithm on the implicit graph of states for a puzzle. `search` is called higher-order since it consumes 2 functions:

- `at-end?`, a predicate that determines if we have reached a valid final state
- `neighbours`, a function that produces all the next steps from the given state

Here is the `search` function, slightly modified for the tutorial:

```scheme
(define (search at-end? neighbours a-state n)
    (local
        [(define (find-route a-state n) . . . )
         (define (find-route/list lostate n) . . . )
         (find-route a-state n)))
```

This is `find-route` for explicit graphs:

```scheme
;; (find-route orig dest G) finds route from orig to dest in G, if one exists
;; find-route: Node Node Graph → (anyof (listof Node) false)
(define (find-route orig dest G)
    (cond
        [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                      (define (find-route/list lostate n) . . . )
                      (find-route a-state n)])]
```

This is `find-route` for explicit graphs:

```scheme
;; (find-route orig dest G) finds route from orig to dest in G, if one exists
;; find-route: Node Node Graph → (anyof (listof Node) false)
(define (find-route orig dest G)
    (cond
        [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                      (define route (find-route/list lostate n) . . . )
                      (find-route a-state n)])])
```
(define (find-route/list los dest G)
    (cond
        ; empty? los?
        [(empty? los) false]
        ; else?
        [else (local [(define route (find-route (first los) dest G))]
            (cond
                ; false? route?
                [(false? route) (find-route/list (rest los) dest G)]
                ; else?
                [else route]))])))
neighbours
We still need to write two more helper functions to complete find-route: solved? and neighbours.

Write the function, neighbours, that will consume a State, a-state, and a positive integer, n. It will produce a list of all the next possible States, for an n x n chessboard.

To find a state's neighbours, find all the positions in the next empty column in which a queen can be placed. Then construct a new state for each new queen that cannot attack any of the queens in the current state. These new states are the neighbours and they may be produced in any order.

solved?
Now write the function, solved?, that will consume a State, a-state, and a positive integer, n. It will determine if a-state is a solution to the n-queens problem. We can assume that a-state is a valid partial solution.

solve-n-queens
It's time to put it all together. Write a function called solve-n-queens that consumes a positive integer n and produces a list of queens that solve the n-queens puzzle. This should simply be a wrapper function for another function you have already written.