PREPARING FOR A DATE:
WHAT SITUATIONS MIGHT I PREPARE FOR?
1) MEDICAL EMERGENCY
2) DANCING
3) FOOD TOO EXPENSIVE

OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?
1) SNAKEBITE
2) LIGHTNING STRIKE
3) FALL FROM CHAIR

HMM, WHICH SNAKES ARE DANGEROUS? LET'S SEE...
DANGER
1) a) CORN SNAKE
b) CARTER SNAKE
2) COPPERHEAD

THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED
AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.

I'M HERE TO PICK YOU UP. YOU'RE NOT DRESSED?

BY LCD, THE INLAND TAIPAN HAS THE DEADLIEST VENOM OF ANY SNAKE!

I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

https://xkcd.com/761/
original input: g, 2, 17
  recursive input: g, 5, 17 → recursive output: false
  recursive input: g, 6, 17 → recursive output: true

original output: true
Define procedure `find_path` that takes a graph, an origin vertex, and a destination vertex, and returns origin-to-destination path if exists.

Path might not exist so use `option`
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree

Algorithm thinks graph is a tree
Applying tree search to non-tree

Algorithm thinks graph is a tree — will visit vertices many times.
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree
Applying tree search to non-tree

If there is a cycle, algorithm will continue forever. :(
Applying tree search to non-tree

If there is a cycle, algorithm will continue forever. :(  
Need a way to keep track of already-visited vertices.
Applying tree search to non-tree

If there is a \textit{cycle}, algorithm will continue forever. :(  
Need a way to keep track of already-visited vertices.  
Use a \textit{set}.  

\[ 
\begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \\
\text{4} \\
\text{5} \\
\text{6} \\
\text{7} \\
\text{8} \\
\text{9}
\end{array} \]
module type GAME =
sig
  type which_player = P1 | P2
  type status =
    | Win of which_player | Draw | Ongoing of which_player
  type state
  type move

  ... 
  val initial_state : state
  val legal_moves : state -> move list
  val game_status : state -> status
  val next_state : state -> move -> state
end
module Nim =
structural
  type which_player = P1 | P2
  type status =
    Win of which_player | Draw | Ongoing of which_player
  type state = ...
  type move = int
  let initial_state = ... 21 ...
  let legal_moves = ...
  let other_player = ...
  let game_status = ...
  let next_state = ...
end

Traditionally, Nim starts with a pile of 21 stones. Players take turns removing 1, 2 or 3. Player who takes the last stone loses. But why start with 21?
module Nim = functor (I: sig val initial: int end) ->
struct
  type which_player = P1 | P2
  type status =
    Win of which_player | Draw | Ongoing of which_player
  type state = ...
  type move = int
  let initial_state = ... I.initial ...
  let legal_moves = ...
  let other_player = ...
  let game_status = ...
  let next_state = ...
end

module NimGame = Nim(struct let initial = 22 end)
module HumanPlayer = functor (Game: GAME) ->

struct
    module PlayerGame = Game
    open PlayerGame

    let rec next_move s =
        try
            let m = move_of_string (read_line ()) in

            (* TODO: replace the below expression (between the if and then) with the proper functionality *)
            if List.exists (function x -> x=m) (legal_moves s) then m
            else
                let () = print_endline "Illegal move."
                in next_move s
        with |
            | End_of_file -> failwith "exiting."
            | Failure message -> print_endline message ; next_move s
    end
module Referee = functor
  (Game : GAME)
  (Player1: PLAYER)
  (Player2: PLAYER) ->
  struct
    module CurrentGame = Game

    let play_game () = function ...

  end

module Ref = Referee
  (NimGame)
  (HumanPlayer(NimGame))
  (HumanPlayer(NimGame));;

Ref.play_game()
module Referee = functor
  (Game : GAME)
  (Player1: PLAYER with module PlayerGame = Game)
  (Player2: PLAYER with module PlayerGame = Game) ->
struct
  module CurrentGame = Game

  let play_game () = function ...

end