# This is a pretty naive implementation of Tic-Tac-Toe meant to
demonstrate the minimax algorithm. Note that it takes awhile to
start up - 30 seconds on a 2010 mac book air. Also note that the
Ruby isn’t all that great as Ruby, but I don’t care as long as it
illustrates the algorithm well.

You can see that there’s a hard-coded assumption that the AI is the
X player, means that no human will ever be able to win. I’m positive
that is only a small taste of what is to come, once robots take
over the world.

class GameState
  attr_accessor :current_player, :board, :moves, :rank

  def initialize(current_player, board)
    self.current_player = current_player
    self.board = board
    self.moves = []
  end

  def rank
    @rank ||= final_state_rank || intermediate_state_rank
  end

  # this is only ever called when it’s the AI’s (the X player) turn
  def next_move
    moves.max{ |a, b| a.rank <=> b.rank }
  end

  def final_state_rank
    if final_state?
      return 0 if draw?
      winner == "X" ? 1 : -1
    end
  end

  def final_state?
    winner || draw?
  end

  def draw?
    board.compact.size == 9 && winner.nil?
  end

  def intermediate_state_rank
    # recursion, baby
    ranks = moves.collect{ |game_state| game_state.rank }
    if current_player == 'X'
      ranks.max
    else
      ranks.min
    end
  end

  def winner
    @winner ||= [
      # horizontal wins
      [0, 1, 2],
      [3, 4, 5],
      [6, 7, 8],
      [0, 3, 6],
      [1, 4, 7],
      [2, 5, 8],
      [0, 4, 8],
      [2, 4, 6],
    ]
```ruby
# vertical wins
[0, 3, 6],
[1, 4, 7],
[2, 5, 8],

# diagonal wins
[0, 4, 8],
[6, 4, 2]
].collect { |positions|
  ( board[positions[0]] == board[positions[1]] &&
    board[positions[1]] == board[positions[2]] &&
    board[positions[0]]) || nil
}.compact.first

end

class GameTree
  def generate
    initial_game_state = GameState.new('X', Array.new(9))
    generate_moves(initial_game_state)
    initial_game_state
  end

  def generate_moves(game_state)
    next_player = (game_state.current_player == 'X' ? 'O' : 'X')
    game_state.board.each_with_index do |player_at_position, position|
      unless player_at_position
        next_board = game_state.board.dup
        next_board[position] = game_state.current_player

        next_game_state = GameState.new(next_player, next_board)
        game_state.moves << next_game_state
        generate_moves(next_game_state)
      end
    end
  end
end

class Game
  def initialize
    @game_state = @initial_game_state = GameTree.new.generate
  end

  def turn
    if @game_state.final_state?
      describe_final_game_state
      puts 'Play again? y/n'
      answer = gets
      if answer.downcase.strip == 'y'
        @game_state = @initial_game_state
        turn
      else
      exit
    end

    if @game_state.current_player == 'X'
      puts '
===============
X's move:'
      render_board
      turn
    else
      if @game_state.current_player == 'X'
        puts '
==============="
X's move:'
        render_board
      end
    end
```
get_human_move
puts "The result of your move:"
render_board
puts ""
end
end
def render_board
  output = ""
  0.upto(8) do |position|
    output << " #{game_state.board[position] || position} "
    case position % 3
      when 0, 1 then output << "|"
      when 2 then output << "\n-----------\n" unless position == 8
    end
  end
  puts output
end
def get_human_move
  puts "Enter square # to place your 'O' in:"
  position = gets
  move = @game_state.moves.find{|game_state| game_state.board[position.to_i] == 'O'}
  if move
    @game_state = move
  else
    puts "That's not a valid move"
    get_human_move
  end
end
def describe_final_game_state
  if @game_state.draw?
    puts "It was a draw!"
  elsif @game_state.winner == 'X'
    puts "X won!"
  else
    puts "O won!"
  end
end
Game.new.turn

# ADDED COMMENTARY CS3342 160203
# 3 class definitions, lines 14 (GameState), 80 (GameTree), and 102 (Game).
# class Game is the main. It is invoked on line 172 where a new object of
# this type is created and then the method turn invoked.
# ** GENERAL NOTE ON NAMES **
# "name characters" = [a-zA-Z0-9_]
# <local block variable name> = [a-z] <name characters>*
# <method name> = [a-z] <name characters>*
# <object instance variable name> = @ <name characters>*
# <class variable name> = @@ <name characters>*
# <constant name> = [A-Z] <name characters>*
# classes and modules are viewed as constants
# <global variable name> = $ <name characters>+
Game.new.turn, the Game.new triggers Games initialize method (line 103)
which does Gametree.new.generate to trigger generate (line 81), which
creates a new GameState triggering initialize in GameState (line 17).

after initial_game_state is initialized, generate_moves is invoked (line
83) and initial_game_state is returned (line 84) which becomes
@game_state in the class Game.

looking at generate_moves (line 87) we see next_player is updated by a
C-style conditional expression that uses current_player in game_state.

note current_player was created on line 15 of GameState using
attr_accessor (which takes a symbol (:current_player), creates a variable
@current_player), a method that reads the variable (current_player), and
a method that writes the variable (current_player=). [note that
attr_accessor is a method that a class evaluates upon creation.] without
the read and write methods, the instance variable wouldn't be easily
accessible outside the object. so, attr_accessor is a method that creates
methods.

line 89: the board field of game_state is an array. each_with_index
is taking the parameterized block do |player_at ...end end and executing
the block on each of the value,index pairs. unless is like if not.
dup makes a shallow copy of the array.
line 95: << adds to the end of the array moves.
line 96: recursive call to generate_moves

having built the game_state, line 172 then invokes the turn method (line
107). turn first checks to see if the game is over by invoking
GameState’s final_state? (line 39), which invokes winner (line 57)
line 58 sets ||= @winner if it is nil (unset) or false. collect
(line 72) creates a new array from applying the block to each of
the positions (3 value arrays in this case). (line 75) false || nil
is nil (true || nil is true). (line 76: compact removes nil entries
from array. first on an empty array returns nil.

other oddities. 0.upto(8) is another way to create an enumerator that
can then be invoked with a block.

line 139 gives us an example of a case statement. note the use of
unless on line 141.

line 33 shows us if end. line 50 shows us if else end. line 162
shows us if elsif else end.

line 136: note output is not a keyword. here we make it a string,
then on line 138 << concatenates to it. note #{} being used to
evaluate an expression in a string ‘object’.