1. Board States

```
# make changes to the value n and k
class Board:
  def __init__(self, n ,k):
    self.board = [];
    self.rows = int(n);
    self.columns = int(k);
    self.whose_move = "userone";
    self.last_whose_move = [];
    self.last_which_row = [];
    self.last_which_column = [];
    self.player = 0;

    for rows_count in range(self.rows):
      rows_temp = [];
      for columns_count in range(self.columns):
        rows_temp.append(0);
      self.board.append(rows_temp);

  def generate_moves(self):
    possible_moves = [];
    possible_moves_column = [];

    for columns_count in range(0, self.columns , 1):
      for rows_count in range(0, self.rows, 1):
        if self.board[rows_count][columns_count] == 0:
          possible_moves.append(str(rows_count) + str(columns_count));
          possible_moves_column.append(columns_count);
```

```
  return possible_moves_column;

def make_move_initial(self, row, column):
  if self.whose_move == "userone":
    self.board[row][column] = 1;
    self.whose_move = "usertwo";
    self.player = 1;
    self.last_whose_move.append("userone");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

  elif self.whose_move == "usertwo":
    self.board[row][column] = 2;
    self.whose_move = "userone";
    self.player = 0;
    self.last_whose_move.append("usertwo");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

def make_move(self, move):
  move_added = False;
  for rows_count in range(self.rows-1, -1, -1):
    if self.board[rows_count][move] == 0:
      if self.whose_move == "userone":
        self.board[rows_count][move] = 1;
        self.whose_move = "usertwo";
        self.player = 1;
        self.last_whose_move.append("userone");
```

```
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      elif self.whose_move == "usertwo":
        self.board[rows_count][move] = 2;
        self.whose_move = "userone";
        self.player = 0;
        self.last_whose_move.append("usertwo");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      move_added = True;
      break;

  def unmake_last_move(self):
    if len(self.last_whose_move) > 0 and len(self.last_which_row) > 0 and len(self.last_which_column) >
0:
      self.board[self.last_which_row[len(self.last_which_row) -
1]][self.last_which_column[len(self.last_which_column) - 1]] = 0;
      self.whose_move = self.last_whose_move[len(self.last_whose_move) - 1];
      self.last_whose_move.pop();
      if self.player == 0:
        self.player = 1;
      else:
        self.player = 0;
      self.last_which_row.pop();
      self.last_which_column.pop();
      #print("Move undo done");
```

```
def last_move_won(self):
  #print(len(self.last_which_row) -1);
  if len(self.last_whose_move) > 0:
    last_rowmove_made = self.last_which_row[len(self.last_which_row) - 1];
    last_columnmove_made = self.last_which_column[len(self.last_which_column) - 1];
    last_whosemove_made = self.last_whose_move[len(self.last_whose_move) - 1];
    last_whosemove_made_number = -1;

    if last_whosemove_made == "userone":
      last_whosemove_made_number = 1;
    elif last_whosemove_made == "usertwo":
      last_whosemove_made_number = 2;

    #check the verticals if there is a possibility for a win.
    vertical_streak = 0;
    for rows_count in range(last_rowmove_made + (self.rows -1), last_rowmove_made - self.rows , -1):
      if rows_count >= 0 and rows_count <= (self.rows-1):
        if self.board[rows_count][last_columnmove_made] == last_whosemove_made_number:
          vertical_streak += 1;
          if vertical_streak == self.rows:
            break;
        else:
          vertical_streak = 0;

    if vertical_streak == self.rows:
      return True;

    #check the horizontal streak if there is a possibility for a win.
    horizontal_streak = 0;
```

```
    for columns_count in range(last_columnmove_made - (self.rows-1), last_columnmove_made +
self.rows, 1):
      if columns_count >= 0 and columns_count <= (self.columns-1):
       if self.board[last_rowmove_made][columns_count] == last_whosemove_made_number:
         horizontal_streak += 1;
         if horizontal_streak == self.rows:
           break;
       else:
         horizontal_streak = 0;

    if horizontal_streak == self.rows:
     return True;

    #check the diagonal streak if there is a possibility for a win. - diagonal left to right
    diagonal_left_to_right_streak = 0;
    columns_count = last_columnmove_made - (self.rows-1);

    for rows_count in range(last_rowmove_made + (self.rows-1), last_rowmove_made - self.rows, -1):
      if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
       if self.board[rows_count][columns_count] == last_whosemove_made_number:
         diagonal_left_to_right_streak += 1;
         if diagonal_left_to_right_streak == self.rows:
           break;
       else:
         diagonal_left_to_right_streak = 0;
      columns_count += 1;

    if diagonal_left_to_right_streak == self.rows:
```

```
        return True;

    #check the diagonal streak if there is a posssibility for a win - diagonal right to left.
    diagonal_right_to_left_streak = 0;
    columns_count = last_columnmove_made - (self.rows-1);

    for rows_count in range(last_rowmove_made - (self.rows-1), last_rowmove_made + self.rows, 1):
      if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
        if self.board[rows_count][columns_count] == last_whosemove_made_number:
          diagonal_right_to_left_streak += 1;
          if diagonal_right_to_left_streak == self.rows:
            break;
        else:
          diagonal_right_to_left_streak = 0;
      columns_count += 1;

    if diagonal_right_to_left_streak ==  self.rows:
      return True;
    else:
      return False;
  else:
    return False;

 def __str__(self):
  print_string = "";
  for rows in range(self.rows):
    column_string = "";
    for columns in range(self.columns):
```

```
        if self.board[rows][columns] == 0:
          column_string = column_string + ".";
        elif self.board[rows][columns] == 1:
          column_string = column_string + "w";
        elif self.board[rows][columns] == 2:
          column_string = column_string + "b";
       print_string = print_string + column_string;

    return str(print_string);
```

2. Player File

```
import random;
import time;

class Player:
  def __init__(self, n, k):
    self.board = [];
    self.rows = int(n);
    self.columns = int(k);
    self.whose_move = "userone";
    self.last_whose_move = [];
    self.last_which_row = [];
    self.last_which_column = [];
    self.player = 0;
    self.timeout = 0;
```

```
  for rows_count in range(self.rows):
    rows_temp = [];
    for columns_count in range(self.columns):
      rows_temp.append(0);
    self.board.append(rows_temp);

def name(self):
  return 'SUICIDE SQUAD';

def last_move_won(self):
  #print(len(self.last_which_row) -1);
  if len(self.last_whose_move) > 0:
    last_rowmove_made = self.last_which_row[len(self.last_which_row) - 1];
    last_columnmove_made = self.last_which_column[len(self.last_which_column) - 1];
    last_whosemove_made = self.last_whose_move[len(self.last_whose_move) - 1];
    last_whosemove_made_number = -1;

    if last_whosemove_made == "userone":
      last_whosemove_made_number = 1;
    elif last_whosemove_made == "usertwo":
      last_whosemove_made_number = 2;

    #check the verticals if there is a possibility for a win.
    vertical_streak = 0;
    for rows_count in range(last_rowmove_made + (self.rows -1), last_rowmove_made - self.rows , -1):
      if rows_count >= 0 and rows_count <= (self.rows-1):
        if self.board[rows_count][last_columnmove_made] == last_whosemove_made_number:
          vertical_streak += 1;
          if vertical_streak == self.rows:
```

```
            break;
          else:
            vertical_streak = 0;

      if vertical_streak == self.rows:
        return True;

      #check the horizontal streak if there is a possibility for a win.
      horizontal_streak = 0;
      for columns_count in range(last_columnmove_made - (self.rows-1), last_columnmove_made +
self.rows, 1):
        if columns_count >= 0 and columns_count <= (self.columns-1):
          if self.board[last_rowmove_made][columns_count] == last_whosemove_made_number:
            horizontal_streak += 1;
            if horizontal_streak == self.rows:
              break;
          else:
            horizontal_streak = 0;

      if horizontal_streak == self.rows:
        return True;

      #check the diagonal streak if there is a possibility for a win. - diagonal left to right
      diagonal_left_to_right_streak = 0;
      columns_count = last_columnmove_made - (self.rows-1);

      for rows_count in range(last_rowmove_made + (self.rows-1), last_rowmove_made - self.rows, -1):
        if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
```

```
        if self.board[rows_count][columns_count] == last_whosemove_made_number:
          diagonal_left_to_right_streak += 1;
          if diagonal_left_to_right_streak == self.rows:
            break;
        else:
          diagonal_left_to_right_streak = 0;
      columns_count += 1;

    if diagonal_left_to_right_streak == self.rows:
      return True;

    #check the diagonal streak if there is a posssibility for a win - diagonal right to left.
    diagonal_right_to_left_streak = 0;
    columns_count = last_columnmove_made - (self.rows-1);

    for rows_count in range(last_rowmove_made - (self.rows-1), last_rowmove_made + self.rows, 1):
      if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
        if self.board[rows_count][columns_count] == last_whosemove_made_number:
          diagonal_right_to_left_streak += 1;
          if diagonal_right_to_left_streak == self.rows:
            break;
        else:
          diagonal_right_to_left_streak = 0;
      columns_count += 1;

    if diagonal_right_to_left_streak ==  self.rows:
      return True;
    else:
```

```
      return False;
    else:
      return False;

  def generate_moves(self):
    possible_moves = [];
    possible_moves_column = [];

    for columns_count in range(0, self.columns , 1):
      for rows_count in range(0, self.rows, 1):
        if self.board[rows_count][columns_count] == 0:
          possible_moves.append(str(rows_count) + str(columns_count));
          possible_moves_column.append(columns_count);

    return possible_moves_column;

  def make_move_initial(self, row, column):
    if self.whose_move == "userone":
      self.board[row][column] = 1;
      self.whose_move = "usertwo";
      self.player = 1;
      self.last_whose_move.append("userone");
      self.last_which_row.append(row);
      self.last_which_column.append(column);

    elif self.whose_move == "usertwo":
      self.board[row][column] = 2;
      self.whose_move = "userone";
      self.player = 0;
```

```
    self.last_whose_move.append("usertwo");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

def make_move(self, move):
  move_added = False;
  for rows_count in range(self.rows-1, -1, -1):
    if self.board[int(rows_count)][int(move)] == 0:
      if self.whose_move == "userone":
        self.board[rows_count][move] = 1;
        self.whose_move = "usertwo";
        self.player = 1;
        self.last_whose_move.append("userone");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      elif self.whose_move == "usertwo":
        self.board[rows_count][move] = 2;
        self.whose_move = "userone";
        self.player = 0;
        self.last_whose_move.append("usertwo");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      move_added = True;
      break;

def unmake_last_move(self):
```

```
    if len(self.last_whose_move) > 0 and len(self.last_which_row) > 0 and len(self.last_which_column) >
0:
      self.board[self.last_which_row[len(self.last_which_row) -
1]][self.last_which_column[len(self.last_which_column) - 1]] = 0;
      self.whose_move = self.last_whose_move[len(self.last_whose_move) - 1];
      self.last_whose_move.pop();
      if self.player == 0:
        self.player = 1;
      else:
        self.player = 0;
      self.last_which_row.pop();
      self.last_which_column.pop();
      #print("Move undo done");


  def get_move(self):
    #print("my_move ", self.find_win(8));
    self.timeout = time.time() + 2;
    return self.find_win(8);


  def find_win(self, depth):


    #performing iterative deepening search for the win.
    while depth >= 0:
      total_nodes_visited = self.generate_moves();
      results = [];
      final_result = None;
      if time.time() > self.timeout:
        available_moves = self.generate_moves();
        return int(random.choice(available_moves));
```

```
player_temp = 0 if self.player == 1 else 1;

for total_nodes_visited_temp in total_nodes_visited:
  if time.time() > self.timeout:
    available_moves = self.generate_moves();
    return int(random.choice(available_moves));
  self.make_move(total_nodes_visited_temp);
  #print(board);
  #Recurse to the next depth
  result = self.alpha_beta_pruning(depth-1, -2, 2, player_temp);

  if final_result is None and result == 1:
    final_result = ""+str(total_nodes_visited_temp);

  results.append(result);
  self.unmake_last_move();



if final_result is not None:
  return int(final_result);
if max(results) == 0:
  available_moves = self.generate_moves();
  return int(random.choice(available_moves));
else:
  available_moves = self.generate_moves();
  return int(random.choice(available_moves));
depth = depth - 1;
```

```
def alpha_beta_pruning(self, depth, alpha ,beta, player_temp):
 if depth == 0:
  if self.last_move_won():
   if self.player == player_temp:
    return 1;
   else:
    return -1;
  else:
   return 0;
 else:
  total_nodes_visited = self.generate_moves();
  for total_nodes_visited_temp in total_nodes_visited:
   self.make_move(total_nodes_visited_temp);
   result = self.alpha_beta_pruning( depth-1, alpha, beta, player_temp);
   if self.player == player_temp:
    if alpha < result:
     alpha = result;
   else:
    if beta > result:
     beta = result;
   self.unmake_last_move();
   if beta <= alpha:
    break;
  if self.player == 0:
   return alpha;
  else:
   return beta;
```

3. Computer Game File

```
import random
class Player:
  def __init__(self,n ,k):
    # counts stores how many tiles are in each column (initalised to 0)
    self.counts = [0] * int(k)
    #print(self.counts);
    self.board = [];
    self.rows = int(n);
    self.columns = int(k);
    self.whose_move = "userone";
    self.last_whose_move = [];
    self.last_which_row = [];
    self.last_which_column = [];
    self.player = 0;

    for rows_count in range(self.rows):
      rows_temp = [];
      for columns_count in range(self.columns):
        rows_temp.append(0);
      self.board.append(rows_temp);


  def name(self):
    return 'RANDOM'

  def make_move_initial(self, row, column):
    if self.whose_move == "userone":
```

```
    self.board[row][column] = 1;
    self.whose_move = "usertwo";
    self.player = 1;
    self.last_whose_move.append("userone");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

  elif self.whose_move == "usertwo":
    self.board[row][column] = 2;
    self.whose_move = "userone";
    self.player = 0;
    self.last_whose_move.append("usertwo");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

def make_move(self, move):
  # every time a move is made the number of tiles in that column increases by one
  self.counts[move]+=1
  move_added = False;
  for rows_count in range(self.rows-1, -1, -1):
    if self.board[rows_count][move] == 0:
      if self.whose_move == "userone":
        self.board[rows_count][move] = 1;
        self.whose_move = "usertwo";
        self.player = 1;
        self.last_whose_move.append("userone");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);
```

```
      elif self.whose_move == "usertwo":
        self.board[rows_count][move] = 2;
        self.whose_move = "userone";
        self.player = 0;
        self.last_whose_move.append("usertwo");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      move_added = True;
      break;

  def get_move(self):
    # first we generate the moves, which is any column that isn't full (has less than 6 tiles)
    moves = []
    for i in range(0, self.columns):
      if self.counts[i] < self.columns:
        moves.append(i)
    # return a random legal move
    return random.choice(moves)
```

4. Search File

```
import board
import random
def perft(board, depth):
  total_nodes_visited = 0;

  #gets the next possible moves allowed to take.
  available_moves = board.generate_moves();
```

```
 should_end = (depth == 0) or (len(available_moves) == 0);
 if should_end:
  return 1;
 else:
  for available_moves_temp in available_moves:
   board.make_move(available_moves_temp);
   #check if the last made move was a win so that it indicates the end of the tree.
   if board.last_move_won():
    total_nodes_visited += 1;
   else:
    #recurse for the next depth
    total_nodes_visited += perft(board, depth-1);
   board.unmake_last_move();
 return total_nodes_visited;

def alpha_beta_pruning(board, depth, alpha ,beta, player_temp):
 if depth == 0:
  if board.last_move_won():
   if board.player == player_temp:
    return 1;
   else:
    return -1;
  else:
   return 0;
 else:
  #gets the next possible moves allowed to take.
  total_nodes_visited = board.generate_moves();
  for total_nodes_visited_temp in total_nodes_visited:
   board.make_move(total_nodes_visited_temp);
```

```
    #prune and set the alpha beta values based on the depth leaves.
    result = alpha_beta_pruning(board, depth-1, alpha, beta, player_temp);
    if board.player == player_temp:
      if alpha < result:
        alpha = result;
    else:
      if beta > result:
        beta = result;
    board.unmake_last_move();
    if beta <= alpha:
      break;

  if board.player == 0:
    return alpha;
  else:
    return beta;

def find_win(board, depth):
  total_nodes_visited = board.generate_moves();
  results = [];
  final_result = None;

  player_temp = 0 if board.player == 1 else 1;

  for total_nodes_visited_temp in total_nodes_visited:
    board.make_move(total_nodes_visited_temp);
    #print(board);
    #prune and set the alpha beta values based on the depth leaves.
```

```
    result = alpha_beta_pruning(board, depth-1, -2, 2, player_temp);

    if final_result is None and result == 1:
      final_result = "WIN BY PLAYING "+str(total_nodes_visited_temp);

    results.append(result);
    board.unmake_last_move();

  if final_result is not None:
    return final_result;
  if max(results) == 0:
    return "NO FORCED WIN IN %d MOVES" % depth;
  else:
    return "ALL MOVES LOSE";
```

5. Testcase File

```
import random;
import time;

class Player:
  def __init__(self, n, k):
    self.board = [];
    self.rows = int(n);
    self.columns = int(k);
    self.whose_move = "userone";
    self.last_whose_move = [];
    self.last_which_row = [];
    self.last_which_column = [];
    self.player = 0;
    self.timeout = 0;
```

```
  for rows_count in range(self.rows):
    rows_temp = [];
    for columns_count in range(self.columns):
      rows_temp.append(0);
    self.board.append(rows_temp);

def name(self):
  return 'SUICIDE SQUAD';

def last_move_won(self):
  #print(len(self.last_which_row) -1);
  if len(self.last_whose_move) > 0:
    last_rowmove_made = self.last_which_row[len(self.last_which_row) - 1];
    last_columnmove_made = self.last_which_column[len(self.last_which_column) - 1];
    last_whosemove_made = self.last_whose_move[len(self.last_whose_move) - 1];
    last_whosemove_made_number = -1;

    if last_whosemove_made == "userone":
      last_whosemove_made_number = 1;
    elif last_whosemove_made == "usertwo":
      last_whosemove_made_number = 2;

    #check the verticals if there is a possibility for a win.
    vertical_streak = 0;
    for rows_count in range(last_rowmove_made + (self.rows -1), last_rowmove_made - self.rows , -1):
      if rows_count >= 0 and rows_count <= (self.rows-1):
        if self.board[rows_count][last_columnmove_made] == last_whosemove_made_number:
          vertical_streak += 1;
```

```
            if vertical_streak == self.rows:
              break;
            else:
              vertical_streak = 0;

        if vertical_streak == self.rows:
          return True;

        #check the horizontal streak if there is a possibility for a win.
        horizontal_streak = 0;
        for columns_count in range(last_columnmove_made - (self.rows-1), last_columnmove_made +
self.rows, 1):
          if columns_count >= 0 and columns_count <= (self.columns-1):
            if self.board[last_rowmove_made][columns_count] == last_whosemove_made_number:
              horizontal_streak += 1;
              if horizontal_streak == self.rows:
                break;
            else:
              horizontal_streak = 0;

        if horizontal_streak == self.rows:
          return True;

        #check the diagonal streak if there is a possibility for a win. - diagonal left to right
        diagonal_left_to_right_streak = 0;
        columns_count = last_columnmove_made - (self.rows-1);

        for rows_count in range(last_rowmove_made + (self.rows-1), last_rowmove_made - self.rows, -1):
```

```
      if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
        if self.board[rows_count][columns_count] == last_whosemove_made_number:
          diagonal_left_to_right_streak += 1;
          if diagonal_left_to_right_streak == self.rows:
            break;
        else:
          diagonal_left_to_right_streak = 0;
      columns_count += 1;

    if diagonal_left_to_right_streak == self.rows:
      return True;

    #check the diagonal streak if there is a posssibility for a win - diagonal right to left.
    diagonal_right_to_left_streak = 0;
    columns_count = last_columnmove_made - (self.rows-1);

    for rows_count in range(last_rowmove_made - (self.rows-1), last_rowmove_made + self.rows, 1):
      if rows_count >=0 and rows_count <= (self.rows-1) and columns_count >=0 and columns_count <=
(self.columns-1):
        if self.board[rows_count][columns_count] == last_whosemove_made_number:
          diagonal_right_to_left_streak += 1;
          if diagonal_right_to_left_streak == self.rows:
            break;
        else:
          diagonal_right_to_left_streak = 0;
      columns_count += 1;

    if diagonal_right_to_left_streak ==  self.rows:
```

```
      return True;
    else:
      return False;
  else:
    return False;

def generate_moves(self):
  possible_moves = [];
  possible_moves_column = [];

  for columns_count in range(0, self.columns , 1):
    for rows_count in range(0, self.rows, 1):
      if self.board[rows_count][columns_count] == 0:
        possible_moves.append(str(rows_count) + str(columns_count));
        possible_moves_column.append(columns_count);

  return possible_moves_column;

def make_move_initial(self, row, column):
  if self.whose_move == "userone":
    self.board[row][column] = 1;
    self.whose_move = "usertwo";
    self.player = 1;
    self.last_whose_move.append("userone");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

  elif self.whose_move == "usertwo":
    self.board[row][column] = 2;
```

```
    self.whose_move = "userone";
    self.player = 0;
    self.last_whose_move.append("usertwo");
    self.last_which_row.append(row);
    self.last_which_column.append(column);

def make_move(self, move):
  move_added = False;
  for rows_count in range(self.rows-1, -1, -1):
    if self.board[int(rows_count)][int(move)] == 0:
      if self.whose_move == "userone":
        self.board[rows_count][move] = 1;
        self.whose_move = "usertwo";
        self.player = 1;
        self.last_whose_move.append("userone");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      elif self.whose_move == "usertwo":
        self.board[rows_count][move] = 2;
        self.whose_move = "userone";
        self.player = 0;
        self.last_whose_move.append("usertwo");
        self.last_which_row.append(rows_count);
        self.last_which_column.append(move);

      move_added = True;
      break;
```

```
  def unmake_last_move(self):
    if len(self.last_whose_move) > 0 and len(self.last_which_row) > 0 and len(self.last_which_column) >
0:
      self.board[self.last_which_row[len(self.last_which_row) -
1]][self.last_which_column[len(self.last_which_column) - 1]] = 0;
      self.whose_move = self.last_whose_move[len(self.last_whose_move) - 1];
      self.last_whose_move.pop();
      if self.player == 0:
        self.player = 1;
      else:
        self.player = 0;
      self.last_which_row.pop();
      self.last_which_column.pop();
      #print("Move undo done");


  def get_move(self):
    #print("my_move ", self.find_win(8));
    self.timeout = time.time() + 2;
    return self.find_win(8);


  def find_win(self, depth):


    #performing iterative deepening search for the win.
    while depth >= 0:
      total_nodes_visited = self.generate_moves();
      results = [];
      final_result = None;
      if time.time() > self.timeout:
        available_moves = self.generate_moves();
```

```
  return int(random.choice(available_moves));

player_temp = 0 if self.player == 1 else 1;

for total_nodes_visited_temp in total_nodes_visited:
  if time.time() > self.timeout:
    available_moves = self.generate_moves();
    return int(random.choice(available_moves));
  self.make_move(total_nodes_visited_temp);
  #print(board);
  #Recurse to the next depth
  result = self.alpha_beta_pruning(depth-1, -2, 2, player_temp);

  if final_result is None and result == 1:
    final_result = ""+str(total_nodes_visited_temp);

  results.append(result);
  self.unmake_last_move();



if final_result is not None:
  return int(final_result);
if max(results) == 0:
  available_moves = self.generate_moves();
  return int(random.choice(available_moves));
else:
  available_moves = self.generate_moves();
  return int(random.choice(available_moves));
depth = depth - 1;
```

```
def alpha_beta_pruning(self, depth, alpha ,beta, player_temp):
 if depth == 0:
  if self.last_move_won():
   if self.player == player_temp:
    return 1;
   else:
    return -1;
  else:
   return 0;
 else:
  total_nodes_visited = self.generate_moves();
  for total_nodes_visited_temp in total_nodes_visited:
   self.make_move(total_nodes_visited_temp);
   result = self.alpha_beta_pruning( depth-1, alpha, beta, player_temp);
   if self.player == player_temp:
    if alpha < result:
     alpha = result;
   else:
    if beta > result:
     beta = result;
   self.unmake_last_move();
   if beta <= alpha:
    break;
  if self.player == 0:
   return alpha;
  else:
   return beta;
```