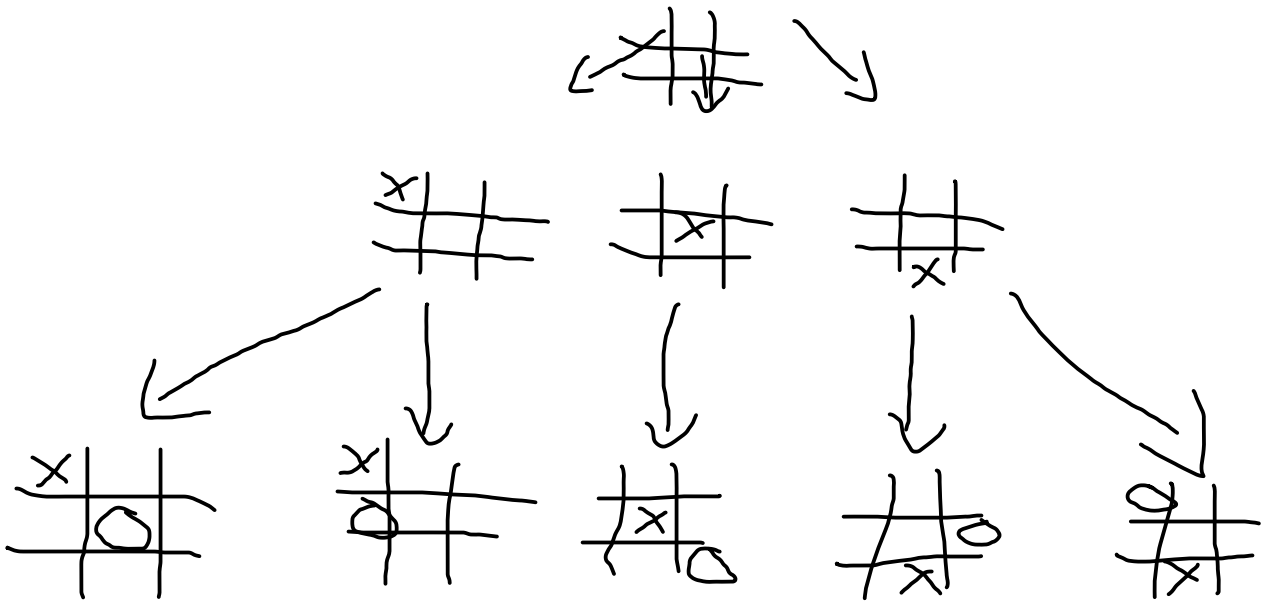
Task 1: Tic-tac-toe

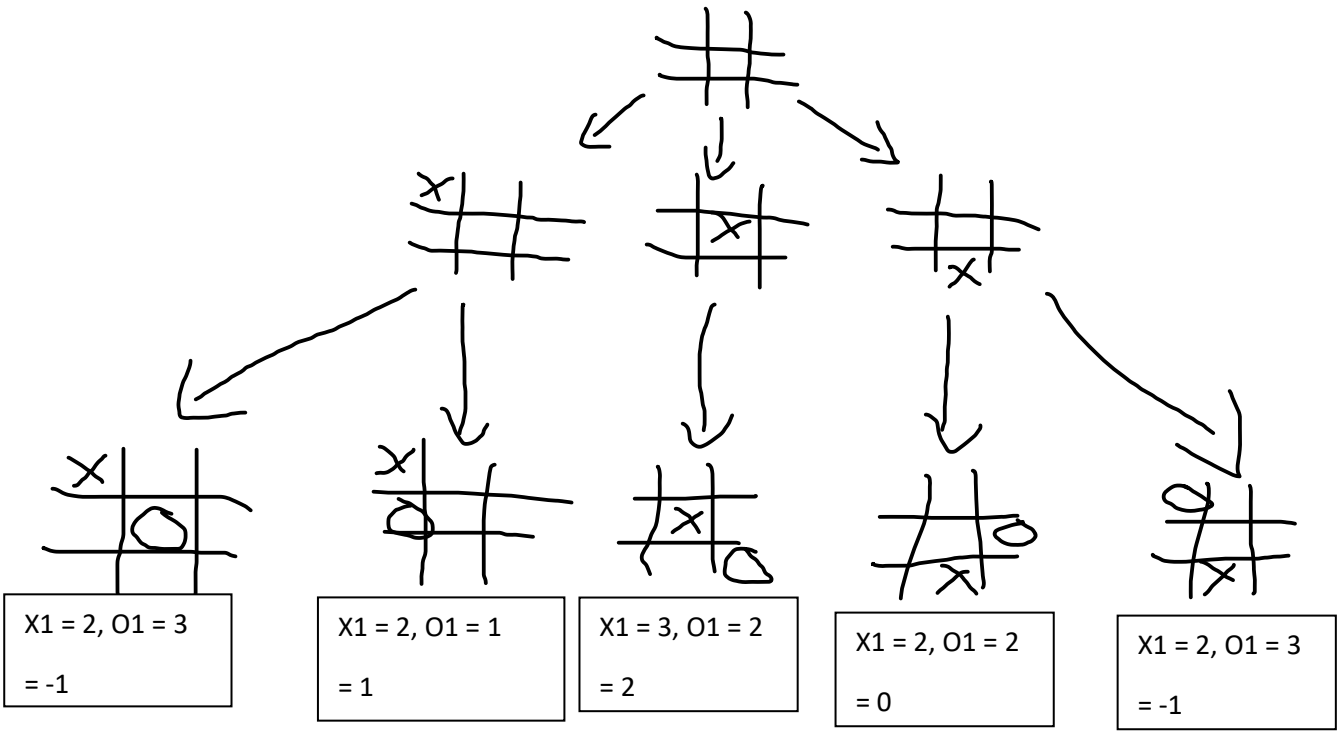
a) 255,168 possible games

b)

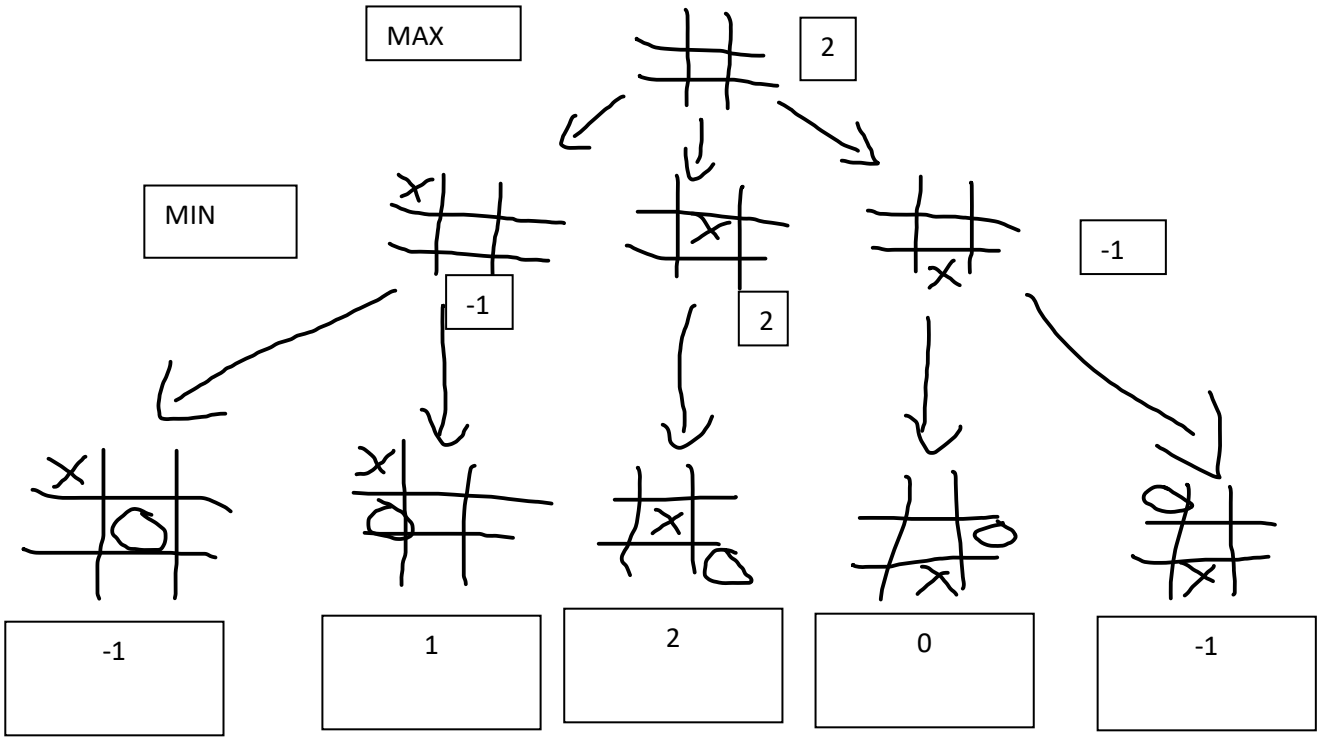


c)

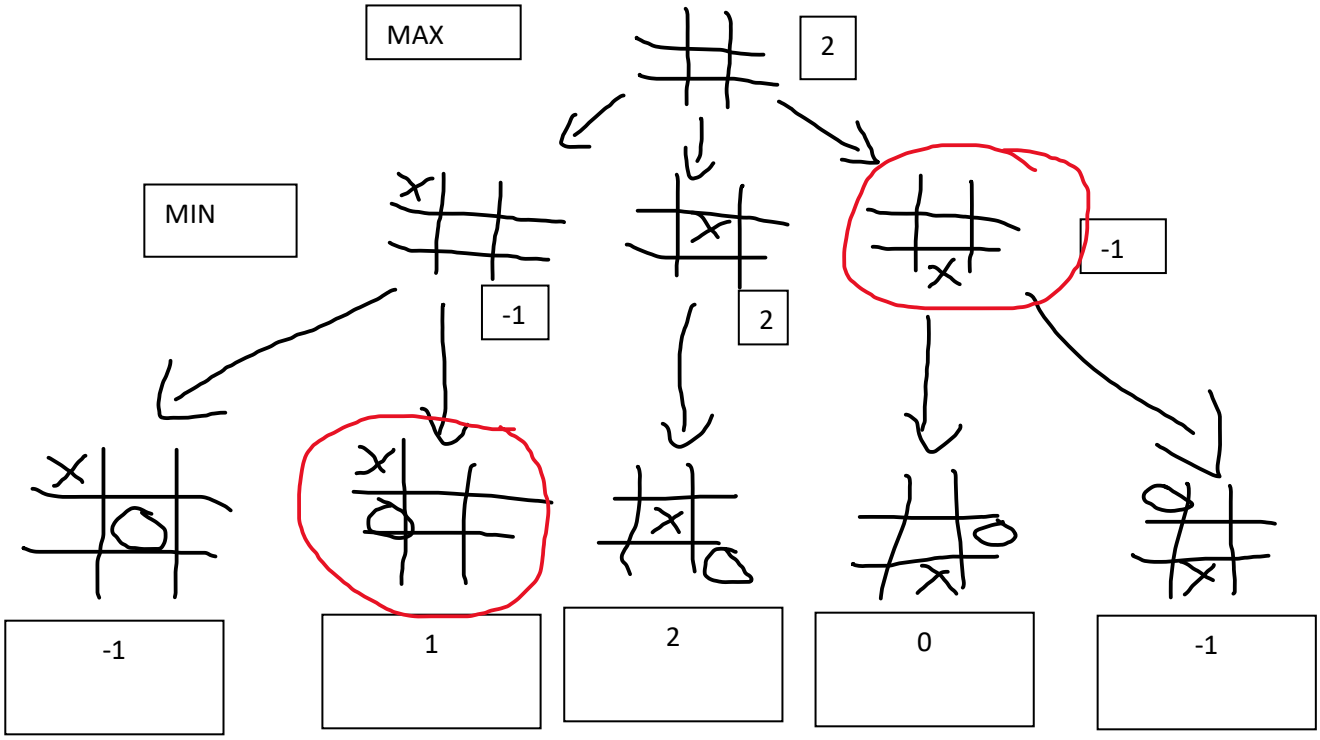
16946889 Justin Yeung



d)



e)



Task 2

For the minimax algorithm, the AI is the minimizing player while the human player is the maximizing player.

When the player marks an X on the board, bestMove() function is called and creates a variable called bestScore initialized to infinity.

```
function bestMove() {
  // AI is the minimizing player
  let bestScore = Infinity;
  let move;
```

The move variable will be used to mark the O for the AI once the minimax algorithm finishes.

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (board[i][j] == '') {
      board[i][j] = ai;
      // For depth limited search, set depth > 0
      let score = minimax(board, 0, true);
      board[i][j] = '';
      if (score < bestScore) {
        bestScore = score;
        move = { i, j };
      }
    }
  }
}
```

In this nested for loop, a temporary position is assigned to the board, then the minimax function is called to check the score of that position. It repeats this for all available positions until it gets the minimal best score and assigns the optimal position to the move variable.

```
  board[move.i][move.j] = ai;
  currentPlayer = human;
```

After the for loops, the AI assigns the optimal move on the board and the game goes to the human player turn

```
let scores = {
  X: 1,
  O: -1,
  tie: 0
}
```

This variable stores what scores will be allocated to; X is 1, O is -1, and tie is 0

```
// The minimax algorithm
function minimax(board, depth, isMaximizing) {
  let result = checkWinner();
  // Check if there's winner first
  if (result !== null) {
    return scores[result];;
  }


  // For depth limited search, base case is when depth == 0
  if (depth <= 0) {
    return eval(); // Return the evaluation
  }

  if (isMaximizing) {
    return maximiseScore(board, depth);
  } else {
    return minimiseScore(board, depth);
  }
}
```

In this function, we first need to check if there is a winner. If there is, then the algorithm ends.

Next if the depth reaches 0, the evaluation is returned and used as the best score

The last if statement checks if the opponent is a maximizing player.

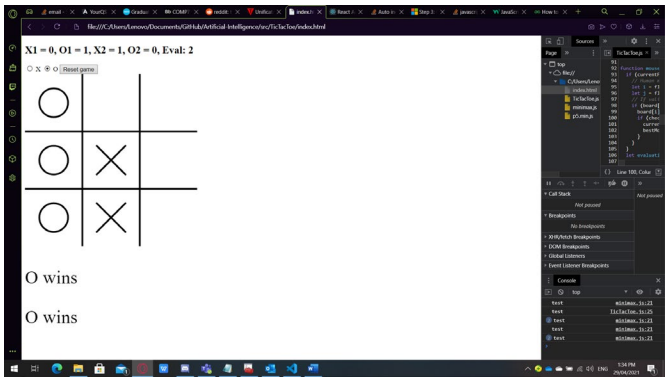
```
// Minimising score function
function minimiseScore(board, depth) {
  let bestScore = Infinity;
  for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
      // Is spot available?
      if (board[i][j] == '') {
        board[i][j] = ai;
        let score = minimax(board, depth - 1, true);
        board[i][j] = '';
        bestScore = min(score, bestScore);
      }
    }
  }
  return bestScore;
}
```

bestScore is set to Infinity for the minimising AI. The for loop go through the whole board and checks if the spot is available. If there is, the AI marks the temporary spot and recursively calls minimax with depth – 1.

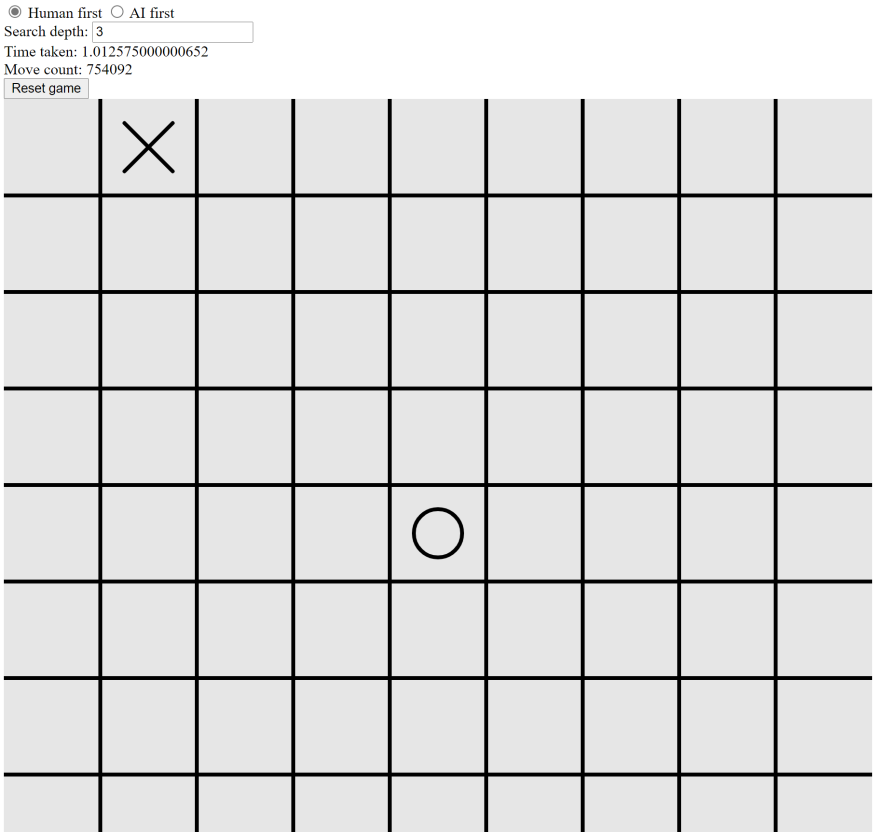
For each recursion, the temporary move is cleared and the bestScore is taken from the minimum score. At the end of the function, bestScore is returned.

The maximisingScore function behaves very similar, with bestScore initially set to -Infinity

Here is an image of the game:

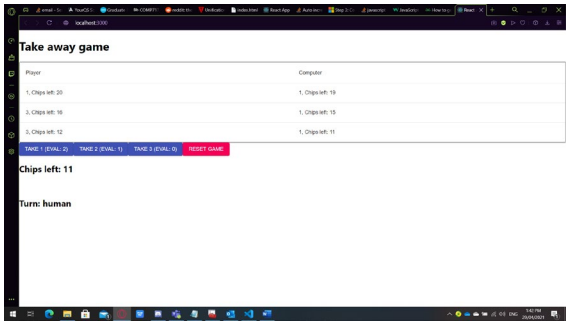


X1 = 2, O1 = 4, X2 = 0, O2 = 0, Eval: -2

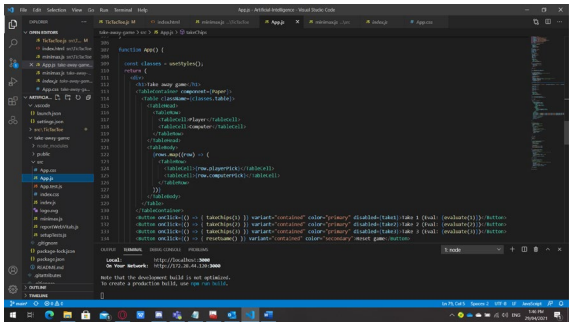


I have also created a 9x9 tic tac toe with alpha beta pruning. This test was done on my Desktop PC with a Ryzen 7 3900X. Increasing the depth beyond 3 would significantly impact performance.

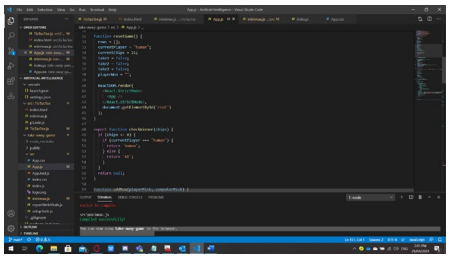
Take away game



This is an image of the take-away game. The evaluations are calculated by taking the modulus current number of chips and how many to take. In this case, we'll assume that the human player is the minimising player and the AI is the maximising player. Therefore, the human will take the ones with the lowest evaluation and the AI will take the highest evaluation score.



This function creates the UI using ReactJS. It uses a table to display the turns, and buttons for the user to click.

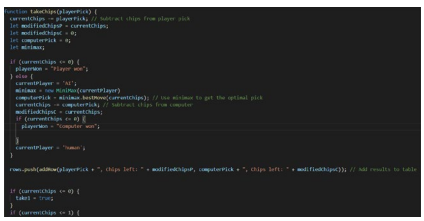


The function resetGame() will reset the take-away game back to its initial state.

The checkWinner() function checks if there is a winner while playing the game.

```
function addRow(playerPick, computerPick) {
  return { playerPick, computerPick };
}
```

This returns a pair that can be used to add a row to the table with the player picks.



This function takes chips from the count. First it subtracts the player picked chips from the stack then checks to see if player won. If not, the game continues and the AI runs the minimax algorithm to determine the best action to take. Then it subtracts computerPick from the current chips and checks if AI won. If not, then current player goes to human player and the cycle repeats until there is a winner. The rows.push adds the picks onto the table.

```
  if (currentChips <= 0) {
    take1 = true;
  }
  if (currentChips <= 1) {
    take2 = true;
  }
  if (currentChips <= 2) {
    take3 = true;
  }

  ReactDOM.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    document.getElementById('root')
  );
}
```

This if statements checks if there is 1, 2 or 0 chips left. This is to disable the specific buttons on the UI if current chips reach that point as to prevent the user from taking chips more than the current amount.

Running 21 chips, each turn took the AI 0.05-0.15 milliseconds, full tree search

```
static/js/main.chunk.js:504:19)
Minimax took:        minimax.js:46
0.054999953135848045 milliseconds
Minimax took:        minimax.js:46
0.14999997802078724 milliseconds
Minimax took:        minimax.js:46
0.014999997802078724 milliseconds
```

With 30 chips, the first turn took 903ms, then subsequent turns took a lot less time.

With 35 Chips, first turn took 5831ms

With 40 chips, first turn took 117299ms

So full tree search is very inefficient when chips >30

The evaluation will be used instead to perform a limited depth tree search, where we can use much larger number of chips

Task 3 – Chess game

Players: Black, White

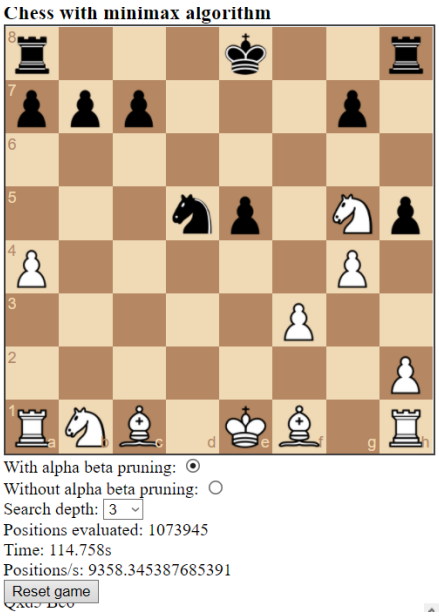
Actions: Player moves pieces if it is legal to do so.

Terminal states: Win, Lose, Stalemate

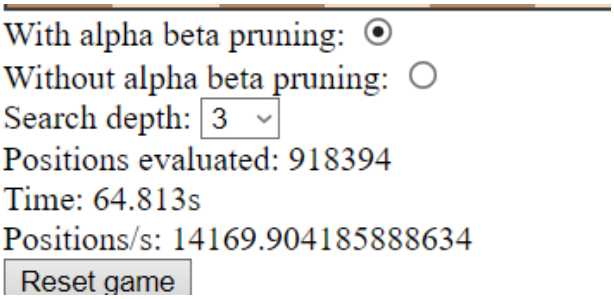
Goal: To win the game

Here is the GDL of this game: https://github.com/parthy/ggp/blob/master/games/chess.gdl

Without alpha beta pruning

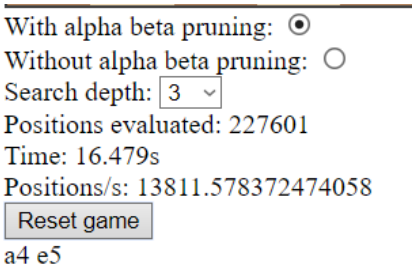


With alpha beta pruning

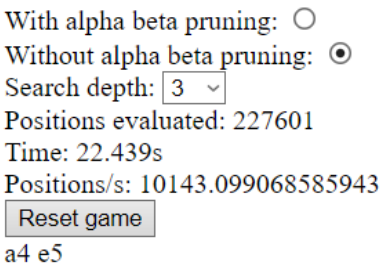


This time we'll do the same move:

With alpha beta pruning



Without alpha beta pruning



Interestingly I found that the positions evaluated were similar but the time difference is significant.

The search I was able to do is up to three because it becomes too slow after that.

Hardware used:

Intel i5-8265U CPU @1.6GHz

16946889 Justin Yeung

RAM: 8GB