Branch and Bound Algorithm in Decision Making for a Connect Four Game

Rinaldy Adin - 13521134
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: rinaldyadin@gmail.com

***Abstract*—This paper aims to implement a popular board game named "Connect Four". The two-player game is played by each player dropping a disc into a vertical board with the goal of creating a line of four discs. The solution proposed implements an algorithm that uses the branch and bound strategy, called Alpha-Beta Pruning, in order to simulate possible moves that can be done to win. The result is that the algorithm is effective in identifying possible opportunities and threats that are given by the opponent, and is able to do the simulations in an efficient manner.**

Keywords—Branch and bound, alpha beta pruning, connect four

I. Introduction

The game of Connect Four has long captivated enthusiasts with its simplicity and strategic depth. As a two-player board game, it offers a challenging environment where players strive to be the first to connect four of their colored discs in a row horizontally, vertically, or diagonally. With a seemingly endless number of possible moves and outcomes, finding an optimal decision-making strategy becomes crucial for success in Connect Four. The board board game is played using a vertical board where players take turns to drop their discs, colored based on the player. Each turn, a player drops their discs into one of the seven columns where the discs will drop as far down as it can go, either dropping into the base of the vertical board or dropping on top of another disc.

Each player has to strategize on which column they are going to drop their disc on. With each player having the final goal of getting four of their discs on a row, players can either drop their discs with the goal of reaching that goal, or by blocking the other player from potentially being able to get four discs on the row.

Figure 1 Connect Four Physical Board Game (https://shop.hasbro.com/en-us/product/connect-4-game/80FB5BCA-5056-9047-F5F4-5EB5DF88DAF4)

One algorithm strategy that could be used to choose an optimal move in the game is by using a branch and bound algorithm. Branch and bound algorithms are used to efficiently explore possible states during solving a certain problem. Branch and bound algorithms branch trough the possible moves a player can make, then evaluating that branched state and scoring the state based on a heuristic. Branching happens until a certain depth is reached or a certain condition is reached. Using the correct heuristic and conditions, the branch and bound algorithm can find an optimal solution to a problem in an efficient manner.

II. Connect Four Respresented in a Branch and Bound Algorithm

As previously mentioned, a connect four game state is represented by a two dimensional grid with seven columns and five rows. Each cell in the grid will represent whether or not the cell is empty and if it is not empty, the cell will represent the color of the disc in the shell, showing which player's disc it is. Then, each turn is represented by receiving an input from the player, then inserting their disc into the chosen column and implementing the logic to "drop" their disc into the bottom of the column. After each turn, the program will have to evaluate for a winnning state based on the resulting positions of discs in the board. The evaluation will consider cases of horizontal, vertical, and diagonal winning states. If a winning state is found, then the player that last inserted their discs wins, and if not, the turn is switched to the other player to insert their own disc. This is repeated until a winning state is found and the winner chosen or when the board is full and no more discs can be inserted, in which case a draw happens between the players.

The game has its appeal from it's simplicity in how to play and how to win, similar in nature to other two player games like tic-tac-toe and chess, albeit being slightly more complex version of tic-tac-toe and also being a much more simpler game than chess. Unlike tic-tac-toe, a game of Connect Four can be quite complex in which two players are strategizing to both get four discs in a row while also trying to block the other player, but unlike a game of chess a player only has seven possible moves during a single turn in a game of Connect Four, while chess players have tens of possible moves. Overall, Connect Four has just the right simplicity for children and adults alike to easily pick the game up and have fun right away.

In order to develop a computer program that’s decent in playing a game of Connect Four, there needs to be an algorithm that can equally match the decision making of a real human that’s playing the game. In order to do that, algorithm strategies that are often used predict and simulate the possible moves and outcomes that it is presented with, and then come up with the best possible move, or at least what it thinks is the best possible move. Simulating those possible moves and outcomes will require a sizeable amount of computing space and resources, although computer power right now has been faster than ever, optimization is still needed so the algorithm won’t have to expand into useless possibilities where it is obvious that the move is not beneficial. One possible algorithm strategy for that is using the branch and bound approach.

Branch and bound algorithm work by branching all possible moves that can be done off of a single state, evaluating and also socing them based on a heuristic, then continuing until a certain condition is reached, such as a winning state or a certain depth of search. The bound in a branch and bound algorithm is done by bounding or discarding the previous states that has a score that’s disadvantageous in comparison to the last found best solution or score. Using those two concepts, branch and bound algorithms are able to simulate many possible moves without having to worry about evaluating obviously disadvantageous possibilities.

In this particular problem of exploring possible moves in a Connect Four game, a state in the exploration is the positions of the colored discs in the board. Branching in this case will be based on the possible moves a player can make, with simulating the alternation of the players turn. While Bounding is done by evaluating a single state based on a heuristic that is deemed able to score whether or not the possible move is advantageuos or not. While common branch and bound algorithms bound or prune branches basesd on whether the score of a branch is higher or lower than the last found best branch, since this exploring the possibilities in a Connect Four game has to consider the turn of the opposing player and also consider the likelihood of the other player winning, an optimized algorithm based on branch and bound is used, called Alpha-Beta Pruning.

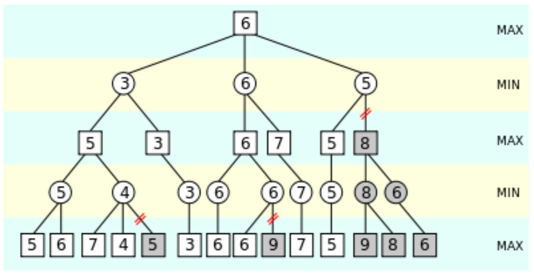


Figure 2 Illusstration of Alpha-Beta Pruning (https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)

Alpha-beta pruning is an optimized branch and bound algorithm commonly used in game tree search, such as in the Connect Four game. It enhances the efficiency of the search by eliminating branches that do not need to be explored further. Unlike traditional branch and bound algorithms that prune branches based on whether the score is higher or lower than the best score found so far, alpha-beta pruning takes advantage of the fact that players take turns in games like Connect Four. It uses two additional values, alpha and beta, to keep track of the lower and upper bounds of the best scores for the maximizing and minimizing players, respectively. By intelligently pruning branches that are guaranteed to be worse than previously explored branches, alpha-beta pruning significantly reduces the number of game states that need to be evaluated, resulting in a more efficient search for the optimal move.

In alpha-beta pruning, the alpha and beta values are used to keep track of the best scores found so far during the search. They represent lower and upper bounds, respectively, on the scores that the maximizing and minimizing players can achieve. The alpha value represents the best (highest) score found for the maximizing player (current player) up to the current point in the search. It starts with an initial value of negative infinity and gets updated as the search progresses. It keeps track of the maximum score that the maximizing player can achieve. The beta value represents the best (lowest) score found for the minimizing player (opponent) up to the current point in the search. It starts with an initial value of positive infinity and also gets updated during the search. It keeps track of the minimum score that the minimizing player can achieve.

As the search algorithm explores different branches of the game tree, it passes the alpha and beta values along to evaluate the possible outcomes. The values are updated as follows:

1. At maximizing (max) nodes:

   When evaluating the children of a max node, if a child's score is found to be greater than the current alpha value, it means the maximizing player has found a better move. In this case, the alpha value is updated with the new, higher score.

2. At minimizing (min) nodes:

   When evaluating the children of a min node, if a child's score is found to be less than the current beta value, it means the minimizing player has found a better move. In this case, the beta value is updated with the new, lower score.

   The pruning occurs when the following conditions are met:

1. At maximizing (max) nodes:

   If a child's score is greater than or equal to the current beta value, the remaining branches under that max node can be pruned. This is because the minimizing player (opponent) would never choose this branch as it leads to a worse outcome for them.

2. At minimizing (min) nodes:

   If a child's score is less than or equal to the current alpha value, the remaining branches under that min node can be pruned. This is because the maximizing player (current player) would never choose this branch as it leads to a worse outcome for them.

By updating and comparing the alpha and beta values, alpha-beta pruning selectively explores only the most promising branches of the game tree while disregarding less desirable branches. This optimization significantly reduces the number of nodes that need to be evaluated, leading to a more efficient search process and improved performance in game-playing scenarios.

III. Implementation

The program used to test and play the Connect Four game is made using python, a simple to use, but powerful programming language. The program will display a Connect Four board that, instead of colored discs, will display X, O, and – (hyphens) as an indicator of whether or not a cell is filled with a discs and which player has which disc. In the implementation the board is simply a two dimensional numpy array fille with numbers corresponding to the player who dropped the disc. In this case, the player that has their moves chosen by the branch and bound algorithm will be the second player, referred to as the "opponent", while the player that gets the first chance to move and will be controlled by the human is the first player.

The main program will only concern with the logic of receiving an input, the logic of validating whether or not the input is valid, and also the logic of checking whether or not a game over scenario is reached, whether it be due to a draw or one of the two players winning. The implementation of the main program is as follows,

```
def play_game():
    board = create_board()
    current_player = 1

    while True:
        if current_player == 1:
            display_board(board)
            col =
get_next_move(current_player)
            print(f"Opponent played {col}")
        else:
            col = get_opponent_move(board,
current_player, DEPTH)

        if is_valid_move(board, col):
            make_move(board, col,
current_player)

            if check_win(board,
current_player):
                display_board(board)
                print(f"Player
{current_player} wins!")
                break

            if np.count_nonzero(board == 0) ==
0:
                display_board(board)
                print("It's a draw!")
                break

            current_player = 3 -
current_player  # Switch players (1 -> 2, 2 ->
1)
```

```
            print()
        else:
            print("Invalid move. Please choose
another column.")
```

The main program also handles the turn of the current player, alternating between the player and the computer algorithm. Checking whether or not there is a winning condition in every turn is done with a brute force approach where every possibility of a consecutive four discs of the same color being positioned horizontally, vertically, or diagonally. The brute force algorithm is as follows,

```
def check_win(board, player):
    # Check horizontally
    for row in range(ROWS):
        for col in range(COLS - 3):
            if board[row][col] == player and
board[row][col + 1] == player and
board[row][col + 2] == player and
board[row][col + 3] == player:
                return True

    # Check vertically
    for row in range(ROWS - 3):
        for col in range(COLS):
            if board[row][col] == player and
board[row + 1][col] == player and board[row +
2][col] == player and board[row + 3][col] ==
player:
                return True

    # Check diagonally (positive slope)
    for row in range(ROWS - 3):
        for col in range(COLS - 3):
            if board[row][col] == player and
board[row + 1][col + 1] == player and
board[row + 2][col + 2] == player and
board[row + 3][col + 3] == player:
                return True

    # Check diagonally (negative slope)
    for row in range(3, ROWS):
        for col in range(COLS - 3):
            if board[row][col] == player and
board[row - 1][col + 1] == player and
board[row - 2][col + 2] == player and
board[row - 3][col + 3] == player:
                return True

    return False
```

After the first player inputs the position to drop their disc, the program immediately switches into the second player and computes the best move to make as the opponent. As previously mentioned before, the opponent uses an alpha beta pruning algorithm, which is a type of branch and bound algorithm that uses extra variables, being alpha and beta, to determine the score, effectiveness, and relevance of the current state being evaluated. The main alpha beta pruning algorithm and the function to extract the best move out of the alpha beta algorithm is as follows,

```
def alpha_beta_pruning(board, player, depth,
alpha, beta):
    best_move = None

    # Check game state and evaluate if depth
is reached or game is over
    if depth == 0 or check_win(board, player)
or no_more_moves(board):
        return evaluate(board, player),
best_move

    for move in get_possible_moves(board):
        make_move(board, move, player)
        score, _ = alpha_beta_pruning(board,
get_opponent(player), depth - 1, -beta, -
alpha)
        score *= -1
        undo_move(board, move)

        if score >= beta:
            return beta, move
        if score > alpha:
            alpha = score
            best_move = move

    return alpha, best_move
```

As shown in the implementation, the get_opponent_move function returns the best possible move evaluated by the alpha_beta_pruning function. The alpha_beta_pruning function returns a tuple containing the score of the evaluated state and also the best move that is most beneficial coming from that evaluated state. The algorithm goes into the max depth that it is allowed to go in this case the max detph is set to 4, or until it reaches a state where one of the player win or the both draw, where in that case the algorithm calculates and return the score for that particular case. The alpha beta pruning doesn’t directly check which player it is currently evaluating, whether it is the maximizing player (opponent) or the minimizing player (human). Instead, the algorithm switches the alpha and beta values of the algorithm everytime it goes to a deeper level and also the algorithm negates the score that it receives from a lower level, since it was maximized for the other player, while in the current level the value for the other player should be minimized.

The alpha beta pruning function compares the values of each state by evaluating a score for each state it has reached, in order to evaluate that state, there needs to be a heuristic calculated for that current state. The heuristic chosen should be able to reduce a certain state into a value that represents the advantage and disadvantage a current player has by being in that state. Knowing that there are some considerations in choosing the heuristic,

1. Threats and Opportunities:

   Identify potential threats and opportunities on the board. Look for patterns that can lead to a win or block the opponent's winning moves. Assign higher values to game states that have more potential winning moves and fewer opportunities for the opponent.

2. Connected Discs:

   Evaluate the number of connected discs in rows, columns, and diagonals. Connected discs contribute to winning possibilities. Assign higher weights to longer chains of connected discs, as they represent stronger positions.

3. Center Control:

   Recognize the importance of controlling the center columns, as they offer more opportunities for creating winning sequences. Assign higher values to game states where the player has discs in the center columns.

4. Safe Moves:

   Evaluate the safety of moves by considering their vulnerability to being blocked by the opponent in subsequent turns. Avoid moves that can easily be countered by the opponent and prioritize moves that maximize the player's chances of connecting four discs.

5. Positional Advantage:

   Consider the positional advantage of certain columns or cells on the board. Assign higher weights to moves that provide strategic advantages, such as creating multiple potential winning sequences or blocking the opponent's important positions.

A simple implementation for the heuristic that considers the treats and opportunities based on the connected discs and also whether or not the move puts the computer in a worse position is by calculating the number of possible winning moves a player can get from a single move from that state and also by evaluating whether or not that state is already a winning state for the current player or the opposing player. The implementation of the evaluate function is as follows,

```
def evaluate(board, player):
    opponent = 3 - player  # Get the opponent
player

    # Evaluate for winning moves
    if check_win(board, player):
        return float('inf')  # Current player
has a winning move, return positive infinity
    elif check_win(board, opponent):
        return -float('inf')  # Opponent has a
winning move, return negative infinity

    # Evaluate for the number of potential
winning moves
    player_winning_moves =
count_winning_moves(board, player)
    opponent_winning_moves =
count_winning_moves(board, opponent)

    # Calculate the heuristic score
    score = player_winning_moves - 
opponent_winning_moves

    return score
```

The evaluate function will return a positive or negative infinity in the case that a certain state will result in a win or loss for the current player. If the check_win function detects that the state is a win for the current player, the function will return positive infinity, causing the alpha_beta_pruning function to identify that state as the worst state. Since the branch and bound function alternates between players to consider, so a win for the current player in the evaluate function means that the opposite player is getting a winning state. But, if the check_win function detects that the current state causes a loss for the current player, the evaluate function will return a negative infinity, causing the alpha_beta_pruning function to identify the state as a best state for the current player int the alpha_beta_pruning function.

In the case that a state is not a winning or losing state, the evaluate functio will score the state based on the heuristic mentioned before. The function will calculate the possible winning moves using the count_winning_moves that will get the number of possible winning a player can get within a single turn. The evaluate function subtracts the possible winning moves of the current player with the possible winning moves of the opposing player. This heuristic should be able to predict which moves are beneficial and which are disadvantageous.

The count_winning_moves function uses a brute force approach, similar to the check_win function. The count_winnning_moves function only calculates the number of possible moves that a player can make which results in a win situation, whether it be a horizontal, vertical, or diagonal. The implementation is as follows,

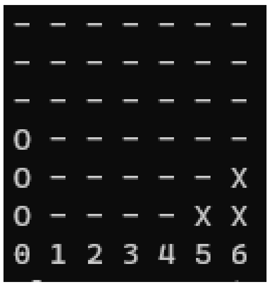
```
def count_winning_moves(board, player):
    count = 0

    # Check horizontally
    for row in range(ROWS):
        for col in range(COLS - 3):
            if board[row][col] == player and 
board[row][col + 1] == player and 
board[row][col + 2] == player and 
board[row][col + 3] == 0:
                count += 1

    # Check vertically
    for row in range(ROWS - 3):
        for col in range(COLS):
            if board[row][col] == player and 
board[row + 1][col] == player and board[row + 
2][col] == player and board[row + 3][col] == 
0:
                count += 1

    # Check diagonally (positive slope)
    for row in range(ROWS - 3):
        for col in range(COLS - 3):
            if board[row][col] == player and 
board[row + 1][col + 1] == player and 
board[row + 2][col + 2] == player and 
board[row + 3][col + 3] == 0:
                count += 1

    # Check diagonally (negative slope)
    for row in range(3, ROWS):
        for col in range(COLS - 3):
            if board[row][col] == player and 
board[row - 1][col + 1] == player and 
board[row - 2][col + 2] == player and 
board[row - 3][col + 3] == 0:
                count += 1


    return count
```

As shown in the implementation the count_winning_moves function brute forces the possible chances of winning. The function is similar to the check_win function, but instead checks whether or not there are three consecutive discs and has an adjacent cell that results in a winnning condition.

IV. Results

In order check the effectiveness of the algorithm, testing is done by playing against the algorithm and putting the algorithm in certain situations to see whether or not the algorithm responds to a situation accordingly.

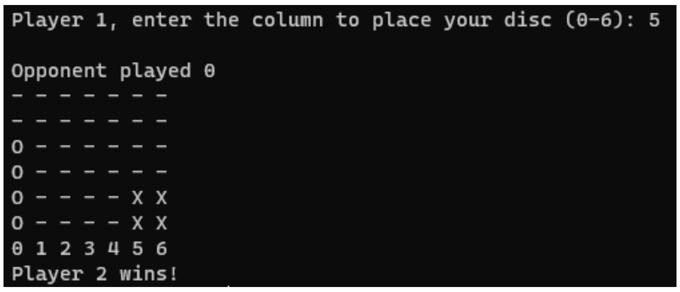
First, the opponent is put in a situation where there is an obvious move where it is beneficial and will result in a win,



```
- - - - - - -
- - - - - - -
- - - - - - -
O - - - - - -
O - - - - - X
O - - - - X X
0 1 2 3 4 5 6
```

Figure 3 Possible Winning Condition for Player Two

In that case, the algorithm (represented with O) can simply place their disc in the 0 column and be able to win the gane, in the case that the player (represented with X) doesn't block that situation,

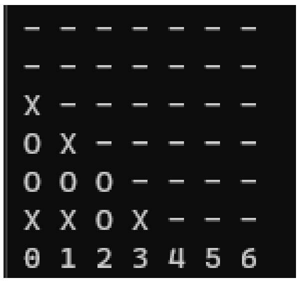


```
Player 1, enter the column to place your disc (0-6): 5

Opponent played 0
- - - - - - -
- - - - - - -
O - - - - - -
O - - - - - -
O - - - - X X
O - - - - X X
0 1 2 3 4 5 6
Player 2 wins!
```

Figure 4 Winning Condition for Player Two

The algorithm is able to identify that dropping a disc in column 0 results in a win.

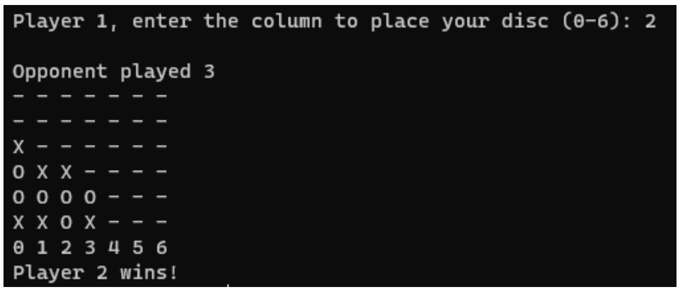
Another case to be tested is by testing whether or not the algorithm identifies winning situations with other orientations, such as horizontal or diagonal. Here the algorithm is given a clear choice of dropping their disc into a winnning position with a horizontal orientation,



```
- - - - - - -
- - - - - - -
X - - - - - -
O X - - - - -
O O O - - - -
X X O X - - -
0 1 2 3 4 5 6
```

Figure 5 Possible Horizontal Winning Condition for Player Two

In that case, the algorithm can simply place their O disc into column 3. The human player will simulate not realizing that losing condition by dropping an X disc into column 2,

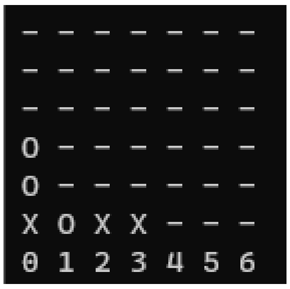


```
Player 1, enter the column to place your disc (0-6): 2

Opponent played 3
- - - - - - -
- - - - - - -
X - - - - - -
O X X - - - -
O O O O - - -
X X O X - - -
0 1 2 3 4 5 6
Player 2 wins!
```

Figure 6 Winning Condition for Player Two

The algorithm is able to identify that dropping a disc in column 3 results in a win.

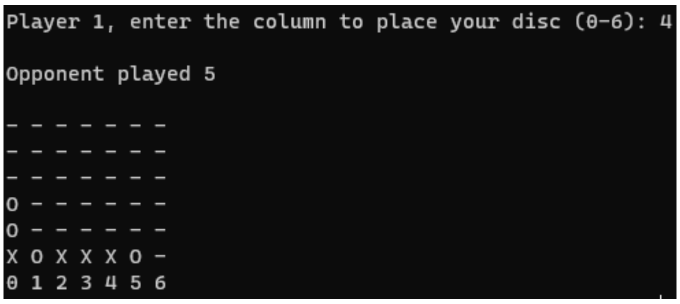
Lastly it should be tested whether or not the algorithm is able to block the human players potential winning condition. Here, the player already has two consecutive discs and dropping a third one means that the algorithm should block possible winning moves for the player,



```
- - - - - - -
- - - - - - -
- - - - - - -
O - - - - - -
O - - - - - -
X O X X - - -
0 1 2 3 4 5 6
```

Figure 7 Possible Losing Condition for Player Two

The human player will drop their X disc into column 4, where if the algorithm fails to block that three horizontal consecutive X discs into four consecutive discs, the algorithm will lose,



```
Player 1, enter the column to place your disc (0-6): 4

Opponent played 5

- - - - - - -
- - - - - - -
- - - - - - -
O - - - - - -
O - - - - - -
X O X X X O -
0 1 2 3 4 5 6
```

Figure 4 Player Two Blocking Player One Winning Condition

The player moving their disc into column four has successfully alterted the algorithm into blocking the players move.

V. Conclusion

Using a branch and bound approach in implementing an algorithm to play connect four is proven to be effective in choosing the right moves in order for the algorithm to win. The alpha beta pruning algorithm has successfully pruned solutions that are deemed useless with branching into a depth level of four is not hindered by performance and the algorithm is able to produce a advantageous solution quickly. A next step in optimizing this implementation can be done by using a more accurate and complex heuristic to calculate an advantageous situation in the branching.

Video LINK

https://bit.ly/video-makalah-13521134

Github Link

https://github.com/Rinaldy-Adin/connect-four-bnb

References

[1] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Pearson Education. p. 167. ISBN 978-0-13-604259-4.

[2] Kuo, J. C. (2021, May 26). *Artificial Intelligence at Play—Connect Four (Minimax algorithm explained)*. Medium. https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f

[3] *Alpha-Beta - Chessprogramming wiki*. (n.d.). Alpha-Beta - Chessprogramming Wiki. https://www.chessprogramming.org/Alpha-Beta#:~:text=The%20Alpha%2DBeta%20algorithm%20(Alpha,of%20overlooking%20a%20better%20move.

[4] *https://shop.hasbro.com/en-us/product/connect-4-game/80FB5BCA-5056-9047-F5F4-5EB5DF88DAF4*. (n.d.). https://shop.hasbro.com/en-us/product/connect-4-game/80FB5BCA-5056-9047-F5F4-5EB5DF88DAF4

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

[illegible]

Rinaldy Adin 13521134