



Department of Computer Science  
Nirma University

---

# Artificial Intelligence

-

## Innovative Assignment

### Connect Four Game using AI

By

17BCE010 Bharat Adhyaru  
17BCE014 Khodidas Chauhan  
17BCE015 Chinmaya Kapopara

## Table of content

1.	About Game	2
2.	Algorithm	6
3.	Flowchart	8
4.	Explanation of Important Functions	9
5.	Code	14
6.	Conclusion	26
7.	References	26

## About Game

Connect Four is a two-player connection board game, designed by Howard Wexler and NEd Strongin, which is also known as Plot Four, Four Up, Four in a Line, Four in a Row, Drop Four and Gravitrips.

### 1. Gameplay

Connect Four has two players playing against each other on a 6x7 (6 row, 7 columns) vertically placed board. Each player is represented by tokens of different color pieces.

When a player takes a turn, he/she puts the token in a given column and it falls down to the last empty cell.

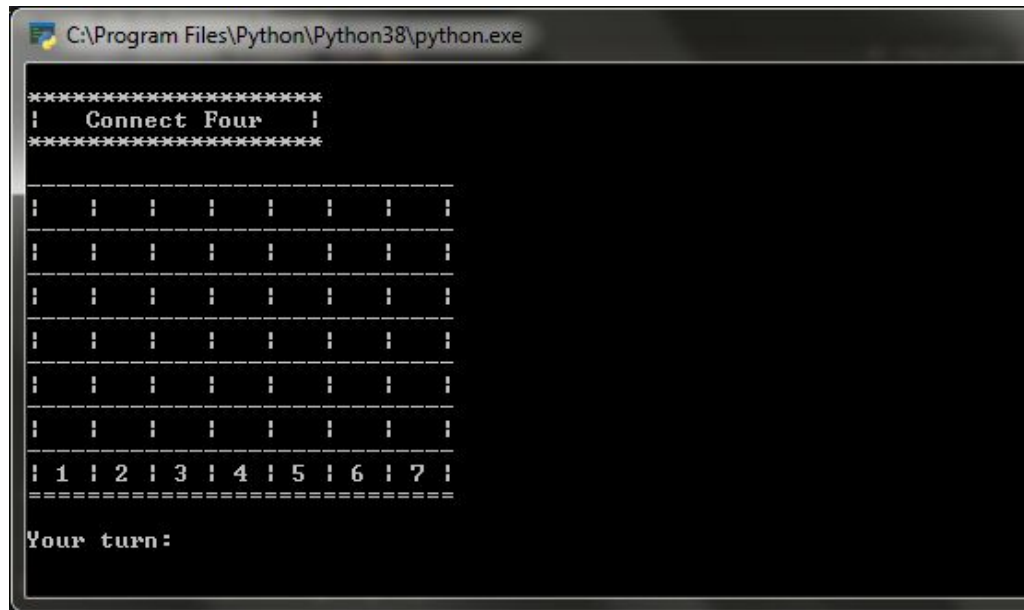
### 2. Rules & Objective

The Rules are very simple that one cannot place a token in a non-empty column. The Objective of the game is to connect four tokens of the same color either in a row, column or any of the diagonals.

Connect Four is a solved game i.e., its complete state space search tree can be made however, the branching factor is 7 ( $b=7$ ) and the maximum depth it can go is 42 ( $m=42$ ). Hence, the total number of nodes in the tree would be  $7^{42} = 3.1197348e+35$



*The Classic Connect Four Board*

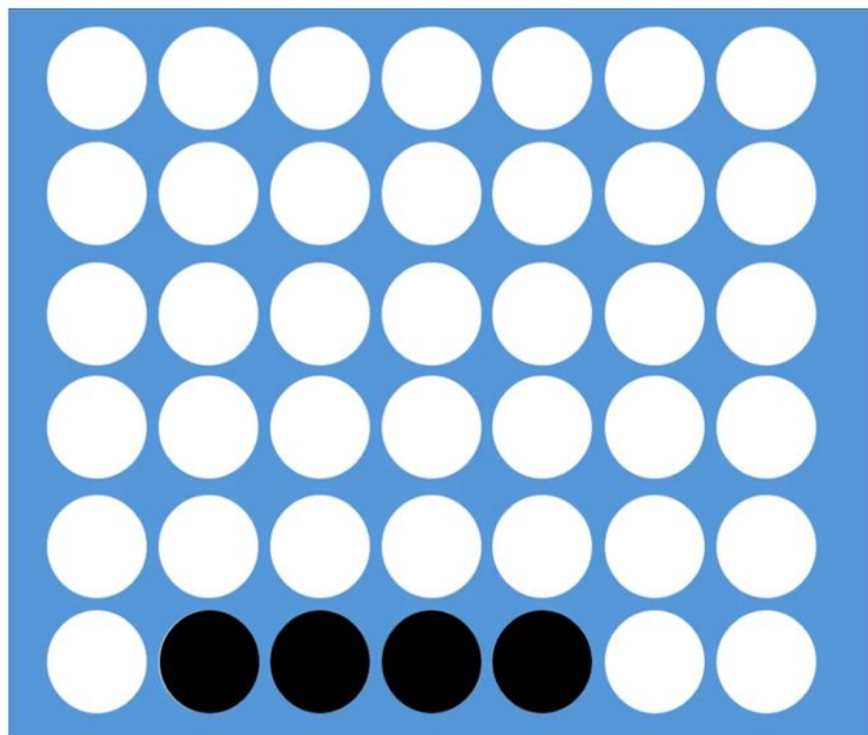


*The Connect Four created by us*

### 3. Winning Conditions

As mentioned above, one can win by connecting four tokens in:

#### a. Row



You can get a row horizontally

```

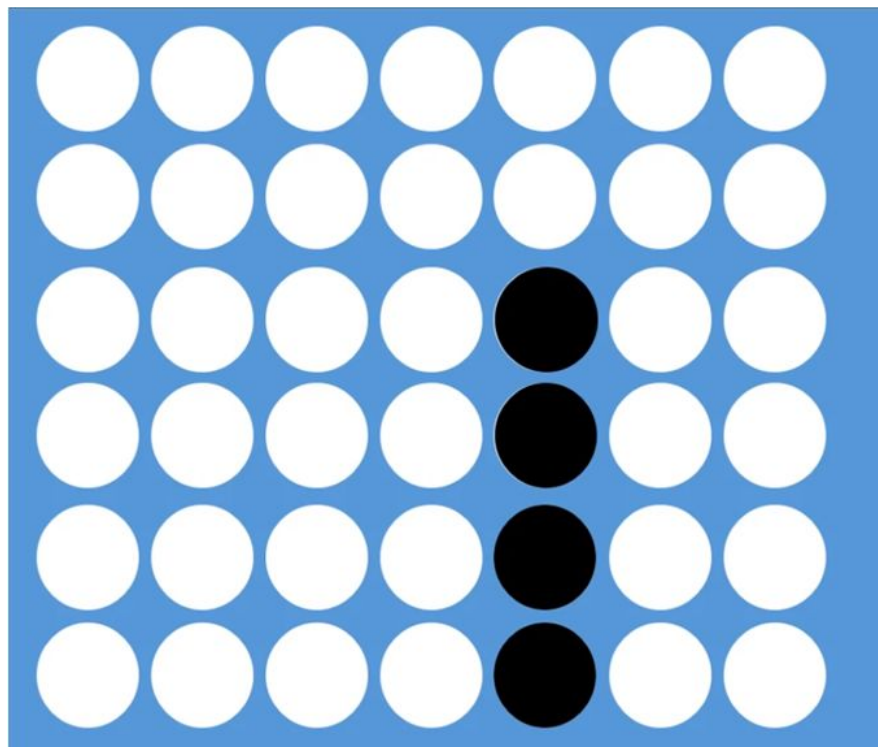
C:\Program Files\Python\Python38\python.exe

| | | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| X | X | X | X | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
=====

Process returned 0 (0x0)      execution time : 0.267 s
Press any key to continue . . .

```

b. Column



You can get a  
row vertically

```

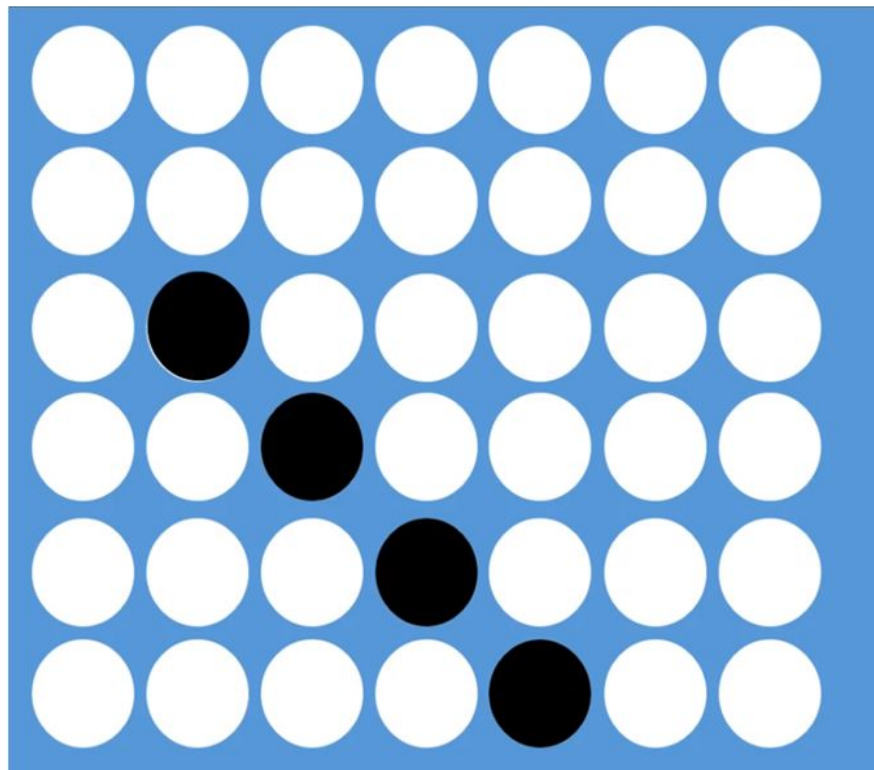
C:\Program Files\Python\Python38\python.exe

| | | | | | | |
| | | | | | | |
| | | 0 | | | | |
| | | 0 | | | | |
| | | 0 | | | | |
| | | 0 | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
=====

Process returned 0 (0x0)      execution time : 0.241 s
Press any key to continue . . . _

```

### c. Diagonals



You can get a row on an angle

```

C:\Program Files\Python\Python38\python.exe

| | | | | | | |
| | | | | | | |
| | X | | | | | |
| | | X | | | | |
| | | | X | | | |
| | | | | X | | |
| | | | | | X | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
=====

Process returned 0 (0x0)          execution time : 0.474 s
Press any key to continue . . .

```

```

C:\Program Files\Python\Python38\python.exe

| | | | | | | |
| | | | | | | |
| | | | | 0 | | |
| | | | | 0 | | |
| | | | 0 | | | |
| | | 0 | | | | |
| | | 0 | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
=====

Process returned 0 (0x0)          execution time : 0.191 s
Press any key to continue . . .

```

## Algorithm

### Overview

The algorithm uses Minimax for game playing, so as to decide which step to take next. We have also used Alpha-Beta Pruning so as to prune the unnecessary branches in the

state space tree, because as mentioned earlier the total number of nodes reaches as much as  $7^{42}$ , which is way too large for a normal computer to compute. Hence, by using Alpha-Beta Pruning and limiting the search to a certain depth (here, depth=5) we optimized our algorithm so it can work on a normal computer.

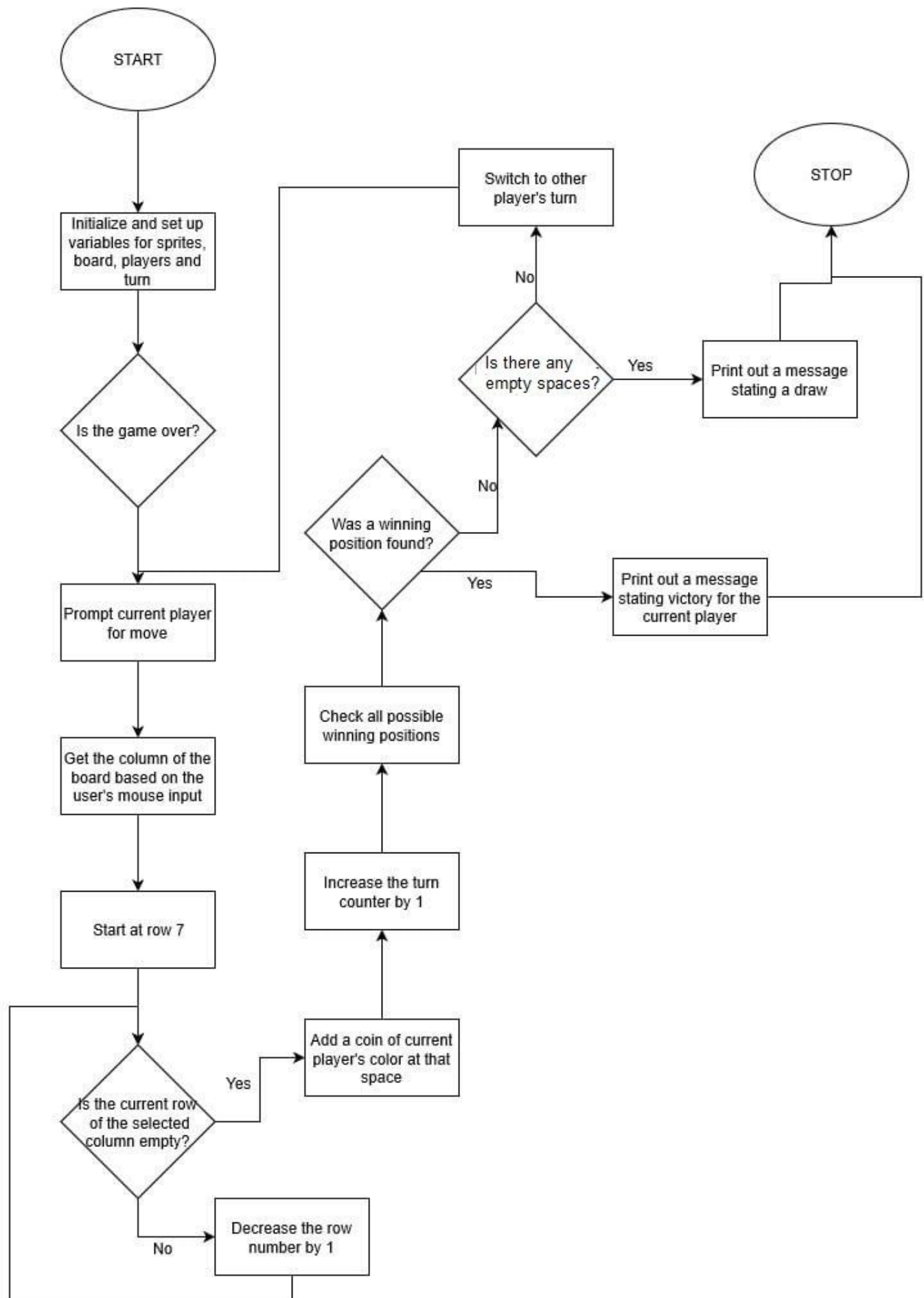
**Note:** Even after limiting the state space tree search to depth 5, the AI may have to check as much as 117,649 moves ( $7^6$ ).

-----

## Flowchart

The Flowchart of our algorithm is mentioned below:

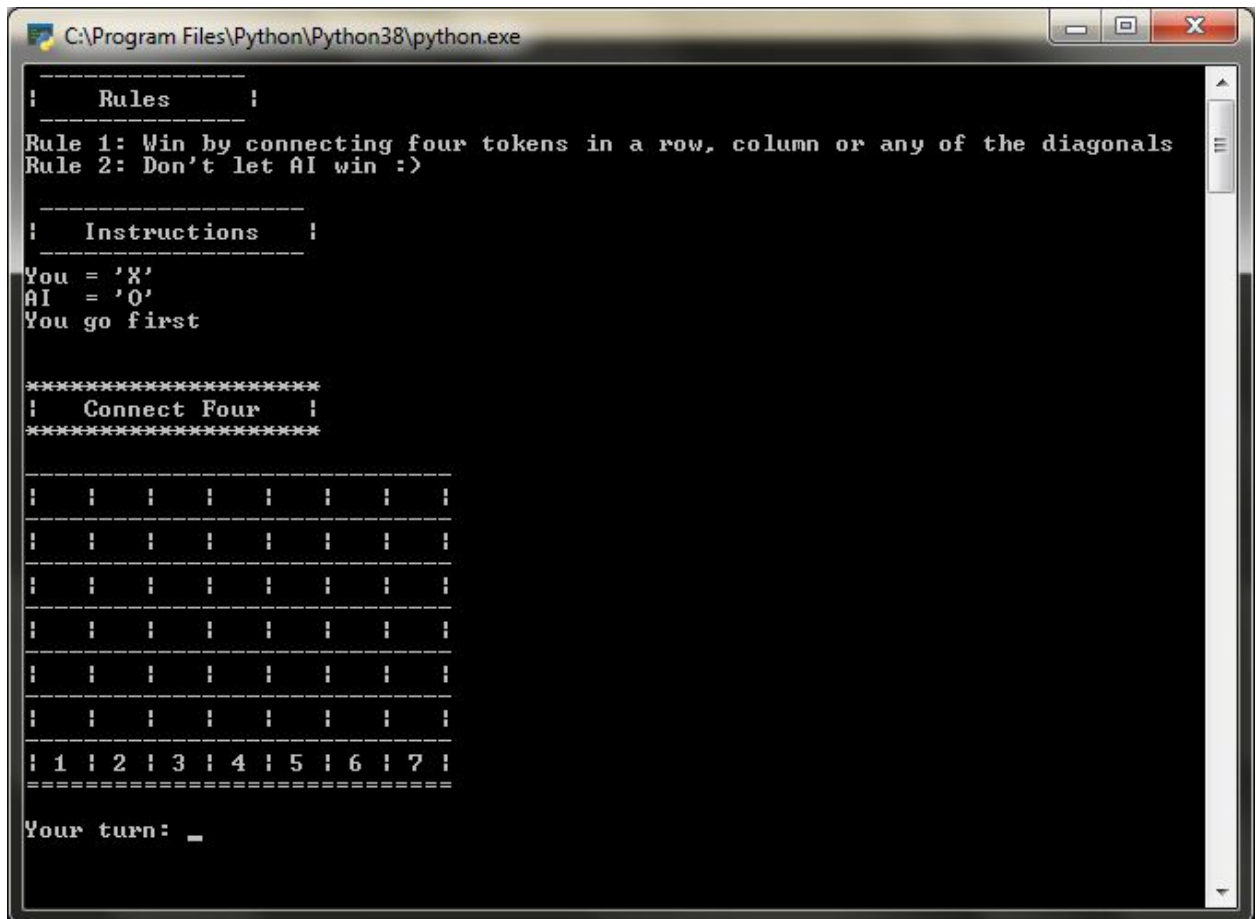




## Explanation of important functions

### 1. printBoard() and printInstructions()

As the name suggests printInstructions() prints the instructions at the beginning while printBoard() prints the board after every move of the player as well as AI.



```

C:\Program Files\Python\Python38\python.exe

!   Rules   !
-----
Rule 1: Win by connecting four tokens in a row, column or any of the diagonals
Rule 2: Don't let AI win :)

!   Instructions   !
-----
You = 'X'
AI   = 'O'
You go first

*****
!   Connect Four   !
*****

! | | | | | | | !
! | | | | | | | !
! | | | | | | | !
! | | | | | | | !
! | | | | | | | !
! | | | | | | | !
! | | | | | | | !
! 1 2 3 4 5 6 7 !
=====

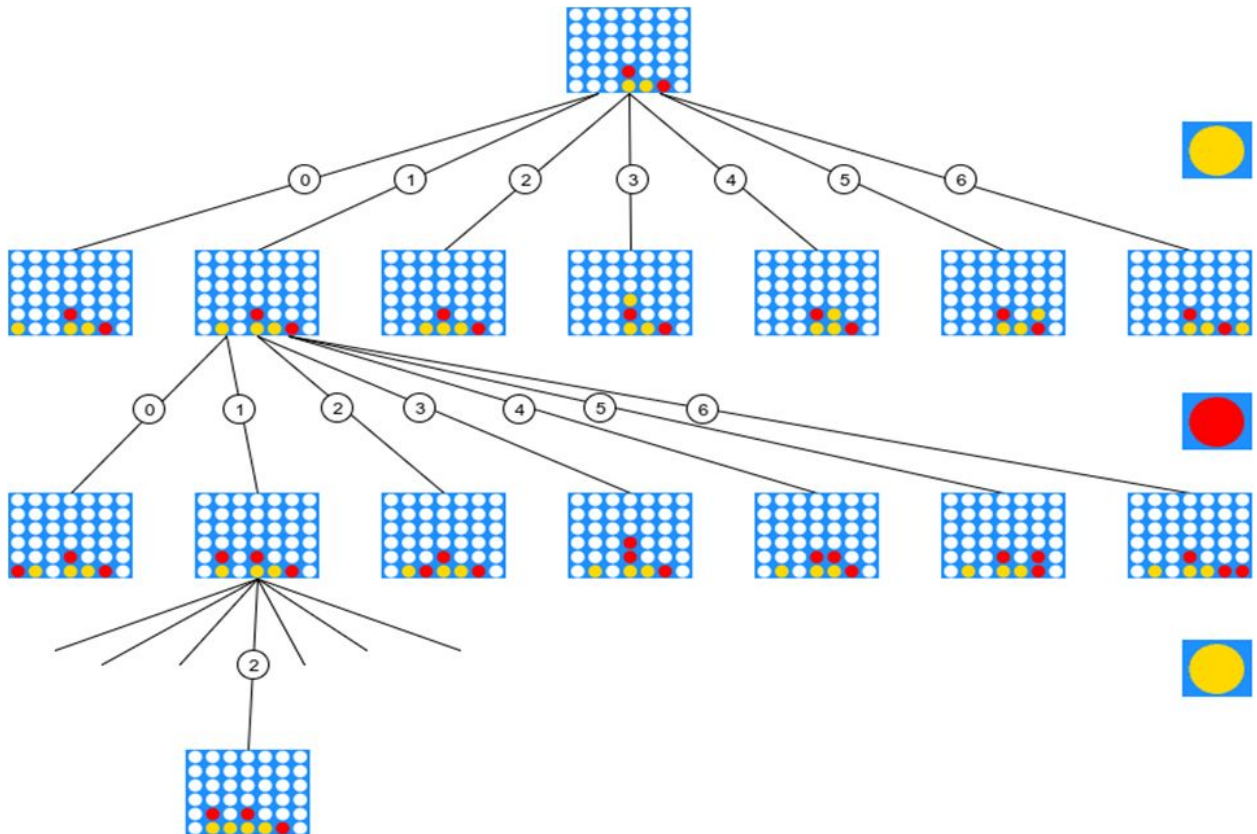
Your turn: _
  
```

*The working of printInstructions() and printBoard() functions*

### 2. findBestMove() and Minimax()

In findBestMove() the AI places its token in every possible column cell and then calls the Minimax function for further processing. Hence, findBestMove() is the starting point of initiating Minimax(). Minimax() then creates the state space tree till a game ends condition (win, lose or draw) or upto to a certain depth.

The working of findBestMove() and Minimax() is shown below:



### 3. Heuristics

The heuristics are precisely designed in such a way that AI focuses a little more on winning rather than focusing on defending and winning equally. We achieved the heuristics value (mentioned in code) by rigorous testing and some online help.

#### a. `winning_move()`

This function checks that in the current state of board is there any possibility for any player (AI or player) to win. If yes, it returns true and the game ends right then, else it returns false and the game continues.

#### b. `anyMovesLeft()`

This function simply checks if there is/are move(s) left i.e., it checks if there is any empty cell. If yes, it returns **True** and the game continues, else it returns **False** and that indicates the game is drawn.

#### c. `evaluateBoard()`

This function simply evaluates the board, after every move made by the AI, in `Minimax()` and assigns a heuristic value based on the state of the board.

**It assigns value in following ways:**

1. +10,000 if AI is winning
2. -10,000 if AI is losing
3. 100 if found |X|X|X|X|
4. 5 if found as |X|X|X|\_| or |X|X|\_|X| and so on
5. 2 if found as |X|X|\_|\_| or |X|\_|X|\_| and so on

The heuristics for point 3 to 5 are multiplied by -1 if it's the player's turn else it is passed as it is in case of AI turns (because AI is the maximizer player here).

**d. checkRow(), checkCol(), checkDiag1(), checkDiag2()**

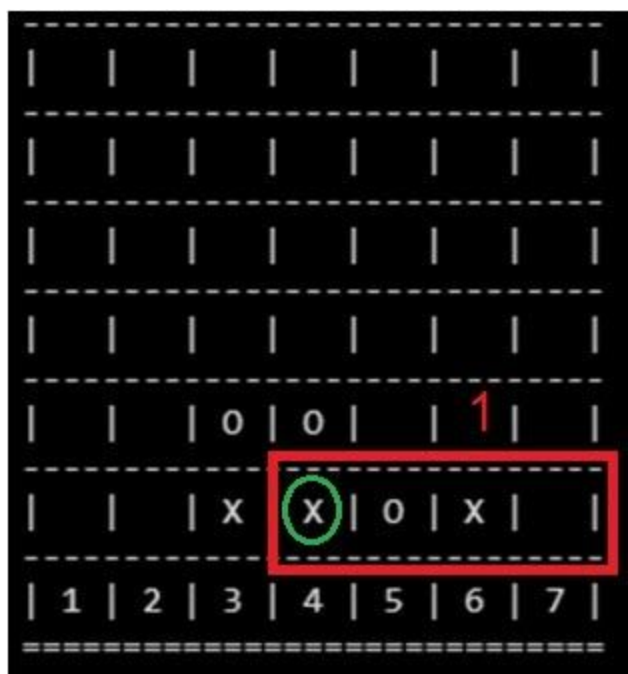
The evaluateBoard() takes help of above mentioned functions to assign heuristic value to the board state.

The rowCheck function evaluates all permutations of a token by placing it in four different positions of four different rows, it can be a part of. And then sums up the score of the heuristic values. This is done in order to take the AI towards a state (of board) where it has the highest chances of winning or lowest chances of losing.

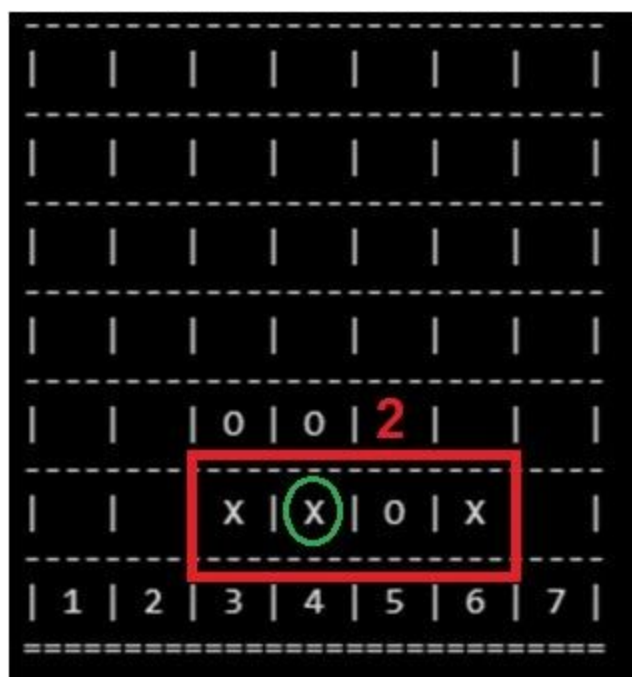
The working of rowCheck() is mentioned below:

(The token whose permutations are checked is mentioned in green circle)

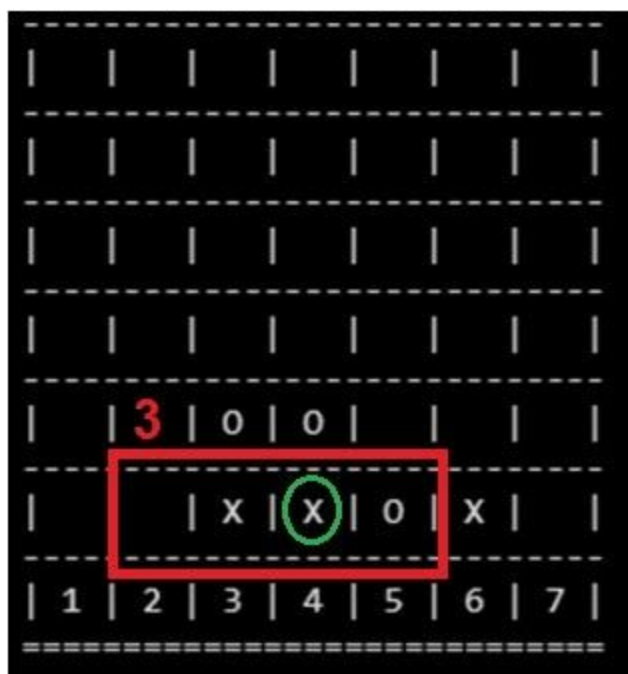
Let say the score of below mentioned state is **score\_1 = 0** (as per heuristics)



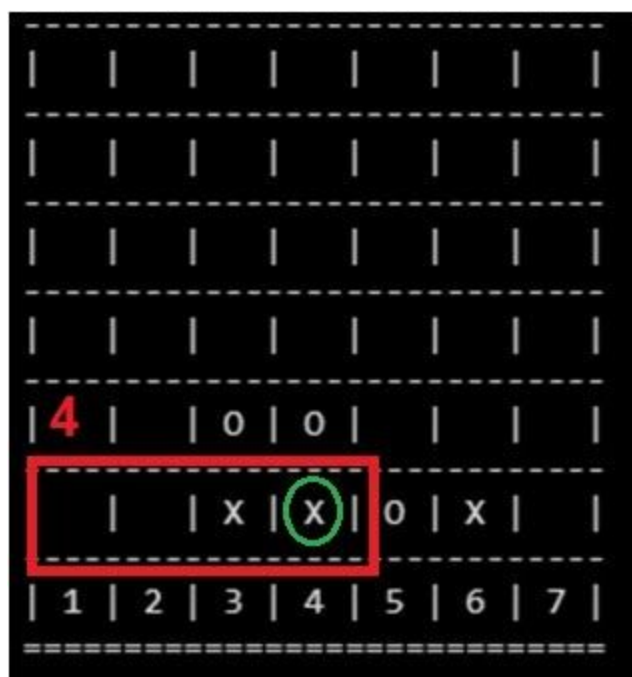
score\_2 = 0



score\_3 = 0



score\_4 = 2



Hence, the final score turns out to be

final\_score = score\_1 + score\_2 + score\_3 + score\_4

final\_score = 2

The final\_score's value is returned to evaluateBoard() by rowCheck().

The same concept of checking all permutations of a token in a column and both diagonals, and assigning a score, is done by checkCol() and checkDiag1() & checkDiag2(), respectively.

-----

## Code

```
# *****
# Crafted from scratch by      #
# 17BCE010 Bharat Adhyaru      #
# 17BCE014 Khodidas Chauhan    #
# 17BCE015 Chinmaya Kapopara   #
# *****

#global variables
player = 'X'
opponent = 'O'
moves_count = 0
best_row = -1
best_col = -1
winner = None
def printInstructions():
    print(" -----")
    print("|      Rules      |\n -----")
    print("Rule 1: Win by connecting four tokens in a row, column or
any of the diagonals")
    print("Rule 2: Don't let AI win :)")
    print("\n -----")
    print("|  Instructions  |\n -----")
    print("You = \'X\'\nAI = \'O\'")
    print("You go first\n")
    print("\n*****")
    print("|  Connect Four  |\n*****\n")
#-----
```

```

def printBoard(board):
    for i in range(6):
        print("-----")
        print("|", end="")
        for j in range(7):
            print("",board[i][j], end = "")
            print(" |", end="")
        print()
        print("-----")
        print("| 1 | 2 | 3 | 4 | 5 | 6 | 7 |")
        print("=====\\n")
#-----

# Improving heuristics because the program cannot run
# on its max recursion limit(10**9)
# Because, legal moves present at every depth is 7 (b = 7)
# and max depth that can be achieved is 42 (m = 42)
# Hence, the time complexity is  $O(b^m) = O(7^{42}) = 3.1197348e+35$  !!!
# Hence, the search needs to be limited to a certain depth
# but still we need to find the best move
# So the heuristics have been improved in the following way
# for row:
# return 0 if found as |X|_|0|X| or |X|0|_|_| and so on...
# return 1 if found as |X|_|_|_| (Note: the remaining three entries
must be empty)
# return 2 if found as |X|X|_|_|
# return 5 if found as |X|X|X|_|
# same applies to columns and diagonals
def evaluateTokenCount(array, main_token):
    token_count = 0
    empty_count = 0
    #count both
    token_count = array.count(main_token)
    empty_count = array.count(' ')

    if token_count == 4:
        found |X|X|X|X|
        return 100
    elif token_count == 3 and empty_count == 1:
        # return 100 if
        # return 5 if

```



```

found as |X|X|X|_| or |X|X|_|X| and so on
    return 5
    elif token_count == 2 and empty_count == 2:      # return 2 if
found as |X|X|_|_| or |X|_|X|_| and so on
    return 2
    #elif token_count == 1 and empty_count == 3:      # return 1 if
found as |X|X|_|_| or |X|_|X|_| and so on
    #return 1
    else:
        return 0                                     # cases like
|X|_|0|_| or |X|0|_|_| and so on
#-----

def checkRow(board, r, c):
    # make subarrays to check for four consecutive places for all the
four subarrays
    # say row = |X|_|_|X|_|0|_| and our c = 4 (board[r][c] = X),
referring this token as MX
    # so four different possibilities that need to be checked are:
    # 1. |X|_|_|MX|
    # 2. |_|_|MX|_|
    # 3. |_|MX|_|0|
    # 4. |MX|_|0|_|
    # because every possibility adds (or reduces) the score and all
of them contributes to score
    # so the final score, for this function would be sum of all
possibilities
    score = 0
    for j in range(4):
        start = c-j
        end = start+3
        if start >= 0 and end < cols:
            row_array = [board[r][i] for i in range(start, end+1)]
            score += evaluateTokenCount(row_array, board[r][c])
#passing main token
    return score
#-----

def checkCol(board, r, c):

```

```

    # make subarrays to check for four consecutive places for all the
    four subarrays

```

```

    score = 0
    for j in range(4):
        start = r+j
        end = start-3
        if start < rows and end >= 0:
            col_array = [board[i][c] for i in range(start, end-1,
-1)]
            score += evaluateTokenCount(col_array, board[r][c])
    return score
#-----

```

```

def checkDiag1(board, r, c):

```

```

    # make subarrays to check for four consecutive places for all the
    four subarrays for one diagonal - top left to bottom right

```

```

    score = 0
    for j in range(4):
        start_row = r-j
        start_col = c-j
        end_row = start_row + 3
        end_col = start_col + 3
        if start_row >= 0 and end_row < rows and start_col >= 0 and
end_col < cols:
            diag1_array = [board[start_row+i][start_col+i] for i in
range(4)]
            score += evaluateTokenCount(diag1_array, board[r][c])
    return score
#-----

```

```

def checkDiag2(board, r, c):

```

```

    # make subarrays to check for four consecutive places for all the
    four subarrays for other diagonal - bottom left to top right

```

```

    score = 0
    for j in range(4):
        start_row = r-j
        start_col = c+j
        end_row = start_row + 3
        end_col = start_col - 3

```



(max\_score - min\_score), the winning and defending is almost equal  
 # experiment with the  
 division value and you will get what we mean

#-----

```
def anyMovesLeft(board):
    global rows, cols
    for i in range(rows):
        for j in range(cols):
            if board[i][j] == ' ':
                return True
    return False
```

#-----

```
def winning_move(board, token):
    # Check horizontal locations for win
    for c in range(cols-3):
        for r in range(rows):
            if board[r][c] == token and board[r][c+1] == token
and board[r][c+2] == token and board[r][c+3] == token:
                return True
```

```
    # Check vertical locations for win
    for c in range(cols):
        for r in range(rows-3):
            if board[r][c] == token and board[r+1][c] == token
and board[r+2][c] == token and board[r+3][c] == token:
                return True
```

```
    # Check positively sloped diagonals for win
    for c in range(cols-3):
        for r in range(rows-3):
            if board[r][c] == token and board[r+1][c+1] == token
and board[r+2][c+2] == token and board[r+3][c+3] == token:
                return True
```

```
    # Check negatively sloped diagonals for win
    for c in range(cols-3):
        for r in range(3, rows):
```

```

        if board[r][c] == token and board[r-1][c+1] == token
and board[r-2][c+2] == token and board[r-3][c+3] == token:
            return True
#-----

def minimax(board, depth, is_maximising_player, alpha, beta):
    global player, opponent, moves_count
    moves_count += 1
    score = 0

    if winning_move(board, opponent):
        return 10000 - depth          #for faster victory
    elif winning_move(board, player):
        return -10000 + depth         #for slower draw/defeat

    if anyMovesLeft(board) == False:
        return 0

    #limiting to a certain depth
    if depth == 5:                    #It will check at most 7^6 moves
        (depth 0 is also included) -> 7^6 = 117,649
        return evaluateBoard(board)

    #if AI's turn
    if is_maximising_player == True:
        best_val = -1000000
        val = -1
        for j in range(cols):
            for i in range(rows):
                if board[i][j] == ' ' and i < rows-1: #finds first
non-empty cell
                    continue
                elif i > 0:
                    if board[i][j] != ' ':
                        i -= 1
                    board[i][j] = opponent
                    val = minimax(board, depth + 1, False, alpha,
beta)

                    best_val = max(best_val, val)

```

```

        alpha = max(alpha, best_val)
        #undo
        board[i][j] = ' '
        #Pruning step
        if beta <= alpha:
            return best_val
        #go for next column
        break
    if i == 0:        #if first row is non-empty go to next
column
        break
    return best_val

    #if player's turn
    if is_maximising_player == False:
        best_val = 1000000
        val = -1
        for j in range(cols):
            for i in range(rows):
                if board[i][j] == ' ' and i < rows-1: #finds first
non-empty cell
                    continue
                elif i > 0:
                    if board[i][j] != ' ':
                        i -= 1
                    board[i][j] = player
                    val = minimax(board, depth + 1, True, alpha,
beta)

                    best_val = min(best_val, val)
                    beta = min(beta, best_val)
                    #undo
                    board[i][j] = ' '
                    #Pruning Step
                    if beta <= alpha:
                        return best_val
                    #go for next column
                    break
            if i == 0:        #if first row is non-empty go to next
column

```

```

        break
    return best_val
#-----

def findBestMove(board):
    global best_row, best_col, opponent, cols, rows
    best_val = -10000000
    # alpha & beta
    alpha = -10000000
    beta = 10000000

    #try to find the best move by placing token in every column
    for j in range(cols):
        val = 0
        #find last empty cell in that column
        for i in range(rows):
            if board[i][j] == ' ' and i < rows-1: #finds first
non-empty cell
                continue
            elif i > 0: #could enter if
board[i][j] != ' ' or i == 5: and do nothing if first cell is
occupied
                if board[i][j] != ' ': #first non-empty
cell, could be i == 5 as well
                    i -= 1 #go to previous
empty cell
                # if board[i][j] == ' ': #last cell of col
empty i == 5

                #do nothing
                board[i][j] = opponent
                #uncomment below three statements to visualize
working of Minimax
                #----- Visualizer -----
                #print("====Before====")
                #printBoard(board)
                #print("i,j =",i,j," | val =",val)
                #-----
                #call minimax
                val = minimax(board, 0, False, alpha, beta)

```

```

        # uncomment below line to visualize just heuristics
        print("col,row =",j+1,i+1," | val =",val)
        #undo move
        board[i][j] = ' '
        if val > best_val:
            best_row = i
            best_col = j
            best_val = val
        break

    if i == 0:        #if first row is non-empty go to next
column
        break
    board[best_row][best_col] = opponent
#-----

#-----
#main()
rows, cols = (6, 7)
#initializations
board = [[' ' for j in range(cols)] for i in range(rows)]
printInstructions()
printBoard(board)
player_col = 0
winner = None

# Game loop
while anyMovesLeft(board) == True:
    player_col = input("Your turn: ")
    #invalid input if not a number
    if not player_col.isnumeric():
        print("Invalid input!! Enter choice from 1 to 7. Try
again.\n")
        continue
    player_col = int(player_col)

    #invalid index
    if player_col > 7 or player_col < 1:
        print("Invalid column index!! Enter choice from 1 to 7. Try

```



```

again.\n")
    continue
    player_col -= 1          #indexing starts from 1 for the consumer

    #invalid move
    if board[0][player_col] != ' ':          #topmost row of that
column not empty
        print("Invalid move!! Column not empty. Try again.\n")
        continue

    #insert into that cols
    for i in range(rows):
        if board[i][player_col] == ' ' and i < rows-1: #finds first
non-empty cell
            continue
            elif i > 0:                                #could enter
if board[i][j] != ' ' or i == 5: and do nothing if first cell is
occupied
                if board[i][player_col] != ' ':          #first
non-empty cell, could be i == 5 as well
                    i -= 1                                #go to
previous empty cell
                board[i][player_col] = player
                break
    print("\n      -----")
    print("      |   Your Move   |\n      -----")
    printBoard(board)
    #check for winner
    if winning_move(board,player):
        winner = player
        break

    #AI's turn
    print("      -----")
    print("      |   AI's Move   |\n      -----")
    findBestMove(board)
    printBoard(board)
    print("Moves checked by AI
=",moves_count,"\n-----\n")

```

```

moves_count = 0
#check for winner again
if winning_move(board,opponent):
    winner = opponent
    break

if winner == player or winner == None:
    printBoard(board)

print("\n*****")
print("|   Game Over   |\n*****\n")
if winner == opponent:
    print("You Lost!!")
    print("Better luck next time :)")
elif winner == player:
    print("You Won!!")
    print("AI: Hats off to You. You defeated me")      #This is never
gonna happen :)
else:
    print("Game Tied!!\n")
    print("AI: Told you, I'm unbeatable ;)")

# The most important thing we noted is that, heuristics is everything
for this kind of board games
# If we change the heuristic values in evaluateTokenCount(), the
moves of AI will change
# Same applies to return statement of evaluateBoard()
# because alpha-beta pruning and minimax at last depends on the score
they recieve

# *****
# Crafted from scratch by      #
# 17BCE010 Bharat Adhyaru      #
# 17BCE014 Khodidas Chauhan    #
# 17BCE015 Chinmaya Kapopara   #
# *****

```

---

## Conclusion

In this innovative assignment, we applied our knowledge of the Minimax algorithm and Alpha-Beta Pruning algorithm to develop the Connect Four game from scratch. While developing the game, we did some rigorous testing of how AI is behaving in different situations which in turn taught us what is the importance of heuristics and how differently AI reacted by a slight change of heuristics value. Hence, we learnt that the main hero for AI based board games is heuristics.

-----

## References

- [1] Minimax Algorithm in Game Theory - <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/>
  - [2] Evaluation Function in Game Theory - <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/>
  - [3] Minimax Algorithm in TicTacToe - <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/>
  - [4] Alpha-Beta Pruning in Game Theory - <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>
  - [5] Connect4 in Python by Keith Galli - <https://github.com/KeithGalli/Connect4-Python>
  - [6] Rijul Nasa<sup>1</sup>, Rishabh Didwania<sup>2</sup>, Shubhranil Maji<sup>3</sup>, Vipul Kumar<sup>4</sup> “Alpha-Beta Pruning in Minimax Algorithm –An Optimized Approach for Games”
  - [7] Ahmad M. Sarhan, Adnan Shaout and Michele Shock “Real-Time Connect 4 Game Using Artificial Intelligence” University of Michigan-Dearborn, USA
- 

Made by  
**17BCE010 Bharat Adhyaru**  
**17BCE014 Khodidas Chauhan**  
**17BCE015 Chinmaya Kapopara**