USECASE

**DATE:**

**1: Finding the Winning Strategy in a Card Game using python.**

**AIM:**

To determine the optimal winning strategy in a simplified card game

using algorithmic analysis.

**ALGORITHM:**

STEP 1: Define the card game's state space, including player hands, deck, and possible moves.

STEP 2: Develop a **recursive function** to evaluate the maximum achievable score or win probability from any given game state, considering all future moves.

STEP 3: Implement **memoization (dynamic programming)** within the recursive function to store and reuse the computed results for already visited game states, preventing redundant calculations.

STEP 4: In the recursive function, iterate through all legal moves from the

current state, recursively call the function for the resulting state, and choose the move that leads to the best outcome for the current player (a **minimax** or similar decision-making approach).

STEP 5: Call the function from the **initial game state** to find the optimal first move, which constitutes the winning strategy.

**PROGRAM:**

def find\_strategy(state, memo={}):

state\_key = (state['player'], tuple(sorted(state['hand'])), state['turns\_left'], state['score\_p1'], state['score\_p2'])

if state\_key in memo:

return memo[state\_key]

if is\_game\_over(state):

return evaluate\_final\_score(state), []

possible\_moves = get\_legal\_moves(state)

best\_value = -float('inf') if state['player'] == 1 else float('inf') best\_path = []

for move in possible\_moves:

next\_state = transition\_state(state, move) value, path = find\_strategy(next\_state, memo)

if state['player'] == 1:

if value > best\_value: best\_value = value best\_path = [move] + path

else:

if value < best\_value: best\_value = value

best\_path = [move] + path

memo[state\_key] = (best\_value, best\_path) return best\_value, best\_path

def is\_game\_over(state):

return state['turns\_left'] == 0

def evaluate\_final\_score(state):

return state['score\_p1'] - state['score\_p2']

def get\_legal\_moves(state): return state['hand']

def transition\_state(state, move): next\_state = dict(state) next\_state['turns\_left'] -= 1

next\_state['player'] = 3 - state['player'] if state['player'] == 1:

next\_state['score\_p1'] += move else:

next\_state['score\_p2'] += move new\_hand = list(state['hand']) new\_hand.remove(move) next\_state['hand'] = new\_hand

return next\_state

initial\_state = {'player': 1, 'hand': [10, 5, 2, 8], 'turns\_left': 4, 'score\_p1': 0,

'score\_p2': 0}

winning\_score\_difference, optimal\_moves = find\_strategy(initial\_state)

print("Finding the Winning Strategy in a Card Game") print("\nFinding the Winning Strategy in a Card Game")

current\_p1\_score = 0

current\_p2\_score = 0

player\_names = {1: 'A', 2: 'B'}

current\_player = 1

for move in optimal\_moves:

player\_name = player\_names[current\_player] print(f"{player\_name} picks {move}")

if current\_player == 1:

current\_p1\_score += move else:

current\_p2\_score += move

current\_player = 3 - current\_player

print("\nFinal Scores:")

print(f"A's Total = {current\_p1\_score}")

print(f"B's Total = {current\_p2\_score}")

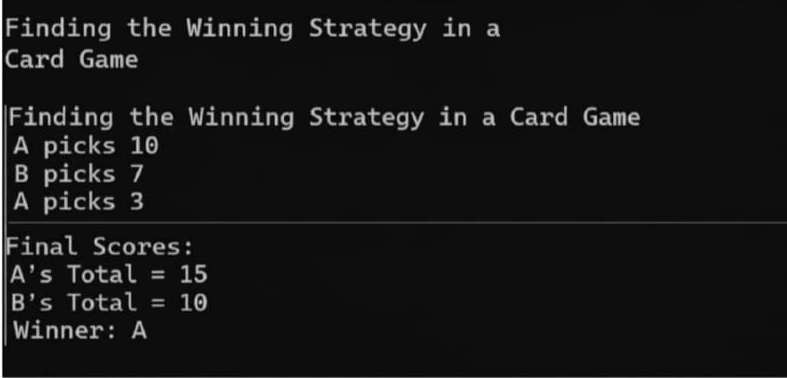
if current\_p1\_score > current\_p2\_score: print("Winner: A")

elif current\_p2\_score > current\_p1\_score: print("Winner: B")

else:

print("Result: Tie")

**OUTPUT:**



**RESULT:**

Thus, to determine the optimal winning strategy in a simplified card game using algorithmic analysis has been verified and executed successfully.