DATE: 29/10/25 13 . USECASE

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
```

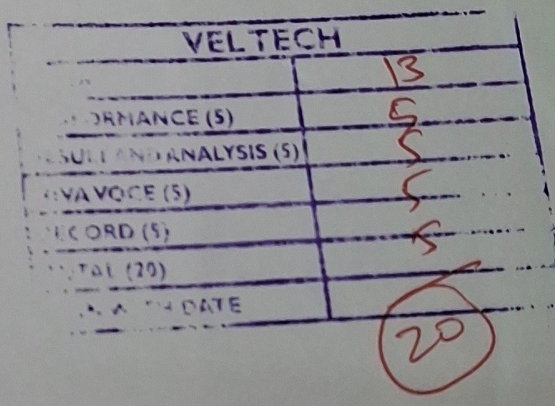
```
            best_path = [move] + path

    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:

```
Finding the Winning Strategy in a
Card Game

Finding the Winning Strategy in a Card Game
A picks 10
B picks 7
A picks 3

Final Scores:
A's Total = 15
B's Total = 10
Winner: A
```



RESULT:

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