USE CASE : 1

use case:- finding the winning strategy in a card game in python

Problem description:

Imagine a card game where each player receives a hand of cards with values. The objective is to find the best way to maximize the score for a player, assuming the players take turns drawing cards. Each player can either pick the first or last card from remaining pile.

Assumptions:-

- Each player tries to maximize their score
- cards are represented by integers, which indicate their values.
- Two players alternate turns, and each player picks a card from either the begining or end of list.

You need to design an algorithm that helps a player find optimal strategy to guarantee the highest possible score given that opponent is also playing optimally.

Plan:-

We can salve this problem using pynamic programming by calculating the optimal score for every possible scenario, taking into account best choices for both players.

Steps:-

1. Define the Game:- Represent the pile of cards as a list of integres.

2. Recursive strategy:- A function will recursively determine the best score a player can achieve.

3. Dynamic programming (memoization):- Store intermediate results to avoid recalculating them.

4. Base cases:- when only one card is left, the current player takes it.

Program:-

```
def find_optimal_strategy(cards):
    n = len(cards)
    # create a memoization table to store subproblem results.
    dp = [0] * n for _ in range[n)]
    # fill the table for subproblems of increasing sizes
    for length in range (1, n+1):
        for i in rang (n-length +1):
            j = i + length - 1
            # if only one card is left, the player takes it if i == j:
                dp [p][j] = cards[i]
            else:
                # Choose the best of two choices:
                # 1. take the left card, and the opponent plays optimally on the remaining (i+1, j)
                # 2. take the right card, and opponent plays optimally on the remaining (i, j-1)
                take_left = cards [i] _ dp [i+1][j]
                take_right = cards [j] - d [j][j-1]
                dp[i][j] = max (take_left, take_right)
    # dp[0] [n-1] will have the optimal score difference
    fo first player return (dp[0][n-1] + sum(cards))
```

// First player's maximum possible score
// Example case.
cards = [3,9,1,2]
Print("First player's optimal score:", Find-optimal-Strategy(cards))

Explanation:-

- Dynamic programming Table (dp):- Each cell dp[i][j] represents the difference in score between the first player and the opponent if game is played between cards form index i to index j.

Two choice:- for each move the player can either:

1. Pick the leftmost card cards[i], leaving opponent to play optimally on remaining cards.
2. Pick the right most cards[j], leaving opponent the rest of the cards.

- Recursive Relation:- The value of each subproble is determined by maximizing the score difference between the current player and the opponent.

Example walk through:

Consider the array of cards: [3,9,1,2]:

1. First player (you) can choose between:
   - Taking leftmost card (3), leaving cards [9,1,2];
   - Taking right most card (2), leaving cards [3,9,1];
2. The opponent will then their turn, playing optimally to minimize the first player's score

This program computes the best possible outcome for the first player.

First player's optimal score: 5

First player if playing optimally, can guarantee a score of 5 regardless of how the opponent plays.

optimizing strategy:

By using Dynamic programming, we ensure that the solution is computed efficient, avoiding redudant calculations. This approach ensoures both players play optimally, and first player gets the highest score possible given opponent's best move.