In this case, we have to devise an optimal strategy for the player 1. I'm assuming that the player 2 would be using the same optimal strategy to minimize player 1's value (profit). This probably would result in a strategy similar to minimax as it is a zero-sum game as player 1 has to maximize his profit while player 2 has to minimize player 1's profit.

A polynomial time function is one whose time complexity is a polynomial function of the input size.

Optimal Substructure

The optimal solution would contain sequence of cards which give max value. The optimal substructure exists as the value of root depends on the value of the subtrees (left and right).

Choice

Let i…. j be the sequence of cards, i being the first card and j being the last card in the deck or sequence.

Player 1 has two choices:

i) Pick from left side, i.e. picks the ith card:

In this case, player 2 would have to choices and we have to take the minimum of the two expressions.

(a) Player 2 picks from the left side as well.
vi (value of card picked by player 1) + cards (arr, i+2, j)

- i + 2 is done because first player 1 picks from left side and then player 2 does the same thing.

(b) Player 2 picks from the right side.

vi (value of card picked by player 1) + cards(arr, i+1, j-1).

ii) Pick from right side, i.e. picks the jth card:

In this case, player 2 would have to choices and we have to take the minimum of the two expressions.

(a) Player 2 picks from the left side.
vj (value of card picked by player 1) + cards (arr, i+1, j-1)

(b) Player 2 picks from the right side as well.
vj (value of card picked by player 1) + cards(arr, i, j-2).

- j - 2 is done as player 1 and player 2 have both picked from the right side.

Recursive

Base-Case:

There are two base cases in this problem. They occur when:

i) Only one card is left in the deck, i.e. i == j.

If i == j
Return array[i] *//jth value could be returned as well.*

ii) There are two cards left to be picked, i.e. j – 1 == 1. In that case we have to return the one with maximum value.

If j – i == 1
Return max (array[i], array[j]) *// return maximum of the two.*

Otherwise:

According to the choice scenarios given above, maximum of the two scenarios has to be returned as we are optimizing for player 1.

```
return max(array(i) + min(recursive(array, i + 2, j), recursive(array, i + 1, j - 1)),
array(j) + min(recursive(array, i + 1, j - 1), recursive(array, i, j - 2)));
```

Note:

- recursive is the name of the method.
- Cards are stored in array.

Bottom-Up:

In different two-dimensional dynamic programming problems, the optimal value is stored at the last index of the matrix. However, it is different in this case as the base-cases are different. It is stored at Dp[0][n-1], where Dp is the matrix used to store values of the sub-problems and n is the number of cards in the game.

Code:

```
int bottomUp(vector<int> cards, int n) {

  int Dp[n][n];

  int x = 0, y = 0;

  //base-case #1. Only one card is left.
  for (int i = 0; i < n; i++)
    Dp[i][i] = cards.at(i); //storing it in the diagonal.

  //base-case #2. Two cards are left to choose from. We take the one with maximum value.
  for (int i = 0; i < n - 1; i++) {
```

```
        Dp[i][i + 1] = max(cards.at(i), cards.at(i + 1));

    }

    for (int i = 0; i < n; i++) {
        for (int j = i + 2; j < n; j++) { //starting j from i + 2 as j can't be lower than i. It is not
possible.
            x = min (Dp[i + 2][j], Dp[i + 1][j - 1]);

            y = min (Dp[i + 1][j - 1], Dp[i][j - 2]);

            Dp[i][j] = max(cards.at(i) + x, cards.at(j) + y);
        }
    }
    return Dp[0][n - 1];
}
```