Daily Coding Problem #220

Problem

This problem was asked by Square.

In front of you is a row of `N` coins, with values v_1, v_1, ..., v_n.

You are asked to play the following game. You and an opponent take turns choosing either the first or last coin from the row, removing it from the row, and receiving the value of the coin.

Write a program that returns the maximum amount of money you can win with certainty, if you move first, assuming your opponent plays optimally.

Solution

At each turn, we have the option of taking either the first or last coin. Whichever coin we choose, our opponent will be faced with the decision of which coin to take from the remaining `N - 1` coins. Then we will need to figure out which coin to take from the remaining `N - 2` coins, and so on. Since we are making the

same decision each turn, with modified input, this is a good use case for recursion.

Let's think through the base cases:

- If there is only one coin left, we have no choice but to take it.
- If there are two coins left, we should take the one with maximal value.

For the general case, let's denote the maximum guaranteed amount of money we can get from a game to be the "profit" of the game. Note that whichever coin we take, our opponent will follow up by taking the coin that maximizes her profit. So after removing our coin, we can only be guaranteed the minimal profit of `(row without first element, row without last element)`. Therefore, we should choose a coin such that we maximize the sum of the coin's value and the smaller of the profits of the rows returned after our opponent's next move.

To take an example, let's say we start out with `[10, 24, 5, 9]`.

- If we take `10`, our opponent can take either `24` or `9`. The smarter decision will be `24`, leaving us with `[3, 9]`.
- If we take `9`, our opponent can take either `10` or `5`. These decisions are equivalent, leaving us with `[24, lesser]`.
- Since `9 + max_profit([24, lesser]) > 10 + max_profit([3, 9])`, we should take the `9` coin.

Our recursive implementation of this is as follows:

```
def max_profit(coins, value):
    if len(coins) == 1:
        return value + coins[0]

    elif len(coins) == 2:
        return value + max(coins)

    else:
```

```
        return value + max(
            coins[0] + min(max_profit(coins[2:], value), max_profit(coins[1:-1],
value)),
            coins[-1] + min(max_profit(coins[:-2], value),
max_profit(coins[1:-1], value))
        )
```

Since we branch off into two decision at each turn, this has time complexity $O(2^N)$.

We can make this more efficient by using dynamic programming. Let's initialize an `N` by `N` upper triangular matrix, `profit`, such that `profit[i][j]` stores the profit that can be made from a row starting at coin `i` and ending at coin `j`. After filling in this matrix, our solution will be `profit[0][N - 1]`.

As above, we can first fill in our base cases:

- If we start and end at the same coin, our profit will be the value of that coin.
- If we only consider two coins, our profit will be the maximum of the two coin values.

Then we consider longer and longer rows of coins, beginning with three in a row and ending in the entire coin list. For each mini-row, we follow the algorithm described above: choose an end coin such that it maximizes the sum of its value and the minimal profit after our opponent's next move.

```
[10, 24*,  15,   ?]
[0,   24, 24*,  29]
[0,    0,   5,  9*]
[0,    0,   0,   9]
```

This may be easiest to see visually. To find the value of the last cell in our table, marked `?`, all we need to know is the value of the three starred cells, each located two moves away. The maximum profit we can achieve for this game will be the maximum of `10 + min(diagonal, bottom)` and `9 + min(left, diagonal)`, which is `33`.

Our dynamic programming solution will use $O(N^2)$ time and space.

```
def max_profit(coins, value):
    n = len(coins)
    profit = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        profit[i][i] = coins[i]

    for i in range(n - 1):
        profit[i][i + 1] = max(profit[i][i], profit[i + 1][i + 1])

    for gap in range(2, n):
        for i in range(n - gap):
            j = i + gap
            left = profit[i][j - 2]
            diagonal = profit[i + 1][j - 1]
            bottom = profit[i + 2][j]
            profit[i][i + gap] = max(
                coins[i] + min(diagonal, bottom),
                coins[j] + min(left, diagonal)
            )

    return profit[0][-1]
```