Edit distance

Your task is to determine the edit distance between two strings x and y, that is to find the cheapest way to convert x to y using only insert, delete, and replace operations. These operations cost c_i, c_d, and c_r.

Example

Suppose, $c_i = c_d = c_r = 1$. If x="apple" and y="sample", the cheapest way to turn x into y is to insert an 's' letter in the beginning of x and replace the first 'p' letter with an 'm' letter. Thus, the minimum cost solution (i.e., the edit distance) would be 2.

Question 1: You are given x and y, and the costs of insert, delete, and replace operations: c_i, c_d, and c_r. Use a top-down dynamic programming approach, write a function, EDITDISTANCE, to compute the the edit distance between these two strings. **Hint**: in each step, guess what operation is performed on suffixes of x and y.

Subproblem: $EditDistance(i, j)$: the edit distance between $x[i:]$ and $y[j:]$
Guess: 3 choices – replace $x[i]$ with $y[j]$, insert $y[j]$ in the beginning of $x[i]$, or delete $x[i]$

Recursive Relation:

$$EditDistance(i,j) = \begin{cases} (|y| - j) * ci, & \text{if } i = |x| \\ (|x| - i) * cd, & \text{if } j = |y| \\ \min(EditDistance(i{+}1, j{+}1) + c_r, & \\ \qquad EditDistance(i, j+1) + c_i, & , \quad \text{otherwise} . \\ \qquad EditDistance(i+1, j) + c_d\ ) & \end{cases}$$

here c_r is assumed to be zero when $x[i] = y[j]$.
Original Problem: $EditDistance(0, 0)$

```
def EditDistance(i, j, memo=None):
    if memo is None:
        memo = {}
    if not (i, j) in memo:
        if i==len(x):
            memo[(i, j)] = (len(y)-j)*ci
        elif j==len(y):
            memo[(i, j)] = (len(x)-i)*cd
        else:
            cost_of_insert = ci + EditDistance(i, j+1, memo)
            cost_of_delete = cd + EditDistance(i+1, j, memo)
            if x[i] == y[j]:
                cost_of_replace = EditDistance(i+1, j+1, memo)
            else:
                cost_of_replace = cr + EditDistance(i+1, j+1, memo)
            memo[(i, j)] = min(cost_of_insert, cost_of_delete, cost_of_replace)
    return memo[(i, j)]
```

Question 2: Analyze the running time of your code.

Let n and m be $len(x)$ and $len(y)$, respectively. The number of distinct subproblems: $O(n \cdot m)$
Computation per subproblem: $O(1)$ as we only take the minimum of three numbers in constant time
Total running time: $O(n \cdot m)$

Question 3: Write a function, EDITDISTANCEBOTTOMUP, to solve the same problem using a bottom-up approach. **Hint**: decide on the order in which you want to solve the subproblems and determine what needs to be stored in the table.

```
def EditDistanceBottomUp(x, y):
    table = [[0 for i in range(len(y))] for j in range(len(x))]
    for i in range(len(x)):
        for j in range(len(y)):
            if i == 0 or j ==0:
                if i > j:
                    table[i][j] = i*cd + (0 if x[i]==y[j] else cr)
                else:
                    table[i][j] = j*ci + (0 if x[i]==y[j] else cr)
            else:
                cost_of_insert = ci + table[i][j-1]
                cost_of_delete = cd + table[i-1][j]
                if x[i] == y[j]:
                    cost_of_replace = table[i-1][j-1]
                else:
                    cost_of_replace = cr + table[i-1][j-1]
                table[i][j] = min(cost_of_insert, cost_of_delete, cost_of_replace)
    return table[-1][-1]
```

Question 4: Analyze the running time of your code.

Each loop index (i and j) takes on at most n and m values. Thus, the total running time of this algorithm is $O(n \cdot m)$.