Dynamic Programming Sample Problem

1. Updated Edit Distance

You saw Edit Distance in class, let's tweak the problem a bit and see if we can still solve it with dynamic programming. Specifically, define the **Weighted Edit Distance** between x, y to be the following:

- Charge 2 points to delete a character from x
- Charge 3 points to insert a character into x
- Charge 4 points to substitute a character in x.

The Weighted Edit Distance is the minimum number of points required to transform x into y.

Sample input 1: Consider the instance with strings $x =$ cat and $y =$ bat.

$$\textbf{Weighted Edit Distance}(\text{cat}, \text{bat}) = 4$$

Sample input 2: Consider the instance with strings $x =$ abcdef and $y =$ azced. It is the case

$$\textbf{Weighted Edit Distance}(\text{abcdef}, \text{azced}) = 10$$

since we substitute b into z for a cost of 4, delete the d from x for a cost of 2 and substitute f into d for a cost of 4, thereby giving a total cost of $4 + 2 + 4 = 10$.

(a) In class, we said for the regular edit distance that the distance between x and y was the same as the distance between y and x. Is that still true? Briefly explain. **Solution:**

Because deletion and insertion cost different amounts, this claim is no longer true. Consider $x = abcd$ and $y = abc$. Transforming x to y is a deletion (cost 2); transforming y to x is an insertion (cost 3).

(b) Write an **English** description of what the recurrence is calculating (especially what the parameter(s) in your recurrence represent) and state the settings of the parameter(s) for the weight edit distance between x, y
Solution:

$\mathsf{OPT}(i, j)$ is the minimum number of insertions, deletions and substitutions to transform the string $x_1, \ldots, x_i$ to $y_1, y_2, \ldots, y_j$ respecting the above point penalties (where deleting is the cheapest and substitution is the most expensive).

(c) Write a recurrence to calculate the weighted edit distance between two strings. (Do not write code, just the recurrence).
Solution:

Note here that it is possible to cleanly represent the base case in 2 cases as opposed to having 3, but this kind of clear separation makes things easier to understand :)

$$\mathsf{OPT}(i,j) = \begin{cases} 0 & i = j = 0 \\ 2i & j = 0, i > 0, \text{ (deleting } i \text{ characters)} \\ 3j & i = 0, j > 0, \text{ (inserting } j \text{ characters)} \\ \min \begin{cases} \mathsf{OPT}(i-1,j) + 2 & \text{(delete)} \\ \mathsf{OPT}(i,j-1) + 3 & \text{(insert)} \\ \mathsf{OPT}(i-1,j-1) + 4 \cdot \mathbb{I}[x_i \neq y_j] & \text{(substitute)} \end{cases} & i, j > 0 \end{cases}$$

(d) What memoization structure would you use to evaluate your recurrence?
Solution:

Let n denote the length of x and m denote the length of y. Then our memoization structure `memo` can be represented using a 2D array with dimensions $n \times m$, where $\texttt{memo}[i][j] = \mathsf{OPT}(i, j)$.

(e) Give a filling order for your memoization structure (you do not have to give the pseudocode to fill it, just an order you could fill it in).
Solution:

The outer loop runs from 1 to n, and the inner loop runs from 1 to m. The final answer to the problem is simply given by $\mathsf{OPT}(n, m)$.

(f) What would the running time of (iterative) code be? Let n be the length of x and m be the length of y.
Solution:

Since we're filling up an array of shape $n \times m$ where each entry takes constant work, the running time is given by $\mathcal{O}(mn)$.