CMPS 2200 Recitation 08

Today we'll program more about Edit Distance. As usual, code goes in `main.py`

In class we proved an optimal substructure property for the **Edit Distance** problem. This allowed us to implement a simple recursive algorithm in Python that was horribly inefficient. We're going to implement an efficient memoization scheme and then implement a way to visualize the optimal sequence of edits.

1) The code for `MED` from the lecture notes is provided as a starting point in `main.py.` We will consider the edit distance problem which allows for insertions and deletions. We will assume that insertions and deletions have the same unit cost. State the optimal substructure property for this version of the edit distance problem and modify `MED` accordingly.

2) Now implement `fast_MED`, a memoized version of `MED`. Test your implementation code using `test_MED`.

3) Now that you have implemented an efficient algorithm for computing edit distance, let's turn to the problem of identifying the actual edits between two sequences.

Notice that in the process of computing the optimal edit distance, we can also keep track of the actual sequence of edits to each position of S and T. Update your implementation of `fast_MED` to return the optimal edit distance as well as an *alignment* of the two strings which show the edits that yield this distance. An alignment just shows what changes are made to S to transform it to T. For example, suppose S=`relevant` and T=`elephant`. If insertion and deletion costs are all equal to 1, then the edit distance between S and T is 4 and an alignment of these two strings would look like this:

```
relev--ant
-ele-phant
```

Implement `fast_align_MED` to return the aligned versions of S and T, and test your code with `test_alignment`.