**How you can break down a large problem instance into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the sub- problems.**

The first question that needs to be realized is whether the problem has an overlapping substructure. We can break up the problem by first comparing the two strings and analyzing the various costs and storing those costs in a 2D matrix. The second part of the problem will require us to name the typos and where they occur in the typo string. This can be done with dynamic programming, since the optimal costs have already been stored in the matrix, instead of traversing through the strings and comparing each character again to discover what typo is present, we can work backwards within the matrix to reconstruct the decisions made at every step along the optimal path within the matrix to discover the typo that lead to that cost.

**What should the parameters be for your recursive function?**

We need a recursive function to go through the previously calculated optimal cost within the matrix and retrace our steps to decide what typo is present within the string based on the cost.

**What recurrence can you use to model this problem using dynamic programming?**

|  |  |
| --- | --- |
| cost(An,Bm) = minimum( | |
|  | cost(An-1,Bm-1) + substitution(An,Bm); |
|  | cost(An,Bm-1)   + insertion(Bm); |
|  | cost(An-1,Bm)   + deletion(An); |
|  | cost(An-2,Bm-2) + transposition(An,Bm); |
|  | ) |

**What are the base cases of this recurrence?**

If both strings are empty, we know we will not need to calculate costs, and there will be no changes within the two strings. If the target is an empty string, then we are not able to compare the typo strong to anything.

**What data structure would you use to recognize repeated problems? You should describe both the abstract data structure, as well as its implementation.**

Using a 2D matrix we can store what the cost is to transform the target string to the typo string given by comparing the characters in both strings. Using the reccurance, we will be able to fill in every cell in the matrix by comparing each cell to its neighboring cells and choosing the minimum cost of those various cells. It is important to note that the strings must be entered in reverse, meaning that the character comparisons will begin with the last character in each string and going until the first character. Reversing the strings will allow successful backtracking within the matrix once the lowest possible cost is totaled allowing us to work from the last cell in the matrix that allowed for the minimum cost, back to the first minimum cost cell that we began with.

**Give pseudocode for a memoized dynamic programming algorithm to solve the problem. Your pseudocode should not describe how to compute the cost for each possible typo.**

**Input**: Target String, A with n characters

**Input**: Typo String, B with m characters

**Output**: Minimum cost to transform target string into typo string and sequence of typos within this cost

**Algorithm**: TypoDeterminaterInitializer

1. M= Matrix (n+1, m+1);
2. Initialize all cells of M to NIL
3. Return TypoDeterminater(A, B)

**Input**: Target String, A with n characters

**Input**: Typo String, B with m characters

**Output**: Minimum cost to transform target string into typo string and sequence of typos within this cost

**Algorithm**: TypoDeterminater



**What is the worst-case time complexity of your memoized algorithm? Give a set of nested loops that could be used in an iterative dynamic programming solution to this problem.**

**Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm? Justify your answer.**

**Describe at least one advantage and disadvantage of the iterative algorithm, and indicate which implementation you believe is better overall.**

An iterative solution would compare each character in both the target and the typo string, then decide whether the two characters are a match, a substitution, a deletion, or a transposition. After that is determined, the algorithm would have to determine what the cost would be each time, and if the cost depended on the distance between keys, the distance function would need to be used to calculate cost. Overall, an iterative solution would be *extremely* slow, since each character comparison would have to go through multiple steps to find the cost and what kind of typo is present. If the string is too long, the algorithm itself would have to go through hundreds of comparisons before an answer could be produced. One advantage however, would be that the solution would be simpler to understand and probably quicker to implement in code. I think it makes more sense to solve the problem dynamically rather than iteratively for the sake of time, and the ability to implement the algorithm on much longer strings while still receiving an accurate result.