***Dynamic Programming***

**COT 4400, Summer 2017**

**Kathelin Castano, Bhavika Shah, Zahra Simpson**

**Introduction**

In this project, the objective was to develop an advanced algorithm that would identify the most likely sequence of typos a text underwent in the process of being typed. Using a dynamic programming model, the developed algorithm successfully accepts two strings, a typo string and a target string, then computes the minimum cost required to transform the target string into the typo string as well as a sequence of typos with this cost. The types of typos to be recognized in the string included insertions, deletions, substitutions and transpositions. These typos, as well as where they fall in the strings are included in the program output.

**Questions**

1. **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. It is also important to analyze where each given character in the typo string falls on the standard QWERTY keyboard, as different characters have different typos costs depending on the location of the keys. 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. We can further break up the first problem of filling in our cost matrix by recognizing that in each typo case, we must keep computing the “edit distance” problem as one or both strings get smaller and smaller until we hit a base case and we can stop the computation.

1. **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, referred to as DP that compares strings A and B, and retrace our steps to decide what typo is present within the string based on the cost. The recursive function needs to be able to compare the various characters in the typo string and the target string and recursively calculate the costs to edit the characters depending on whether the character has been substituted, inserted, deleted or transposed. The parameters of the recursive function will include which cell of the matrix we are currently in and how it will be traversed depending on the comparisons between the two strings. We know that within the matrix we must refer to the cell above, behind and diagonally above the cell we are currently in. We must refer to the matrix DP at position [A-1, B], DP at position [A, B-1], DP at position [A-1, B-1] and DP at position [A-2, B-2]

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

Where A and B are the target and typo strings, *n* and *m* are the lengths of the two strings respectively, and DP is a matrix of size n+1, m+1.

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

1. **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.

1. **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.

1. **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. DP = Matrix (n+1, m+1);
2. Initialize all cells of M to -1 (sentinel value?)
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**: MemoizedTypoDeterminater

**If** A=B=NULL **then**

**Return** 0;

**Else if** A = NULL **then**

**Return** 0;

**If** DP[An, Bm] > 0

DP[An, Bm] = minimum MemoizedTypoDeterminater (An-1,Bm-1 + substitution(An,Bm), An,Bm-1   + insertion(Bm), An-1,Bm   + deletion(An), An-2,Bm-2 + transposition(An,Bm));

**Return** DP[An, Bm]

1. **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.**
2. **Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm? Justify your answer.**
3. **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.

**Questions**

**Max length for sentences email Hendrix**

**Why is space not included in the keyboard in the project guidelines ask Hendrix**

**Backtracking within the matrix?**

**Implemement distance**

**Iterative solutions**

**Replace recursive calls with lookup and may be able to reduce space complexity**