Project 2 Report

David Decker, Thomas Pannozzo, Panos Valavanis

The objective of this project is to develop a program that will identify the most likely sequence of typos a string underwent while being typed. This is done through comparing a "target" string with a "typo" string character by character, and determining the type of typo that the string had. There are four types of typo instances we must account for: inserting, deleting, substituting, and transposing. Each instance has its own cost based on key placement, hand placement, and spaces. The typo string tests whether more than one type of instance occurred, and chooses the one with the minimum cost. To do this correctly, we developed a cost function which would run the instances. The main issue of this problem lies in the implementation: how can we accurately describe the typo while having a small time-complexity. The solution to this is a memoized recursive algorithm which stores computed costs, so they are not recomputed when the same instance is encountered from a recursive call.
Below, are the answers for the questions from the project document:

1. One way a large problem can be broken down into smaller sub-problems is by case analysis. Each case needs to be solved to either get a value, or be used to help solve the next case. Putting these solved cases together solves the main problem. In our case, our main problem to solve is the cost of the typo in the string. To solve this, we broke it down into subcases which were the type of typos. For example, if the end of the target string has been reached and there are additional characters left in the typo string, the subcase requires an insertion. Identifying subcases that required a definite action simplified later steps. After identifying each subcase, we broke those cases down to the cost of the characters encountered. Putting all of this together, we are able to calculate the cost of the typo and the type of typo.
2. The parameters for our recursive function should be the target and typo strings, and the indices of our strings.
3. Each call of the function contains two indices which travel the strings character by character, making comparisons at each step. The recurrence we used to model this problem is based on the minimum value of three recursive calls: one that increments the index travelling the target string, one that increments the typo string's index, and one that increments them both together. Every recursive call will branch the function in three directions, and then the minimum of these three calls will be chosen as the most optimum route to be saved in the memoized data structure.
4. We have four base cases in which the function in which the function will cease recursion: when the algorithm finds a memoized entry in the data structure it immediately returns it, if the indices are both at the end of their strings the function will terminate and return, if the target string has terminated but the other is not it will call insertions until they've both terminated, and if the typo string has terminated but the other has not it will call deletions until they've both terminated.
5. To recognize repeated problems, we implemented a 2D array. Our algorithm compares every possible pair of characters between the strings with each other to find the optimal route, and a two-dimensional array is a data structure which allows easy access to such pairs. One axis of the array is the target string, and the other axis is the typo string. Each index holds the calculated cost for that spot of the typo string compared to the target string, which is looked up instead of being re-calculated.
6. Structures used throughout these algorithms are declared in main

```
Algorithm: Distance
Input: 2 characters
Output: Distance between the two characters
    Compute distance based on row/hand/finger placement of characters
    Return distance

Algorithm: Insert
Input: Target string, typo string, index of target, index of typo
Output: Cost of insert operation
    Conditional statements around all possible cases of insertion typos
    Determine cost of insertion
    Return cost

Algorithm: Delete
Input: Target string, typo string, index of target, index of typo
Output: Cost of delete operation
    Conditional statements around all possible cases of deletion typos
    Determine cost of deletion
    Return cost

Algorithm: Substitute
Input: Target string, typo string, index of target, index of typo
Output: Cost of substitute operation
    Conditional statements around all possible cases of substitution typos
    Determine cost of substitution
    Return cost

Algorithm: Transpose
Input: Target string, typo string, index of target, index of typo
Output: Cost of transpose operation
    Conditional statements around all possible cases of transposition typos
    Determine cost of transposition
    Return cost

Algorithm: Cost
Input: Target string, typo string, index of target, index of typo
Output: Cost of minimum value between the four types of typos
    Return  min(Insert algo, min(Delete algo, min(Substitute algo, Transpose algo))

Algorithm: Typo
Input: Target string, typo string, index of target, index of typo
Output: 2D array (called DS) of min costs between the two strings
    If a memoized value is found in DS[i][j]
        Return DS[i][j]
    Else If the characters being compared are equal
        Increment both indices
        Recursively call Typo() on the next characters for both strings
```

```
                Return DS[i + 1][j + 1]
        Else If Target string has terminated but Typo string has not
                Recursively increment Typo string’s index and call Insert()
                Return DS[i][j]
        Else If Typo string has terminated but Target string has not
                Recursively increment Target string’s index and call Delete()
                Return DS[i][j]
        Else If both Target string and Typo string have terminated
                Return DS[i][j] = 0
        Else
                Obtain the cost of the current position’s typo using cost(), store in costVal
                Store the minimum output of the following three recursive calls:
                        Typo(target, typostring, i + 1, j)
                        Typo(target, typostring, i, j + 1)
                        Typo(target, typostring, i + 1, j + 1)
                DS[i][j] = The sum of costVal & the minimum of the three recursive calls
                Return DS[i][j]
```

Algorithm: Main

```
Input: # of operations, target string, typo string from text file
Output: Text file with type of typo and its respective indices
        Declare 2D array (DS) and initialize to 0
        Open file, store the number of problem cases
        Perform loop over problem cases:
                Stream strings into “Target” and “Typostring”
                Typo(target, typo, 0, 0)
                Output data to text file
                Clear data structure of values
        Close file
```

7. Worst case time complexity of the implemented memoized algorithm is O(n*m), where ‘n’ is the length of the target string and ‘m’ is the length of the typo string. This is because for the worst case, we have to compute every single value, and there is no instance of a value being looked up in the 2D array. All helper functions used to determine the cost and type of typo can be considered to be constant time. This is because they are passed two constant values, and a constant mathematical operation takes place between them.
8.
```
for i = 0 to length of target
        for j = 0 to length of typo
                check what kind of typo has occurred and chose min value for cost
                store result in a structure, 2D array
                if an insertion or deletion has occurred
                        update indices accordingly
        end

end

//moving down the diagonal of the array will give us the final result
```

9. For this problem, no. The iterative and memoized algorithms both require a 2D array which will be filled with values.

10. One advantage of the iterative algorithm is that it does not rely on any recursive calls, therefore we do not have to worry about a run-time stack to compute a value. One disadvantage is that there can be values computed more than once.