CSC 3430 Algorithm Design and Analysis
Homework 6 “Final Exam” take-home Lab/Project
Minimum Edit Distance between two strings – Dynamic Programming
Due: Wednesday, March 15, 2017 10:30am (final course deadline) – print + Zip

Design and implement a C/C++ program that uses a Dynamic Programming solution to compute the “minimum editing distance” between two strings and shows the operations necessary to transform the first string into the second string.

A problem description, analysis and an initial sample solution are shown in

- Kleinberg Textbook: Section 6.6 “Sequence Alignment”
- 06DynamicProgrammingII-Wayne.pptx powerpoint slides (available from Canvas)
- http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/
- http://algorithmist.com/index.php/Edit_Distance
- . . . several other web locations

The http:// sample solutions contain two functions to compute the minimum editing distance, a dynamic programming solution and a recursive solution that repeatedly re-computes partial solutions to smaller problems. You should also implement the recursive solution, and when you experiment executing the code, you’ll notice how much more efficient the dynamic programming solution is compared to the other solution.

Your primary task for this assignment is to make changes and additions to the dynamic programming solution to determine not only the “minimum editing distance” between two strings, but also to determine the set of “single-character editing operations” that are required to transform the value of the first string into the value of the second string.

If you visually line the two strings up on consecutive lines, there are three possible editing operations that might be used:

1. A character in the first string might be replaced by a different character in the second string (‘r’: a replacement);
2. A character that is present at a position in the first string might not be present in the second string (‘d’: a deletion);
3. A character that does not appear at a particular position in the first string may need to be inserted to complete the transformation into the second string (‘i’: an insertion).

As an example, to transform “Sundays” into “Saturday” minimally requires 4 editing operations:

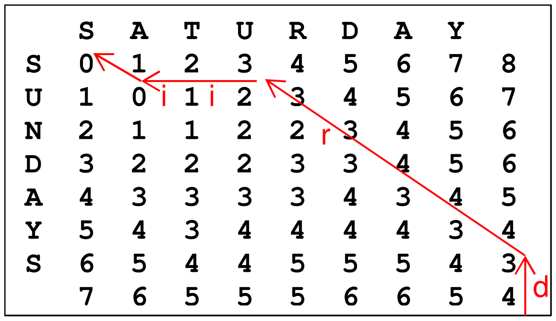
```
S  undays
 ii r   d              ← symbols for the editing operations: r, d, i
Saturday
```

e.g., insert the letters ‘a’ and ‘t’ between the ‘S’ and the ‘u’, replace the ‘n’ with the ‘r’, and delete the last ‘s’ character. The other letters that are “lined up” beneath each other appear in both strings in those positions. Note that other sequences of editing operations are possible that would transform the first string into the second string (for example, delete all the characters in the first string and insert all the characters in the second string); however, the dynamic programming solution determines a solution requiring the fewest number of editing operations.

The Dynamic Programming solution iterates row by row through the two string values, filling in the cache Memoization table for all of the smaller subproblems. At the end, the value M[m][n] contains the solution: a count of the minimum number of editing operations required for the transformation. However, your program must also determine and output the specific editing operations that are required, as shown in the example above.

To complete this lab, you must carefully study the dynamic programming code solution to understand how the cache "memoization" table that stores solutions to smaller sub-problems is constructed and utilized. For the "Sundays" : "Saturday" example above, this table is:

	S	A	T	U	R	D	A	Y	
S	0	1	2	3	4	5	6	7	8
U	1	0	1	2	3	4	5	6	7
N	2	1	1	2	2	3	4	5	6
D	3	2	2	2	3	3	4	5	6
A	4	3	3	3	3	4	3	4	5
Y	5	4	3	4	4	4	4	3	4
S	6	5	4	4	5	5	5	4	3
	7	6	5	5	5	6	6	5	4



	S	A	T	U	R	D	A	Y	
S	0	1	2	3	4	5	6	7	8
U	1	0	1	2	3	4	5	6	7
N	2	1	1	2	2	3	4	5	6
D	3	2	2	2	3	3	4	5	6
A	4	3	3	3	3	4	3	4	5
Y	5	4	3	4	4	4	4	3	4
S	6	5	4	4	5	5	5	4	3
	7	6	5	5	5	6	6	5	4

You must examine the table values to determine where decisions were made as the "editing distances" were calculated, and implement an algorithm to systematically map those decisions on to the corresponding i)nsert, d)elete, and r)eplace operations. Then your program must produce output similar to that shown in the example, the 2 strings printed on different lines with appropriate internal character spacing in the strings and the corresponding i, d, and r editing operations shown at the proper locations.

Some details and requirements:

- Design your program to prompt the user to input the two string values that are to be compared.

- You may assume that the maximum lengths of the strings to be compared is 50 characters. This allows your program to create a fixed size 2-dimensional array as the Dynamic Programming memoization "cache" table. e.g.
  ```
  int  M [50] [50];
  ```
  which allows the C++ compiler to properly access array elements M [i] [j].
  Shorter strings can use this same table by just utilizing the top few rows and columns and ignoring the rest of the larger table.

- Your program must print out the relevant portion of the cached M table so that you can study it and to document the correct operation of your solution.

- Your solution should also allocate two additional character string variables (or cstring array buffers) to hold the copies of the original 2 strings, but with sufficient extra space to allow the letters in the strings to be positioned with the necessary spacing to show where inserts or deletes have occurred (e.g., these buffers might need to have 100 length).

- The executable version of the instructor solution is available on Canvas. It runs from the command line, and can optionally print out the table by specifying the keyword 'table' on the commandline:
  ```
     EditDistance
  or EditDistance  table
  ```

Turn in on due date:

- Submit a ZIP of your "cleaned" Visual Studio solution directory in Canvas.
- Turn in a printed copy of your source code listings for this project.