Christian Armatas

CS 325 – Analysis of Algorithms

Implementation 2 – Report

2/23/2017

**Pseudo-Code:**

**minDistance**( str1, str2 ):

# Check base cases

if length( str1 ) == 0:

return length( str2 )\*2

if length( str2 ) == 0:

return length( str1 )\*2

len1 = length( str1 ) + 1

len2 = length( str2 ) + 1

# Create distanceMatrix[ len1 ][ len2 ]

for i = 0 … len1:

for j = 0 … len2:

delete = distanceMatrix[ i-1 ][ j ] + 1

insert = distanceMatrix[ i ][ j-1 ] + 1

substitute = distanceMatrix[ i-1 ][ j-1 ]

if str1[ i-1 ] != str2[ j-1 ]:

substitute += 2

distanceMatrix[ i ][ j ] = min { delete, insert, substitute }

**call** **traceback**

**return** distanceMatrix[ len1 ][ len2 ]

**traceback** ( first, second, distanceMatrix ):

i = length( first ); j = length( second )

while ( i = length( first ) > 0 ):

# Check if we are traversing left, down, or diagonal

if (DIAGONAL) <= (LEFT) & (DOWN):

i -= 1; j -= 1

str1 = first[i] + str1

str2 = second[j] + str2

# Traceback continued…

elif (DOWN) <= (LEFT) & (DIAGONAL):

i -= 1

str1 = first[i] + str1

str2 = "-" + str2

else:

j -= 1

str1 = "-" + str1

str2 = second[j]+ str2

# If we reach end of a string, add “-“ to front of completed string and

# add the remaining characters to other string

if (j <= 0):

while (i != 0):

i -= 1

str1 = first[i] + str1

str2 = "-" + str2

**break**

elif (i <= 0):

while (j != 0):

j -= 1

str1 = "-" + str1

str2 = second[j] + str2

**break**

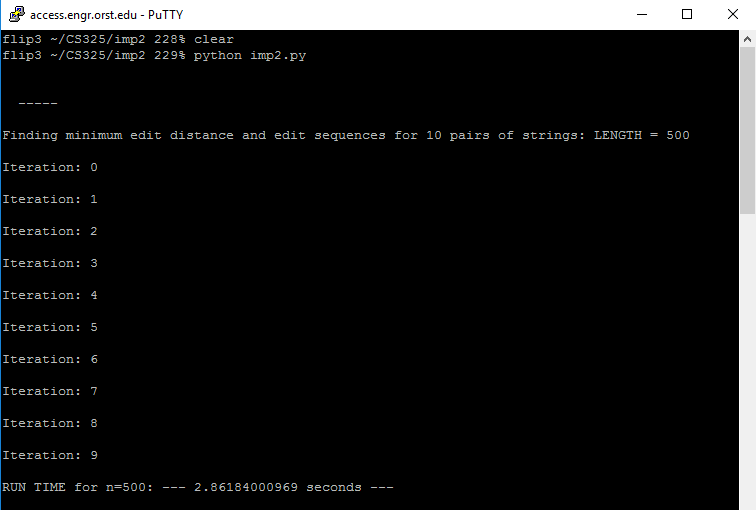
**return** str1 + “,” + str2

**Asymptotic Analysis of Run Time:**

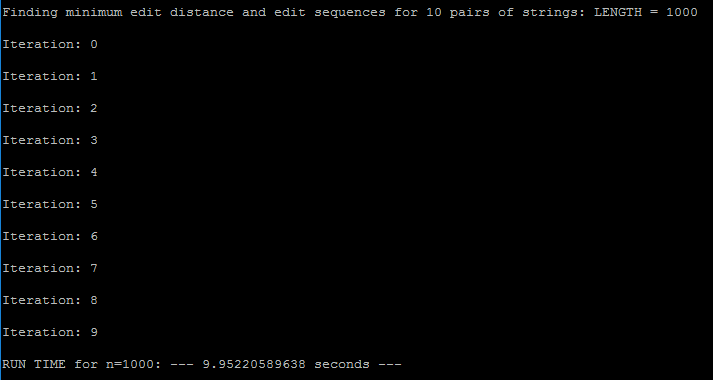
If N is the length of the longer of the two strings, and D is the minimum edit distance between the two strings, the most optimized runtime for finding the minimum distance and reconstructing the edit sequences of the two strings would be O(n + d2). Worst case, we must traverse through the entire length of the longest string to compare against the other string and to find the minimum distance between the two; this gives us the “n”. We then must add “d2,” which represents the worst-case runtime for reconstructing the edit sequences. At its worst, string1 and string2 are nothing alike and we must traverse the entire minimum edit distance (d) for each value in the minimum edit distance path.

**Recording and Plotting of Run Time:**

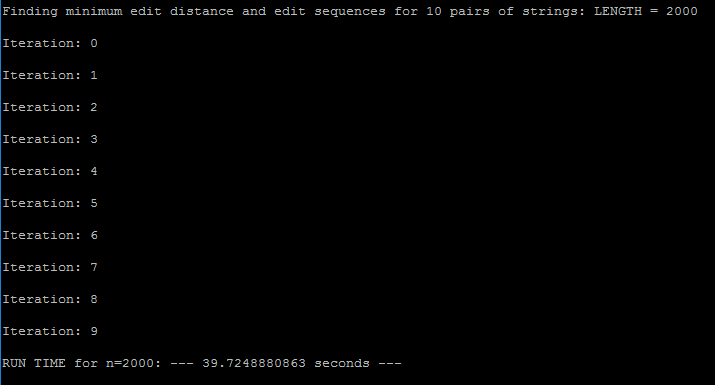
**N = 500**



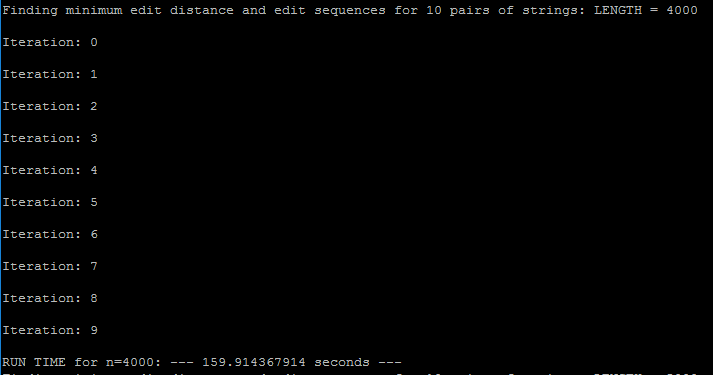
**N = 1000**



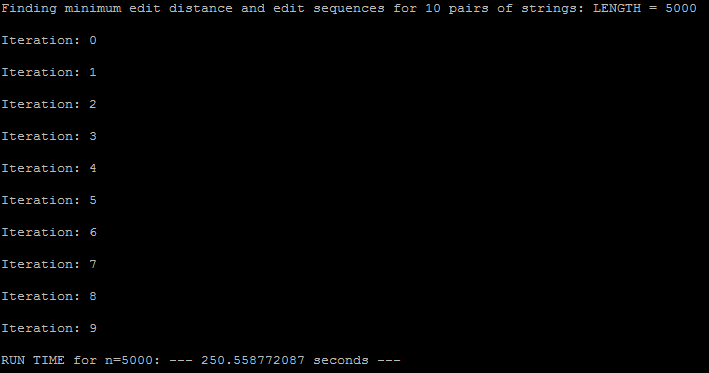
**N = 2000**



**N = 4000**



**N = 5000**



**Plotting Run Time vs. Number of Inputs:**

**Interpretation and Discussion:**

I initially programmed “imp2.py” using the naïve approach to test the differences between it and the more optimized approach. When I did this, I found (as expected) that the more I increased M and N (lengths of string1 and string2), the runtime increased exponentially. Then I began to program the more optimized approach, which I found to be significantly faster. Using the run time data above, I found that run time would still increase dramatically the greater the length of the strings. Below is a table demonstrating the supposed runtime using the optimized runtime formula, **O(n + d2)**:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **D (on average)** | **Calculated Run Time** | **Actual Run Time** |
| 500 | 360 | 1.30100 | 2.86184 |
| 1000 | 702 | 4.93804 | 9.952205 |
| 2000 | 1388 | 19.28544 | 39.72488 |
| 4000 | 2794 | 78.10436 | 159.91436 |
| 5000 | 3464 | 120.04296 | 250.55877 |

As you can see, I didn’t quite hit the most optimized runtime. In fact, for all values of n (n = 500, 1000, 2000, 4000, and 5000), it seems that my run times were just over double the calculated runtime. This is most likely because I would open and close the “imp2input.txt” and “imp2output.txt” files on every iteration to ensure the files would not become corrupted during processing. Though there were some additional seconds added to the each of the runtimes, all my actual runtimes still follow a consistent growth ratio to that of the optimized runtime, just as a multiple of 2.