Brandon Dring

CS 325 HW #2 Report

How to run the program:

To run the program you can simply just type

```
python Align.py
```

This is assuming that imp2input.txt and imp2cost.txt are in the same directory that Align.py is in. The program reads ALL the lines in the imp2input.txt and writes to imp2output.txt

Pseudocode:

The first step was creating my cost book, So I created a dictionary with all variations from the cost file. So there was a dictionary looked like
E.G:

```
costBook = {“AT”: 1, “TA”: 1… }
```

Then you read the input file and grab a sequence that ended at a new line, then split it by a comma to separate the two sides.

Now create an empty 2D table that has length of each respective side + 1
E.G:

```
array = [[0] * (len(Split_Line[0]) + 1) for i in range(len(Split_Line[1]) + 1)]
```

Fill in all the base cases of each 0th element with the cost of the previous cell + the cost of deleting/inserting the character at the current point
E.G:

```
for i in range(1, len(xLine) + 1):
    array[i][0] = array[i - 1][0] + int(costBook['-' + xLine[i - 1]])
for i in range(1, len(yLine) + 1):
    array[0][i] = array[0][i - 1] + int(costBook[yLine[i - 1] + '-'])
```

Now you need to go and fill in the table so you need a double for loop. And when filling in the table you want to choose the surrounding cells lowest possible values + the cost of doing whatever operation whether it be insert, delete, or swap. Then use the lowest value found for the current cell.
E.G:

```
for i in range(1, len(xLine) + 1):
    for j in range(1, len(yLine) + 1):
    array[i][j] = min(array[i - 1][j] + diff('-', xLine[i - 1], costBook),
        array[i][j - 1] +
        diff(yLine[j - 1], '-', costBook),
        array[i - 1][j - 1] + diff(yLine[j - 1], xLine[i - 1], costBook))
```

Now that the table is filled in you need to back trace through it, taking the lowest possible value at each step.
E.G:

```
if (array[i][j] == array[i][j - 1] + int(costBook['-' + yLine[j - 1]])):
    StringA += '-'
    StringB += yLine[j - 1]
    array[i][j] = 0
    j -= 1    ..etc
```

If the lowest cost is to the left, then there was an insertion or deletion on the First sequence with the letter on the Second Sequence.
If the lowest cost is directly above, then the second sequence had an insertion or deletion with the first.
If the lowest cost is diagonally then there was a match or swap between the two characters

At each step you add the respective case to your strings, then decrementing depending on the cases. Flip the strings and return the final cost and Viola! Done.

So:

1. Build costBook
2. Zero Fill Array
3. Build Base cases at 0th position
4. Fill table with minimum values from previous cells + current cost
5. Trace back through the table and create strings

buildCostbook()

For i.. N
    For i .. m
        Array[i][j] = 0

For i.. N + 1

```
        Array [i][0] = previous cell + insertions/deletions cost on x axis
For i .. M +1
        Array [0][i] = previous cell + insertions/deletions cost on y axis

For i.. N
        For j.. M
                Array[i][j] = minCost of surrounding cells + cost of altering current cell

//Backtrace

While (i != 0 or j != 0)
        Match current cell cost with surrounding cells cost + cost of altering current cell
        Depending on case, add to current strings with either a '-' or letter
```

Asymptotic Analysis of RunTime:

So this algorithm should run in an O(xy) runtime due to having to build a 2D table with a double for loop. Plus O(x+y) when back tracing through the table but that is dominated by the building of the table in general.

Reporting and plotting the runtime:

To measure the runtime, I created a timer that starts right after it has read in the line to be align. Then it stops after it has built the table and done the backtrace and built the strings and writing them to its respective file.
I tested it on the school server just to make sure that it works. But overall the testing and building was done on my 2015 Macbook Pro with 8gb of RAM and a 2.7Ghz i5. To find the average run time I built a small shell script that ran the algorithm 5 times on the different input sizes(500,1000, etc) then ran another script inside that to calculate the averages of the 5 runs and saved that to another file. So each input size was ran 5 times and averaged. I should note that when I created the various input sizes, let's say for 500 I just made one sequence length 250 and the other 250 for simplicity sake. I figured various lengths got tested with the given input file from canvas.

Runtimes as functional input:

f(500) = 0.14487719535827637
f(1000) = 0.6694761753082276
f(2000) = 3.0277688026428224

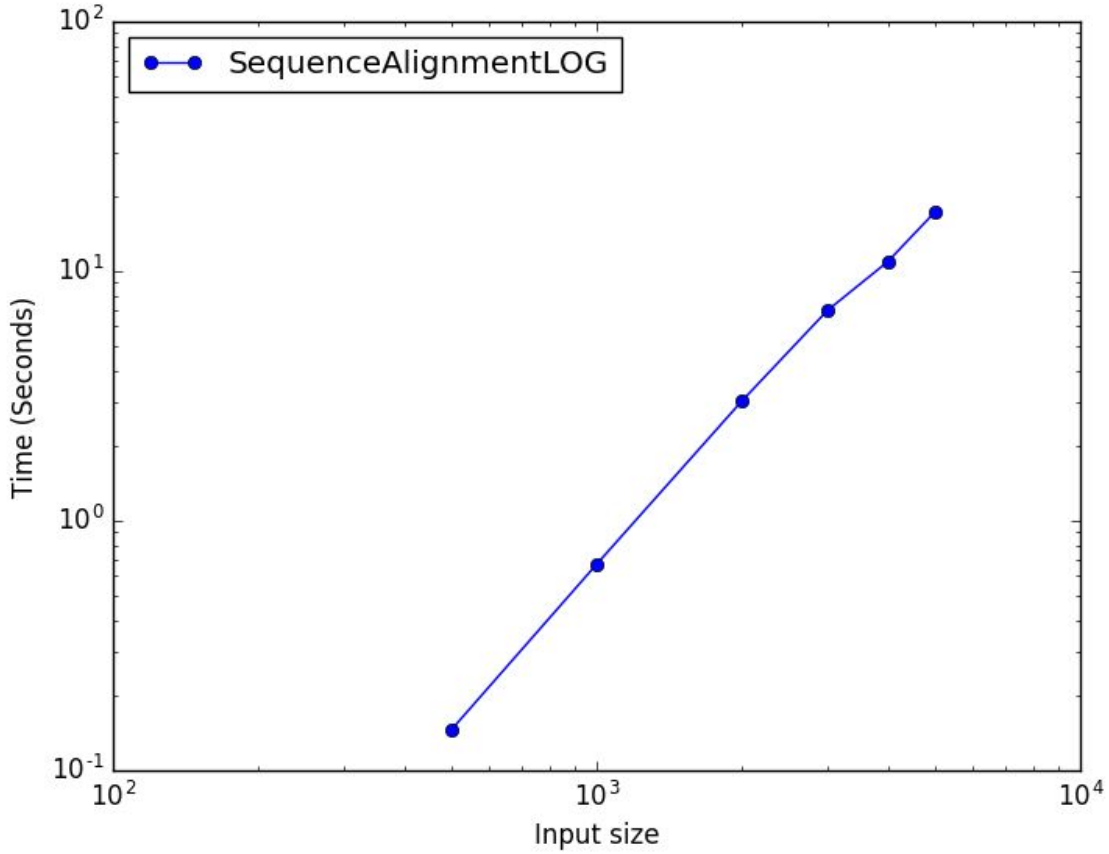
f(3000) = 6.957123565673828
f(4000) = 10.9172354221344
f(5000) = 17.20967264175415

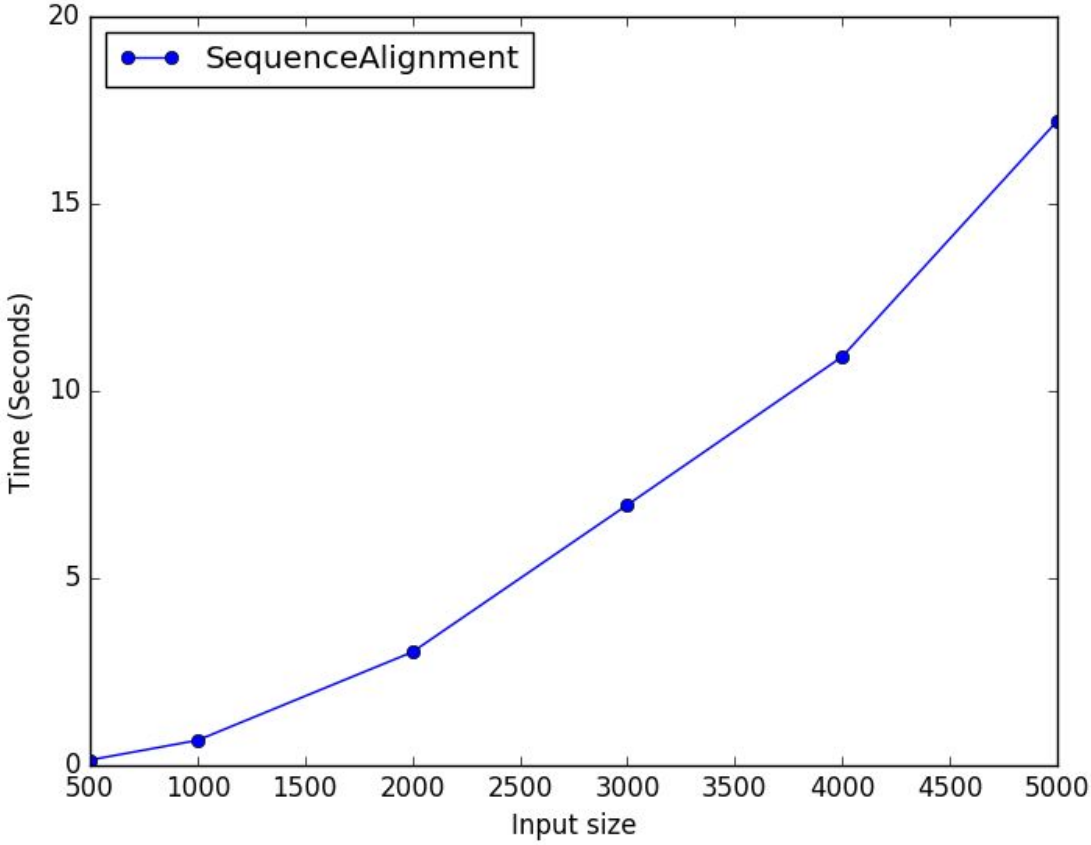
The slope for the graphs was: 0.00396. So for every additional input size, it made it take all of 0.004 seconds.. Roughly , *I have also included some computer generated graphs in the folder.* This roughly accounts for the time it takes to run the program in O(nm) time, plus the un accounted for O(n+m) runtime of the backtrace.

Interpretation and discussion:

For my own generated input, the scale should be about O(n^2) since each sequence has equal length. But looking at 2 points say input size of 2000 and input size of 4000. The input size doubles, but the average time it takes is roughly squared, plus change. Which seems right on point, that has roughly an O(n^2) run time, plus whatever the difference is for the backtrace of O(n+m)

Graphs:


SequenceAlignmentLOG
Time (Seconds)
Input size
10^{2}
10^{1}
10^{0}
10^{-1}
10^{2}
10^{3}
10^{4}


SequenceAlignment
Time (Seconds)
Input size
0
5
10
15
20
500
1000
1500
2000
2500
3000
3500
4000
4500
5000