CS 325 - Analysis of Algorithms
February 23, 2017
Group members: Eric Stevens, Jong Park, Zhenggan Zheng

Programming Assignment 2 - DNA Edit Distance

Pseudo-code

```
1. Take the input strings and turn it into string1, string2
2. Prepend an empty string “_” to list1 and list2.
    a. New length of string1 and string2 will be one larger than previous. Denote this new length as string1Length and string2Length.
3. Take the cost input strings and create cost matrix (penalty chart)
4. Create a cost chart of size string1Length by string2Length
5. Initialize the first row and column of the cost chart by looping through both strings populating them with the penalty of appending the characters up to string1[i] to a null string for string1 and the penalty of turning characters up to string2[j] to a null string for string2.
6. Create direction chart
    a. If moved in j direction, it moved right.
    b. If moved in i direction, it moved down.
    c. If moved in both i and j direction, it moved diagonally
7. Loop through the rest of the indexes in the cost chart and direction chart simultaneously
8. Take the minCost(left, up, diag) + (value of previous index).
    a. Append that value to the cost chart at current index.
    b. Take the direction that minCost(left, up, diag) took and append that to the direction chart.
9. Loop through the direction chart to recreate the aligned strings.
    a. Call them aligned1 and aligned2, they will initially be empty strings.
    b. Also create 2 variables call S1 and S2 which will be pointers to an element in string1 and string2
    c. If it came from up, append ‘-’ to aligned1, and string2[S2], S2++
    d. If it came from left, append ‘-’ to aligned2, and string1[S1], S1++
    e. If it came from diagonal, append string1[S1] to aligned1 and string2[S2] to align2, S1++, S2++
10. Take the minimum edit distance of the two strings based on direction and penalty chart.
```

Asymptotic Analysis of run time

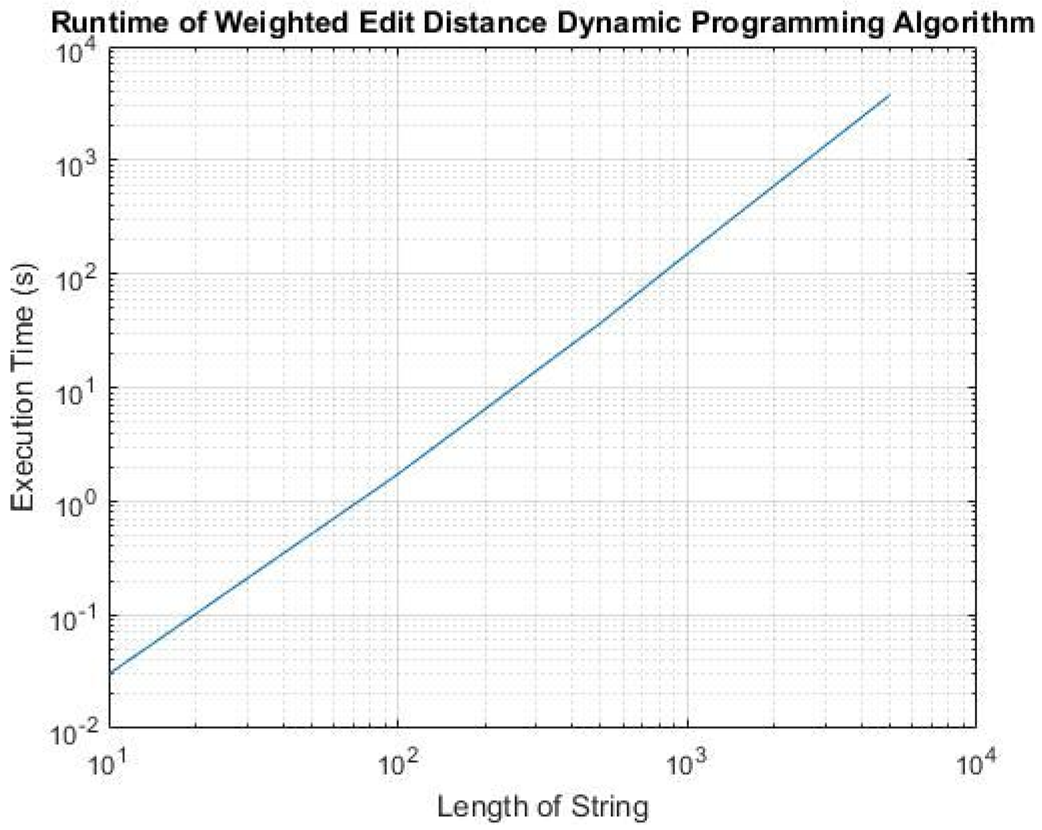
m = length of string1
n = length of string2

Since loop is :

```
For i to m+1:
  For j to n+1:
```

Number of recursive calls is (m+1)*(n+1) = O(mn). The runtime required to trace back and figure out the aligned values will be O(m+n) because this is the worst case scenario for when the backtrace must traverse the length of m and the length of n. The final runtime becomes O(mn)+O(m+n). For very large values of m and n, this becomes O(mn). When m and n are the same length, the runtime is O(n^2).

Plot of the Runtime



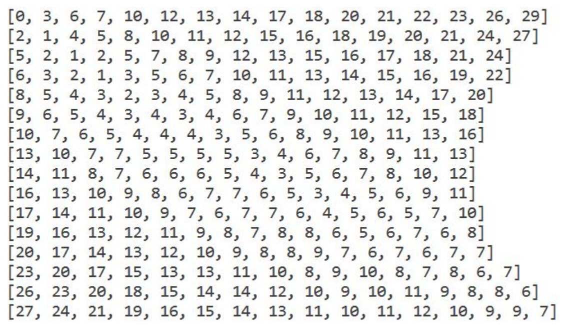
Below is a table of our average of 10 runtimes for each string input length:

String Length	Runtime (seconds)
10	.03
100	1.73
500	36.84
1000	150.34
2000	600.05
4000	2393.3
5000	3731.32

The running time in our experiment was measured by generating two random string of letters A, T, G, C of lengths 500, 1000, 2000, 4000, and 5000. Then the two strings are passed into the algorithm function and is timed. This is repeated 10 times for each string length and then is averaged. The data is then passed into a Matlab function and plotted on a log-log plot. The slope of the log-log plot according to the Matlab function polyfit is 1.9. This means that the runtime is O(n^1.9) which is very close to O(n^2) which is the expected runtime.

Interpretation and discussion

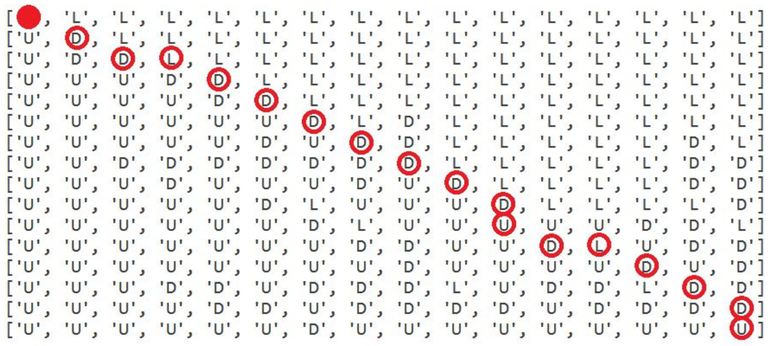
The runtime plot is based on comparing two strings of the same size. In reality this will not always be the situation. We should be able to compare strings of different sizes and obtain edit This results in a run time that is more complicated than simply O(n^2). Let "n" be the length of a string and "m" be the length of a second string, the run time of the algorithm will be O(mn). This is not only the big O() of the algorithm but it is also the expected runtime. This is because for any instance of the algorithm needs to calculate all values within an mxn matrix in order to come to a final solution. This can be seen in the image below.



```
[0, 3, 6, 7, 10, 12, 13, 14, 17, 18, 20, 21, 22, 23, 26, 29]
[2, 1, 4, 5, 8, 10, 11, 12, 15, 16, 18, 19, 20, 21, 24, 27]
[5, 2, 1, 2, 5, 7, 8, 9, 12, 13, 15, 16, 17, 18, 21, 24]
[6, 3, 2, 1, 3, 5, 6, 7, 10, 11, 13, 14, 15, 16, 19, 22]
[8, 5, 4, 3, 2, 3, 4, 5, 8, 9, 11, 12, 13, 14, 17, 20]
[9, 6, 5, 4, 3, 4, 3, 4, 6, 7, 9, 10, 11, 12, 15, 18]
[10, 7, 6, 5, 4, 4, 4, 3, 5, 6, 8, 9, 10, 11, 13, 16]
[13, 10, 7, 7, 5, 5, 5, 5, 3, 4, 6, 7, 8, 9, 11, 13]
[14, 11, 8, 7, 6, 6, 6, 5, 4, 3, 5, 6, 7, 8, 10, 12]
[16, 13, 10, 9, 8, 6, 7, 7, 6, 5, 3, 4, 5, 6, 9, 11]
[17, 14, 11, 10, 9, 7, 6, 7, 7, 6, 4, 5, 6, 5, 7, 10]
[19, 16, 13, 12, 11, 9, 8, 7, 8, 8, 6, 5, 6, 7, 6, 8]
[20, 17, 14, 13, 12, 10, 9, 8, 8, 9, 7, 6, 7, 6, 7, 7]
[23, 20, 17, 15, 13, 13, 11, 10, 8, 9, 10, 8, 7, 8, 6, 7]
[26, 23, 20, 18, 15, 14, 14, 12, 10, 9, 10, 11, 9, 8, 8, 6]
[27, 24, 21, 19, 16, 15, 14, 13, 11, 10, 11, 12, 10, 9, 9, 7]
```

Cost calculations are performed for every possibility.

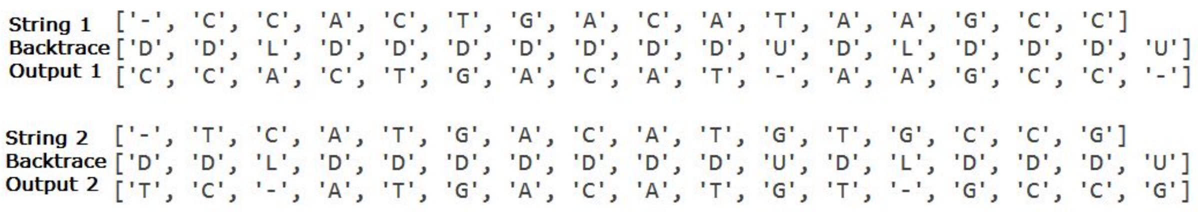
The traceback portion of the algorithm can be ignored for calculating the overall runtime because its runtime is O(n+m). This is because the highest number of operations a trace back could encounter would be a upward movements of the same number as there are letters in string two and a number of left moves equal to that of the length of string one. The backtrace process can be seen below.



```
[●, 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'D', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'D', 'D', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'U', 'U', 'D', 'D', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'U', 'U', 'U', 'D', 'D', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'U', 'U', 'U', 'U', 'U', 'D', 'L', 'D', 'L', 'L', 'L', 'L', 'L', 'L', 'L']
['U', 'U', 'U', 'U', 'U', 'D', 'U', 'D', 'D', 'L', 'L', 'L', 'L', 'L', 'D', 'L']
['U', 'U', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'L', 'L', 'L', 'L', 'L', 'D', 'D']
['U', 'U', 'U', 'D', 'U', 'U', 'U', 'D', 'U', 'D', 'L', 'L', 'L', 'L', 'D', 'D']
['U', 'U', 'U', 'U', 'U', 'D', 'L', 'U', 'U', 'U', 'D', 'L', 'L', 'L', 'L', 'D']
['U', 'U', 'U', 'U', 'U', 'U', 'D', 'L', 'U', 'U', 'U', 'U', 'U', 'D', 'D', 'L']
['U', 'U', 'U', 'U', 'U', 'U', 'U', 'D', 'D', 'U', 'U', 'D', 'L', 'U', 'D', 'D']
['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'D', 'U', 'U', 'U', 'U', 'D', 'U', 'D']
['U', 'U', 'U', 'D', 'D', 'U', 'D', 'D', 'D', 'L', 'U', 'D', 'D', 'L', 'D', 'D']
['U', 'U', 'U', 'U', 'D', 'D', 'U', 'D', 'D', 'D', 'D', 'U', 'D', 'D', 'D', 'D']
['U', 'U', 'U', 'U', 'U', 'U', 'D', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U']
```

Traceback routine takes O(m+n), starts from bottom right.

The regeneration of the output strings and the calculation of the edit distance also has runtime of O(n+m). More specifically, it matches the value of the initial trace back route calculation. The method by which the output string is generated is by using the backtrace result to determine where inserts, deletions, and alignments should be performed and lookup tables can be used to get the scores of each. Relative to string one, A downward movement would result in an insertion, the diagonal movement is an alignment, and a rightward movement represents a deletion. The result of this process can be seen below.



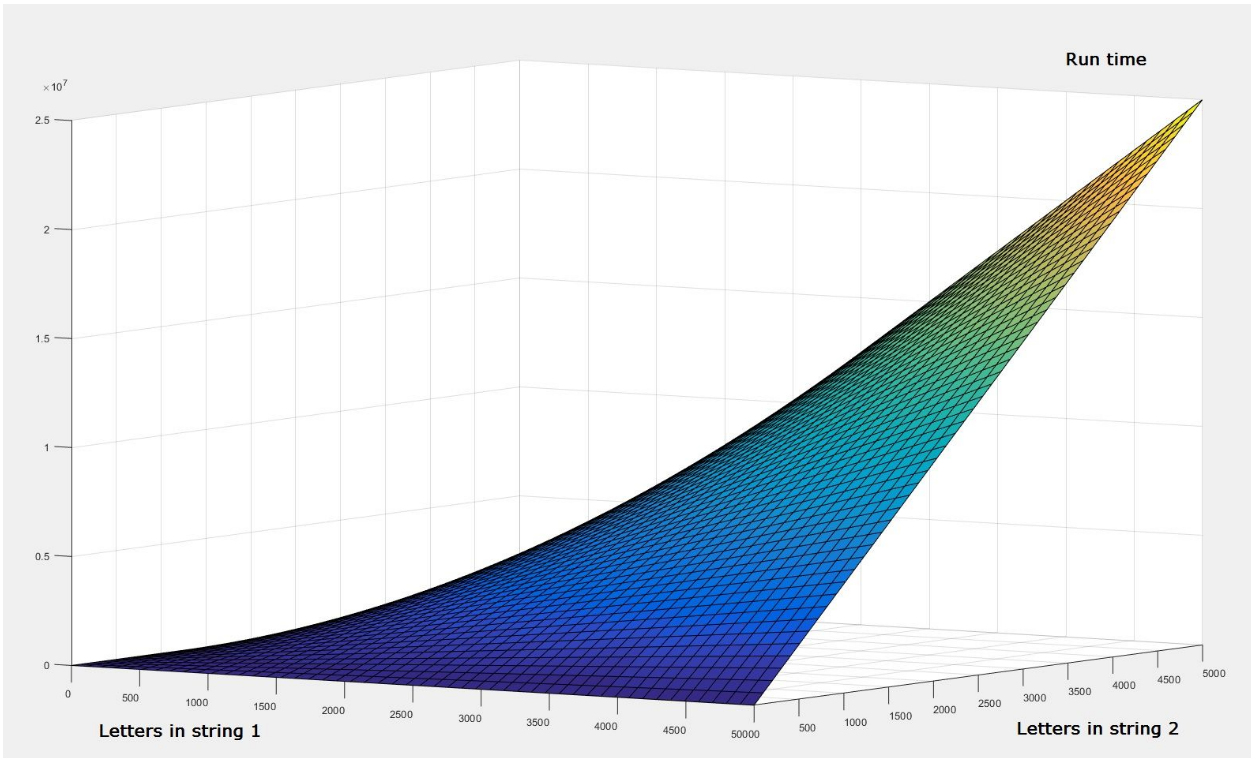
```
String 1  ['-', 'C', 'C', 'A', 'C', 'T', 'G', 'A', 'C', 'A', 'T', 'A', 'A', 'G', 'C', 'C']
Backtrace ['D', 'D', 'L', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'U', 'D', 'L', 'D', 'D', 'D', 'U']
Output 1  ['C', 'C', 'A', 'C', 'T', 'G', 'A', 'C', 'A', 'T', '-', 'A', 'A', 'G', 'C', 'C', '-']

String 2  ['-', 'T', 'C', 'A', 'T', 'G', 'A', 'C', 'A', 'T', 'G', 'T', 'G', 'C', 'C', 'G']
Backtrace ['D', 'D', 'L', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'U', 'D', 'L', 'D', 'D', 'D', 'U']
Output 2  ['T', 'C', '-', 'A', 'T', 'G', 'A', 'C', 'A', 'T', 'G', 'T', '-', 'G', 'C', 'C', 'G']
```

Output string and edit distance calculation process.

The final result of this process is a run time of:

$$T(n,m) = O(nm) + 2O(n+m) + (n+m)(\text{lookup})$$
$$= O(nm)$$

The theoretical result of this runtime can be seen in the graph below.



Theoretical run time based on length of the strings.

Appendix

Below are some code that was not part of the main algorithm but was used for timing purposes:

Matlab code to graph runtime:

```
runTimes = textread('RunTimes.out');

%Calculate the slope of the line
slopeandIntercept = polyfit(log(runTimes(:,1)), log(runTimes(:,2)), 1); %Slope was 1.89

%Plot the runtime in log-log scale
loglog(runTimes(:,1),runTimes(:,2));
grid on;
title('Runtime of Weighted Edit Distance Dynamic Programming Algorithm');
xlabel('Length of String');
ylabel('Execution Time (s)');
```

Timing Code:

```
#!/usr/bin/env python3

import time as timer
import genout as io

def measureRunTime(fileInput,fileOutput):

	testTimes=[]

	for i in range(10):
		#testList = io.output(numInputs, outFile)

		start = timer.clock()
		io.output(fileInput,fileOutput)
		end = timer.clock()
		time = end-start
		testTimes.append(time)
		print("Test",fileInput, "done.\n")

	averageTime = float(sum(testTimes)/len(testTimes))
	return averageTime


# file io names.
fileInput = ["pair10.in","pair100.in","pair500.in", "pair1000.in", "pair2000.in",
"pair4000.in", "pair5000.in"]
fileOutput = ["pair10.out","pair100.out","pair500.out", "pair1000.out", "pair2000.out",
"pair4000.out", "pair5000.out"]
# Int is easier for graphs.
numPairs=[10, 100, 500, 1000, 2000, 4000, 5000]

#write to external file and plot in matlab
testFile = open('RunTimes.out', 'w+')
index = 0

for item in fileInput:
        stuff=open(fileInput[index], 'r')
        averageTime=measureRunTime(fileInput[index],fileOutput[index])
        testFile.write("%d %f\n" % (numPairs[index], averageTime))
        index=index+1
        print ("looping", index, "times, done.\n")
        stuff.close()
testFile.close()

print("Task All Finished, please see \"RunTimes.out\" for run time results.\n")
```

Random Pair Generator:

```
#!/usr/bin/env python3
import sys
import random

NUM_LINES = 10
KEY_LEN = int(sys.argv[2]) # argv 2: String Length
fileout = str(sys.argv[1]) # argv 1: Filename

def key_gen():
    keylist = [random.choice(['A','C','G','T']) for i in range(KEY_LEN)]
    return ("".join(keylist))

fout = open(fileout, 'wt')
for i in range(0, NUM_LINES):
        x = key_gen()
        y = key_gen()
        print (x,',',y,sep='', file=fout)
fout.close()
```