David Olson, Section 2

Project 4 Report

PseudoCode

\*see my attached code too if needed. I documented the algorithm and complexity pretty well there also. It makes more sense to look at my scoring function first

\*also note that I conceptually have the first input word at the top of our scoring table and the second input word is vertically placed to the left of that table.

Scoring Algorithm

Create two arrays that are the length of the first word, call them top and bottom array

Fill in top array with the score equal to 5\*(column number) [O(n)]

[O(n^2)]

For each character in the second word:

The first index of bottom array is 5\*(row number) [row increases by one each char]

For all other indices in bottom array(or for each character in the first word):

The score is defined by the algorithm in the slides, use top array as previous row

When finished, swap top and bottom array

Finally, return the last index of top array(the arrays swapped one last time) as the optimal score

This is a pretty simple algorithm. We essentially navigate through an n^2 table, and in each cell we grab some values from strings or arrays, make some comparisons, and insert into an array all in constant time. So, this is a O(n^2) algorithm. The space complexity is just O(n) though because we are keeping two arrays of size n, so 2n. We only need the current row and the one above because we only care that the final index is discovered.

Extraction Algorithm

Create two 2D arrays called score array and pointer array with the dimensions of the input words. Naturally, one keeps track of the optimal score and the other holds the back pointers.

The first row and first column are inserted with values equal to 5 \* (row/col number)

And the corresponding pointer cells are marked as either Left or Up [O(n)]

[O(n^2)]

For each character in the second word:

For all unfilled cells in the row corresponding to a character in the first word:

The score is defined by the algorithm in the slides

The corresponding pointer cell is inserted as Left, Up, or Diagonal

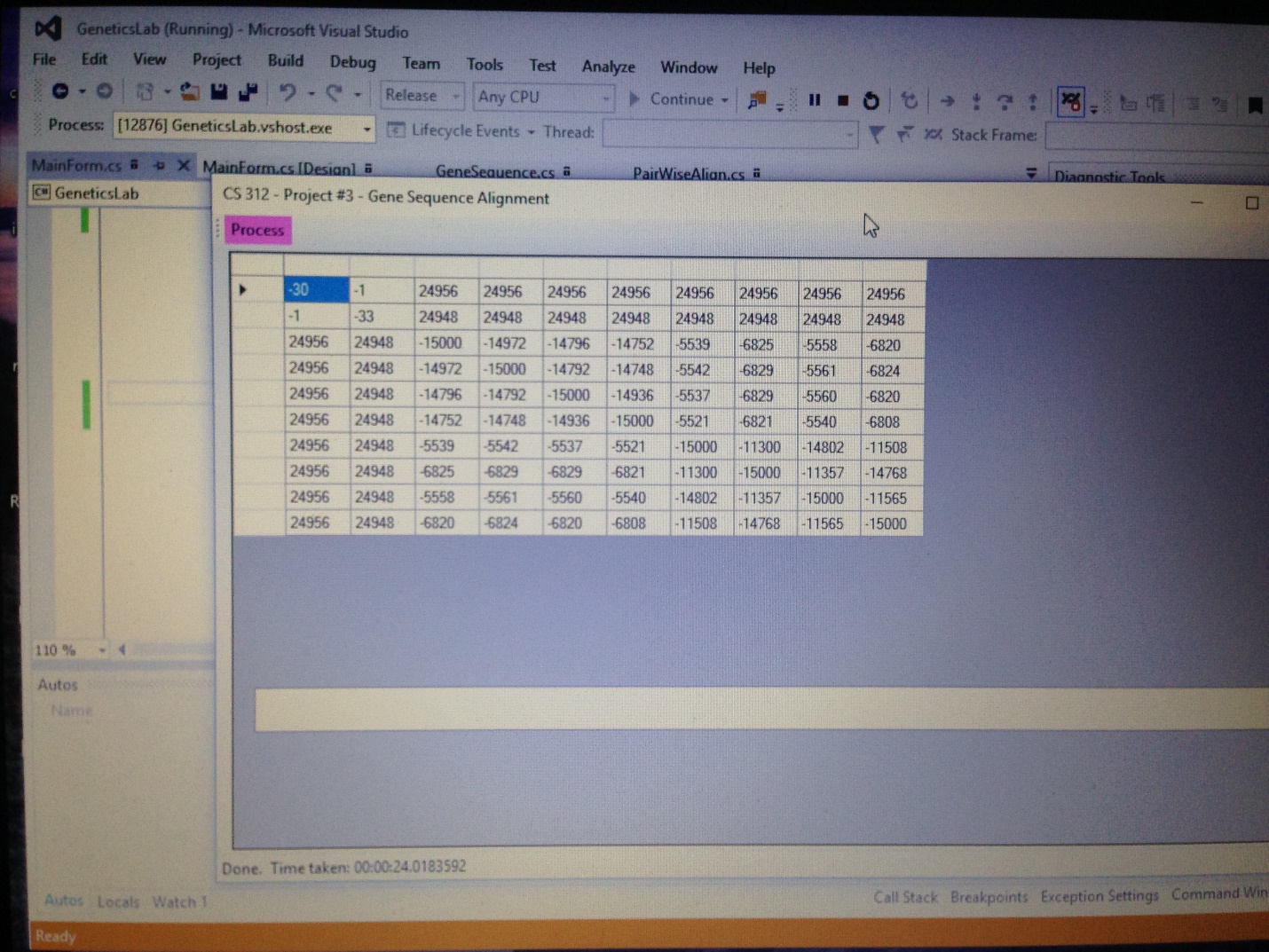
In this algorithm we have the same shell of nested for loops, which allow us to traverse the entire score and pointer array at the same time. We still only need constant time to find the min of three values. Depending on which value was the min, we put the correct pointer into the pointer array cell, also a constant time procedure. So we again have O(n^2) time. Our space complexity is also O(n^2) because we two 2D arrays in memory to remember the score and back pointers, and 2n^2 is in O(n^2) space

Retrieving the Alignment

Begin with two empty alignment strings. We navigate from the bottom right to the top left of our pointer array. For each cell, if the pointer was diagonal, we tack on the character of the corresponding index of each input string(the indices will be off by one from the row and column) **to the front** of the two alignment strings we are building. If the pointer was left, we tack on a hyphen to the second alignment string and tack on the corresponding character of the first input string to the first alignment string. If the pointer was up, we tack on a hyphen to the first alignment string and tack on the corresponding character of the second input string to the second alignment string. On a diagonal, both the row and column count are decremented, and on left and up, the column and row counts are decremented respectively.

The optimal alignment is these two strings displayed one above the other. Since the traversal never backtracks and stays within our pointer table, and reading strings and inserting onto the end of strings should be constant time, this part of the algorithm is also in O(n^2) time.

Score Results



In release mode, my 10x10 matrix filled in 24 seconds. Just the top half finished in about 11

Alignment Results

First Half:

gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgtt

-ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgtt

Second Half:

agatcttttcataatctaaactttataaaaacatccactccctgt-agtcta-

agtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtgtccat

My alignment was done by extracting the first 100 characters of each string then running the algorithm

Attached Source Code

The score algorithm is in the pairwise align file, and the extraction algorithm is in the mainForm file because it takes places in a click event