John Goza
EN.605.620.81.SP23 – Algorithms for Bioinformatics
Lab 3 – Dynamic LCS
Due 7 May 2023

Abstract

In Lab 3 we implemented and evaluated the bottom up dynamic programming version of the longest common subsequence (LCS) algorithm proposed in *Introduction to Algorithms* (Cormen). The implementation of this algorithm in Python proved to be fairly straightforward, but the analysis required careful consideration. I found that my implementation of the algorithm produced the same expected efficiencies as claimed in the book when loop iterations were counted, but on counting comparisons, the algorithm did not meet the efficiency theorized by the book. This lab gave me a deeper understanding of the subtleties involved with real world algorithmic analysis, and a better level of comfort with bottom up dynamic algorithms.

Data Structures and Design

The required input file for this assignment consisted of multiple lines of key value text pairs formatted as key = value, with every pair on a new line and no quotation marks surrounding the keys or values. To parse this and check for errors in the input, I created an ingestion module called "input_handler.py". This module checks for file existence, file validity (i.e. not a directory), and appropriate file permissions. It also checks to ensure that the file contained at least two lines of valid input. If the file does not contain two or more lines of valid input, the module throws an error, as comparison requires two items. File format checking is limited; as strings can be composed of any set of characters, I assumed that the only criterion for a valid line of input was an equals sign in-between two blocks of characters. This allowed me to include test input with special characters, numbers, and other "non-standard" elements, showcasing the robustness of the implementation. Input is parsed into a dictionary and then returned to the main module ("Lab3.py") for processing.

The main module contains the logic necessary to handle arguments provided by the caller, and to conduct the data in an appropriate flow. When the input is returned from the ingestion module, the main module uses the Python itertools package to produce an array containing all possible combination pairs of the input values. For each pair, the program builds an LCS table and then "walks" that table in order to determine the longest subsequence as well as the count of comparisons performed during table and path generation. The LCS module ("lcs_functions.py"), which handles those functions, is described in more detail later. Finally, the main module sends the results to the output module ("output_handler.py"), which intelligently formats the output and writes it to either the console, a file, or both depending on the calling user's preference.

In the LCS module there are three functions which are named in a self descriptive fashion: "init_tables", "generate_lcs_table", and "walk_lcs_path". The "generate_lcs_table" and "walk_lcs_path" functions are heavily based off of the pseudocode on pages 394-395 of *Introduction to Algorithms* (Cormen). Improvements were made based off of the suggestions found on page 396 of *Introduction to Algorithms* (Cormen). The largest improvement was found by eliminating the *b* table described in the pseudo-code and instead using the *c* table for "walking". This cut memory usage in half with no change in asymptotic computational efficiency.

As a general rule, no special data structures or design patterns were used in this project. The relative straightforward nature of the problem allowed for approachable, "simple" code, with two notable exceptions: 1) the itertools usage found in the main module, and 2) the output module. As the itertools package has significant documentation available we chose not to discuss it in this report. The

output module, however, is worth discussion. The output module must be instantiated as a class before it can be used, and the constructor takes optional arguments for detailed output, output method, and filename. This allows the output route to be set once, at instantiation time, and then used for the rest of the program life cycle without the need for extra parameters on all future calls. I'm quite proud of it to be honest.

Results

In *Introduction to Algorithms* (Cormen), the authors claim that the LCS-LENGTH function runs in $\Theta(m \cdot n)$ time, while the PRINT-LCS function runs in $O(m + n)$ time (where m and n are the lengths of the strings undergoing comparison). The required input (Table 1) was run through the program and produced the results seen in Table 2[1].

Sequence Name	Sequence Content	Length
S1	ACCGGTCGACTGCGCGGAAGCCGGCCGAA	29
S2	GTCGTTCGGAATGCCGTTGCTCTGTAAA	28
S3	ATTGCATTGCATGGGCGCGATGCATTTGGTTAATTCCTCG	40
S4	CTTGCTTAAATGTGCA	16

Table 1) Required input for assignment. Input has been formatted to fit publication.

M	N	Length M	Length N	$m \cdot n$	Loops (table)	Comparisons (table)	$m + n$	Comparisons (walk)	LCS
S1	S2	29	28	812	870	2298	57	46	GTCGTCGGAAGCCGGCCGAA
S1	S3	29	40	1160	1230	3279	69	81	ACCGGCGATGCGGAACCCG
S1	S4	29	16	464	510	1339	45	55	CTGCTAAGGCA
S2	S3	28	40	1120	1189	3132	68	70	GCTTCGGGCCGTGCTTGTAA
S2	S4	28	16	448	493	1273	44	46	GCTTAATGTGCA
S3	S4	40	16	640	697	1808	56	71	TTGCTTATGTGCA

Table 2) Results produced from required input. Columns corresponding to the predicted computational efficiencies ($m \cdot n$ for table construction, $m + n$ for path walking) are provided for convenience. Note the significant difference between table loops and table comparisons.

The results show that the algorithm was implemented correctly, as the loop iterations were within a constant factor of $m \cdot n$ (the expected value from Cormen) and the comparisons needed for walking the table to produce the longest string were within a constant factor of $m + n$ (the expected value from Cormen). In this project we were tasked with counting the individual comparisons as well, which are seen in the same table. Comparisons were consistently larger than iterations, but the pattern was not strong enough to draw a generalized conclusion from. To assist in this effort, I created test input with a variety of larger strings. This input contained nine strings of 50 characters, nine strings of 250 characters, nine strings of 500 characters, and six strings of 1000 characters. The strings were

1 While the program outputs the built LCS matrix for each pair of strings, these are not included in the report due to formatting constraints. Unmodified output can be verified by running the program locally.

assembled from genetic sequences, xml, and quasi-random log output files in order to represent a "diverse" dataset. All strings were in the same file and thus were compared to all other strings in the file. Results for this dataset were collected and averaged according to input string sizes. The aggregate results can be found in Table 3. As seen in the small scale results, the loop iterations matched very closely to the expected value of $m \cdot n$, but the number of comparisons was much greater. This trend

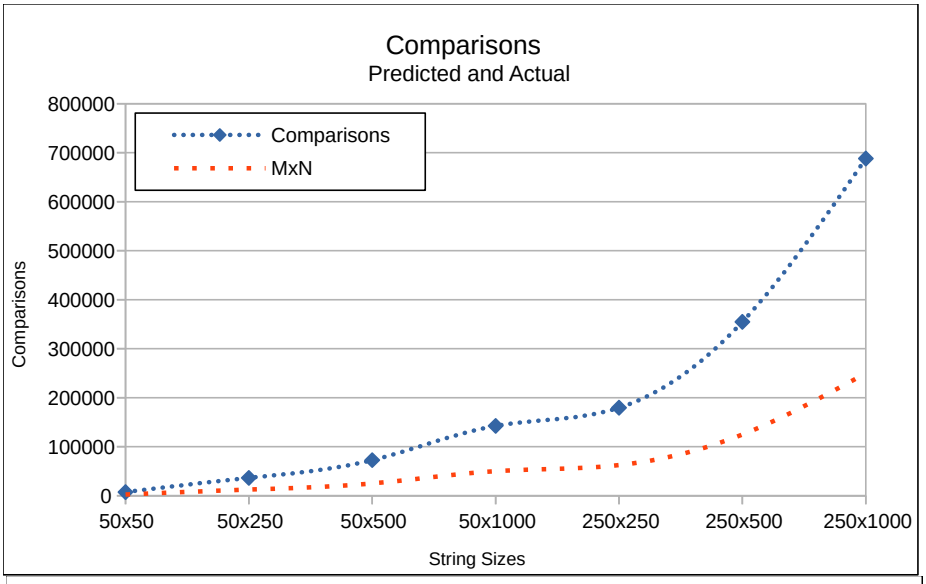
String Sizes	$m \cdot n$	Loops (Table)	Comparisons (Table)	$m + n$	Walk steps
50x50	2500	2601	7322	100	90
50x250	12500	12801	36366	300	308
50x500	25000	25551	72655	550	382
50x1000	50000	51051	142490	1050	231
250x250	62500	63001	179515	500	498
250x500	125000	125751	354757	750	832
250x1000	250000	251251	688046	1250	883

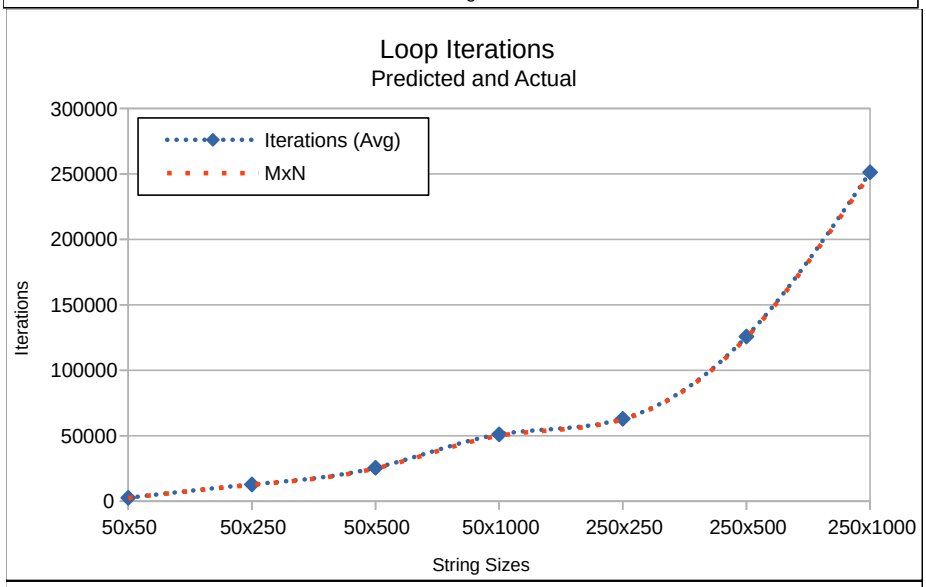
Table 3) Results of large-scale string comparison.

was consistent across all string input size combinations. In an attempt to understand the assymtotic difference between the two measurements, charts were produced of the aggregate data, as seen in Figure 1. While iterations matched expected efficiency at all levels, both comparisons and LCS generating path walking diverged from their expected outcomes to a significant degree. On the comparison chart, the trend suggests that the efficiency may be greater than the expected value of $m \cdot n$, but further research would be needed to confirm that. Path walking performed in a unpredictable manner, and no large-scale conclusion could be drawn from the data.

Reflections

While the performance of the implementation matched efficiency in iterations, comparisons were well above the expected norm. If I were to do this lab again, I would spend more time trying to optimize that portion of the program. The distinction between iterations and comparisons is an important one, especially when we consider this program in a bioinformatics context. LCS programs like this one are commonly used in genetic matching, annotation, and classification. These problems require efficiency, as genetic data can be very large and computationally expensive. This lab reinforced that concept for me, and forced me to better understand the complexity of algorithmic analysis.





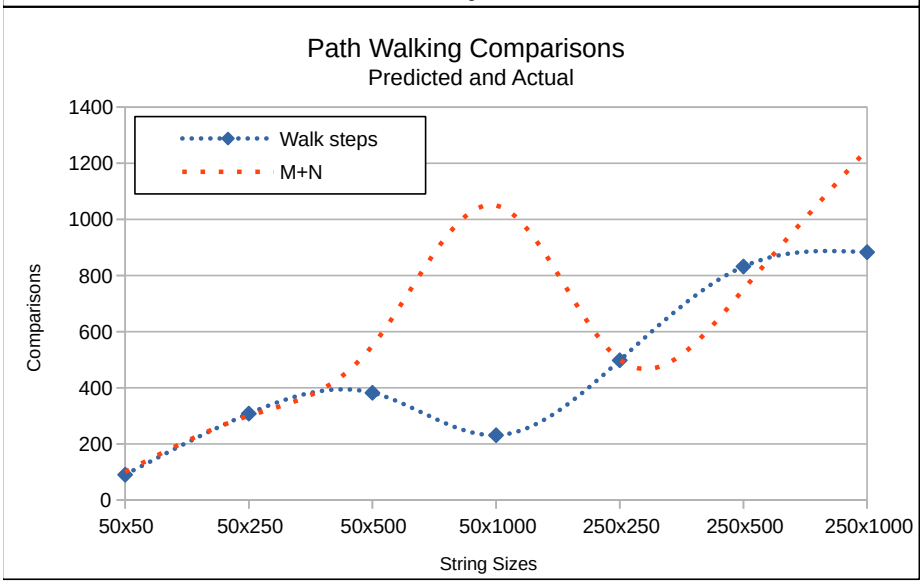


Figure 1) Measurements of relevant statistics compared to their predicted values. Comparisons exhibited a significant divergence from expected efficiency, while loop iterations matched almost exactly. Path walking had seemingly no correlation to expected efficiency and requires further study.

References (APA)

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Dynamic Programming. In *Introduction to algorithms* (pp. 405–405). essay, Mit Press