Hazelyn Cates
11/27/22
EN.605.620.81.FA22

Project 3

Implementation and Design

This program implements two functions that find the longest common substring between two DNA sequences. It requires the user to input the name of the text file that contains the sequences and to input the number of sequences in the file. The two functions are described in detail as follows:

LC_substring(s1, s2, s1_len, s2_len, string_count1, string_count2)

Takes six arguments: the length of the first string (s1), the length of the second string (s2), the lengths of strings 1 and 2 (s1_len and s2_len, respectively), and the indices at which strings 1 and 2 are stored in the "sequence" array after being read in from the file (string_count1 and string_count2, respectively).

This function begins by creating a dynamic programming table of size (s2_len + 1) rows and (s1_len + 1) columns, called "LC_suffix". This table stores results of subproblems, which in this case is whether or not the substrings match at a given position, indicated by the indices in the table.

The outer for loop "i" in this function iterates from 0 to size (s2_len + 1), which are the rows, and the inner for loop "j" iterates from 0 to size (s1_len + 1), which are the columns. Once in the inner for loop, values start getting assigned to the table by comparing the characters of the two sequences. In the first "if" statement, if either i or j is set to zero, which is the initial case, then the corresponding position in the table is set to zero. If the strings match at a certain index, then 1 plus the value to the left one and up one is added to the coordinate in the table where the match occurs. If there is no match between the strings, then the max value of the previous column and same row versus the same column and previous row is taken and assigned to that current index. This continues for the length of the strings until all spots in the table are filled.

Once this occurs, the **find_LC_suffix** function is called, which is described below.

find_LC_suffix(LC_suffix, s1, s2, s1_len, s2_len, string_count1, string_count2, start_substring)

Takes eight arguments: the dynamic programming table populated in the **LC_substring** function (LC_suffix), two strings (s1 and s2), the lengths of strings 1 and 2 (s1_len and s2_len, respectively), the indices at which the strings are stored in the "sequence" array after being read in from the file (string_count1 and string_count2) and the start time of the **LC_substring** function call.

This function aims to find the longest common suffix of two strings, meaning it starts comparing characters at the ends of the strings and working backwards. It starts by establishing the length of the longest common substring (LCS), which is given by the last index in the LC_suffix table (last row, right-most column). An array the size of the LCS, called "LCS", is then initialized to hold the resulting LCS. A variable to record the number of comparisons is also initialized to zero, and m and n are variables set to the lengths of string 1 and string 2, respectively.

The program then enters a while loop that loops through the strings as long as their lengths are not equal to zero. Inside the while loop, there are three conditionals:

The first if statement checks if string 1 and string 2 are equal at the indices m-1 and n-1, respectively. If they are, then the corresponding character is added to the end of the "LCS" array, and the length of both strings and the length of the LCS gets decremented by 1 and the comparisons gets incremented by 1.

If the first conditional fails, the elif statement compares the value in the LC_suffix table at the same column and previous row to the value in the LC_suffix table at the same row but previous column. If the first of the two values are greater, indicating that string 2 is longer, n gets decremented by 1 and m stays the same.

The last conditional is executed if the first two fail, indicating that string 1 is longer, and m gets decremented by 1 and n stays the same.

Once the LCS is found, the function then computes the total number of possible substring combinations that could theoretically occur between the two strings. It is important to note that this result is irrespective of if the strings match, but just has to do with string length using combinatorics (see "Efficiency and Analysis" section for more detail). The total time to execute the function call is also calculated.

Lastly, the function writes the results to the output file: the strings being compared and their lengths, the LCS and its length, the number of comparisons carried out, the total number of theoretical substring combinations between the two strings, and the execution time for the function call.

Efficiency and Analysis

Dynamic programming is a way to optimize a program that would typically use a recursive method or a brute force method by storing results to previously solved subproblems in a table instead of recalculating the same subproblems more than once ("Dynamic Programming", 2022). This has positive implications in efficiency, with dynamic programming having the ability to reduce run time of a recursive program from exponential to polynomial ("Dynamic Programming", 2022). Generally speaking, using a dynamic programming approach has a time complexity of O(n * c), where n is the number of subproblems and c represents the time needed to carry out that subproblem, which can be constant (in which c = 1) or some other variable (M, 2018).

In the case of this problem, finding the longest common substring between two strings, a dynamic programming approach has the ability to reduce the time complexity from $O(n * m^2)$ via a brute force method where there are m^2 possible substrings, or $O(2^n)$ via a worst-case recursive approach, to O(n * m), where n and m are the lengths of the two strings ("Longest Common", 2022; "Longest Common", n.d.). In the dynamic programming approach of this problem, the optimal substructure is given through computing the longest common suffix, that is starting at the end of both of the strings being compared and checking if they match, working backwards to the front of the strings. More specially, by starting at the ends of both strings, if the characters match, the length of both strings gets

reduced by one. If they do not match, the result is given as zero and the next pair of characters are investigated ("Longest Common", 2022). Regarding the space complexity of the dynamic programming approach of finding the longest common substring, it takes O(n * m) space to store both strings ("Longest Common", 2022).

When doing comparisons using the dynamic programming approach, the best case is that, when the strings are the same length, all the characters match, or if the strings vary in length, then the longest common substring will be the length of the shorter string, since the shortest string limits the length of the LCS. Compared to the worst case would, which is if none of the characters match. However, in either case, the entire length of both strings must be traversed, therefore the time and space complexity are driven by the string with the longer length.

Additionally, an important aspect of implementing a dynamic programming approach is the use of a dynamic programming (DP) table. As mentioned above, a DP approach of solving a problem involves storing the results of subproblems so they can be accessed if the subproblem is encountered at a later time to prevent redundant recalculation. These subproblems are stored in a DP table, where each element in the table depends on the one before it and serves as a visualization of the results ("DP Table", n.d.). Consider the following example using two strings of different lengths:

string1 = ATGCCC

string2 = ATCC

The goal is to find the longest common substring between the two strings, allowing for gaps between matching characters in the strings. The expected result is: "ATCC". Using a DP approach, a table consisting of seven columns (string1 length + 1) and five rows (string2 length + 1) is built and filled in corresponding to matching characters between the strings to yield the following complete DP table:

		A	**T**	**G**	**C**	**C**	**C**
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
T	0	1	2	2	2	2	2
C	0	1	2	2	3	3	3
C	0	1	2	2	3	4	4

The red arrows in the above table indicate where matches between the strings occurred, and the yellow highlight shows the "path" of the matches. As stated above, the values in a DP table are dependent on previous values. So in this case, in the above table, two characters in the string matching means that the value at the location up one and left one gets incremented by one and placed in the spot where the matching occurs (i.e. where the matching characters "intersect"). For instance, the A's matching increments the 0 to a 1, the T's matching increments the 1 to a 2, there is no change when looking at the G, the first C's matching increments the 2 to a 3, and the second C's matching increments the 3 to a 4, and the third C in string1 does not match anything in string2 since string1 is larger, therefore there is no change in value in the last position in the table. In addition to giving information

about where in the strings the matching occurred, the DP table also tells you how long the longest common substring is, given by the last index in the table, which has a value of 4.

Following the arrows in the table, it can be gathered that the longest common substring between string1 and string2 is ATCC, which was the expected result in both characters and length.

When continuing to explore the longest common substring, an additional calculation, while not dependent on if the strings match, is determining the number of possible substring combinations between two strings of lengths n and m, irrespective of if there is matching or not. This equation is given by the following combinations (or choose) equation, also referred to as the binomial coefficient:

$$C(n,m) = \frac{n!}{m!(n-m)!}$$ (Equation 1, "n Choose k", n.d.)

Where n is the length of the longer string and m is the length of the shorter string of the pair. In the above equation, the result tells you how many substrings of size m can be generated from a string of size n, so called "n choose m" ("n Choose k", n.d.). It is important to note that the above equation does not take order into account ("n Choose k", n.d.). That is, how many strings of length m (the shorter string) can be picked from a string of length n (longer string) (Furey, n.d.). As can be seen in equation 1, the length of string n, that is, the longer string, is directly related to the number of possible substring combinations, meaning that as n increases, so does C. Additionally, it can be observed that the closer n and m are in value, the smaller the number of possible combinations there will be.

Results

Four input files total were utilized for testing this program. The first input file contained four sequences ranging in length from 16 to 40, the second input file contained four sequences ranging in length from 50 to 100, the third input file contained four sequences ranging in length from 101 to 200, and the fourth input file contained four sequences ranging in length from 201 to 400.

The number of comparisons conducted to find the longest common substring (LCS) from each input file related to sequence length can be seen in the following tables:

Strings compared:	**1st string length**	**2nd string length**	**LCS length:**	**# of comparisons:**
1 and 2	29	28	20	33
1 and 3	29	40	19	50
1 and 4	29	16	11	33
2 and 3	28	40	20	45
2 and 4	28	16	12	29
3 and 4	40	16	13	42

Table 1: Results of first dataset, showing the length of the strings, the length of the longest common substring (LCS) and number of comparisons between each string pair.

Strings compared:	1st string length	2nd string length	LCS length:	# of comparisons:
1 and 2	50	74	35	83
1 and 3	50	100	42	94
1 and 4	50	65	30	85
2 and 3	74	100	52	121
2 and 4	74	65	36	97
3 and 4	100	65	49	115

Table 2: Results of second dataset, showing the length of the strings, the length of the longest common substring (LCS) and number of comparisons between each string pair.

Strings compared:	1st string length	2nd string length	LCS length:	# of comparisons:
1 and 2	101	145	79	166
1 and 3	101	200	81	193
1 and 4	101	178	82	197
2 and 3	145	200	103	224
2 and 4	145	178	102	213
3 and 4	200	178	117	260

Table 3: Results of third dataset, showing the length of the strings, the length of the longest common substring (LCS) and number of comparisons between each string pair.

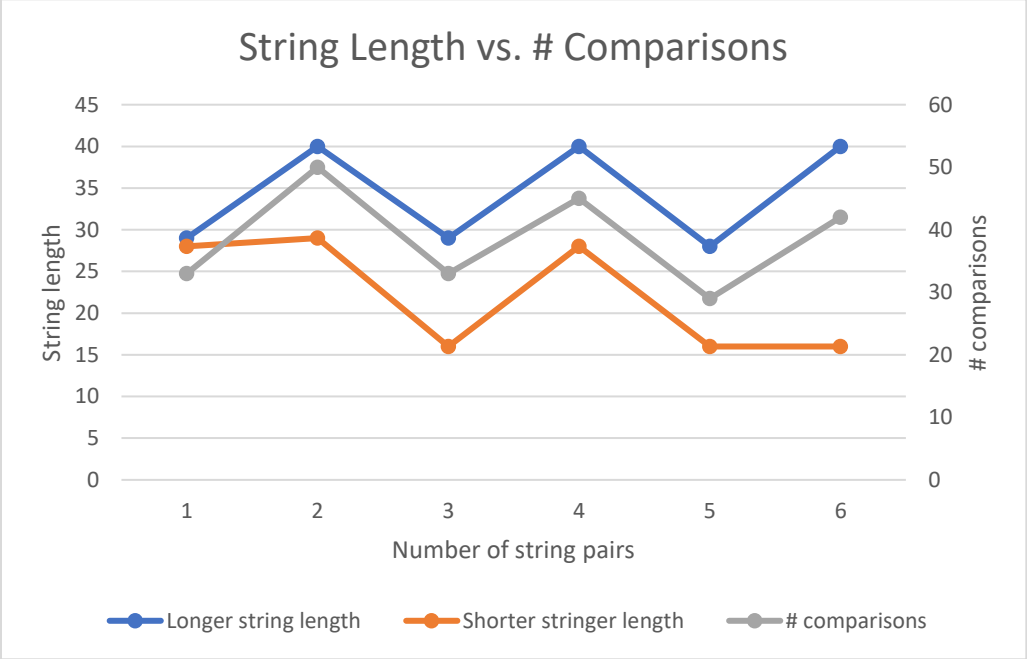
Strings compared:	1st string length	2nd string length	LCS length:	# of comparisons:
1 and 2	201	300	149	349
1 and 3	201	400	177	421
1 and 4	201	267	146	320
2 and 3	300	400	221	474
2 and 4	300	267	188	384
3 and 4	400	267	201	466

Table 4: Results of fourth dataset, showing the length of the strings, the length of the longest common substring (LCS) and number of comparisons between each string pair.

As can be seen in the above tables, the lengths of the strings directly impact the number of comparisons needed between two strings to find the longest common substring. More specifically, as

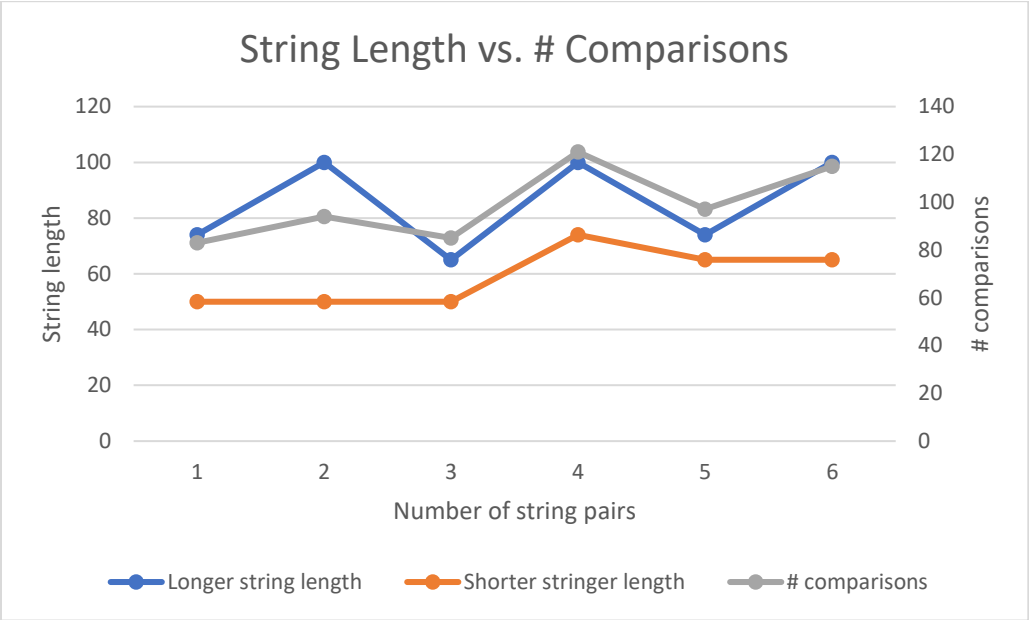
can be observed in the above tables, the closer the strings are in length, a fewer number of comparisons are needed versus if the strings differ significantly in length. This makes sense, as the number of comparisons is dependent on the length of the longest string in the pair since comparisons between the strings are being made character by character.

The relationship between the number of comparisons and the string length is better visualized in the following graphs for each of the four datasets. Note in the graphs, the left primary y-axis represents string length, the right secondary y-axis represents the number of comparisons, and the x-axis refers to each row in each table, where x = 1 refers to the comparison between strings 1 and 2, x = 2 refers to comparison between strings 1 and 3 and so on for each dataset.

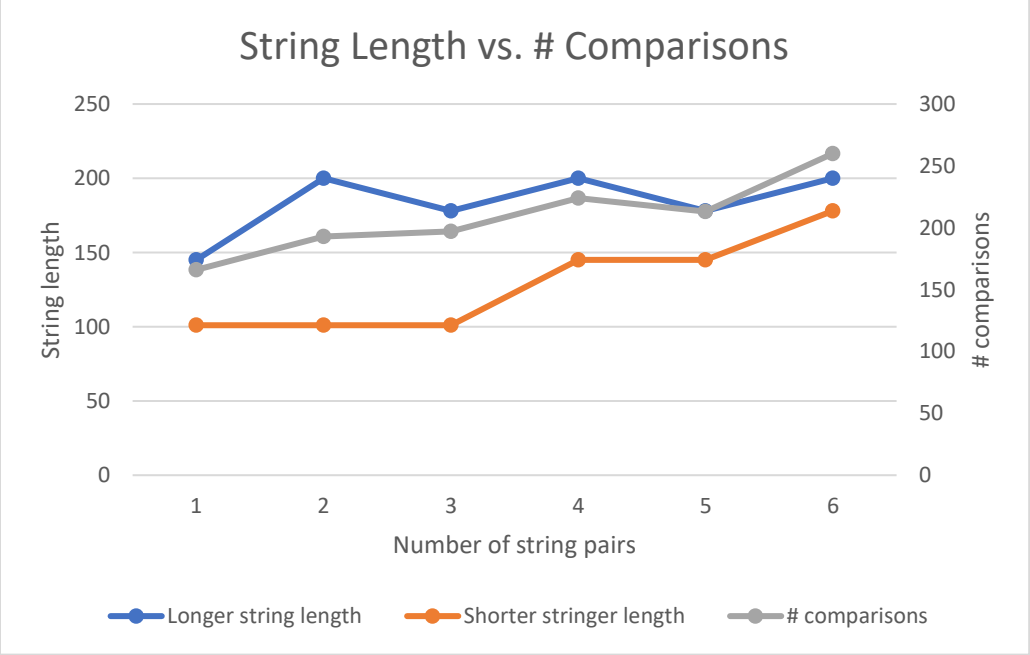


Graph 1: representing the data shown in table 1. The longer string of the pair at each corresponding x coordinate is shown by the blue line, the shorter string of the pair shown by the orange line, and the number of comparisons per string pair is shown by the gray line. The dots that line up vertically at each x-coordinate correlate to the same respective string pair.

Hazelyn Cates
11/27/22
EN.605.620.81.FA22

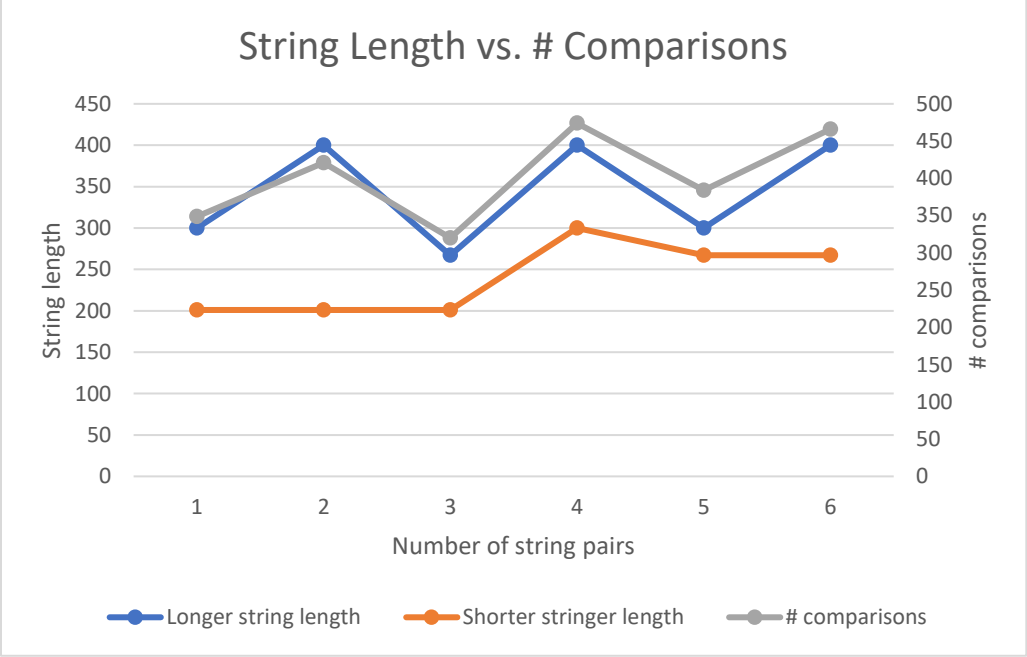


Graph 2: representing the data shown in table 2. The longer string of the pair at each corresponding x coordinate is shown by the blue line, the shorter string of the pair shown by the orange line, and the number of comparisons per string pair is shown by the gray line. The dots that line up vertically at each x-coordinate correlate to the same respective string pair.



Hazelyn Cates
11/27/22
EN.605.620.81.FA22

Graph 3: representing the data shown in table 3. The longer string of the pair at each corresponding x coordinate is shown by the blue line, the shorter string of the pair shown by the orange line, and the number of comparisons per string pair is shown by the gray line. The dots that line up vertically at each x-coordinate correlate to the same respective string pair.



Graph 4: representing the data shown in table 3. The longer string of the pair at each corresponding x coordinate is shown by the blue line, the shorter string of the pair shown by the orange line, and the number of comparisons per string pair is shown by the gray line. The dots that line up vertically at each x-coordinate correlate to the same respective string pair.

As can be observed in each of the four graphs above, the blue lines corresponding to the longer string length in each comparison pair in each dataset closely follows the gray line, which shows the number of comparisons needed to generate the longest common substring for each string pair. This suggests, as mentioned above, that the number of comparisons needed to find the longest common substring between two strings is dependent on the length of the longest string. Again, this is expected, since the dynamic programming approach finds the longest common suffix, and the entire length of both strings must be traversed. Therefore, it is possible that a substantial amount of comparisons have to be done before the strings even begin to overlap (i.e. there is an overhang of bases in the longer string against the shorter string). Given this, it is observed that the efficiency is mainly driven by the length of the longer string.

As stated above, the dynamic programming approach of finding the longest common substring between a pair of strings follows a time complexity directly related to the sizes of the strings in the pair (m and n), that is, O(m*n).

Additionally, as would be expected, the number of total possible substring combinations, calculated using equation 1 above, was observed to increase as the string lengths increased. The following tables and graphs show the trend between all possible substring combinations and string lengths, and are built from the same datasets as the results in tables 1-4 and graphs 1-4, with table and graph 5 correlating the dataset 1, table and graph 6 correlating to dataset 2, table and graph 7 correlating to dataset 3, and table and graph 8 correlating to dataset 4:

Strings compared:	**1st string length**	**2nd string length**	**# all possible substring combinations**
1 and 2	29	28	29
1 and 3	29	40	2.31×10^{9}
1 and 4	29	16	6.79×10^{7}
2 and 3	28	40	5.59×10^{9}
2 and 4	28	16	3.04×10^{7}
3 and 4	40	16	6.29×10^{10}

Table 5: Results of first dataset showing lengths of the strings in each pair and the calculated total number of all possible substring combinations for each pair.

Strings compared:	**1st string length**	**2nd string length**	**# all possible substring combinations**
1 and 2	50	74	1.75×10^{19}
1 and 3	50	100	1.01×10^{29}
1 and 4	50	65	2.07×10^{14}
2 and 3	74	100	7.00×10^{23}
2 and 4	74	65	1.11×10^{11}
3 and 4	100	65	1.10×10^{27}

Table 6: Results of second dataset showing lengths of the strings in each pair and the calculated total number of all possible substring combinations for each pair.

Strings compared:	**1st string length**	**2nd string length**	**# all possible substring combinations**
1 and 2	101	145	3.21×10^{37}
1 and 3	101	200	8.97×10^{58}
1 and 4	101	178	4.56×10^{51}

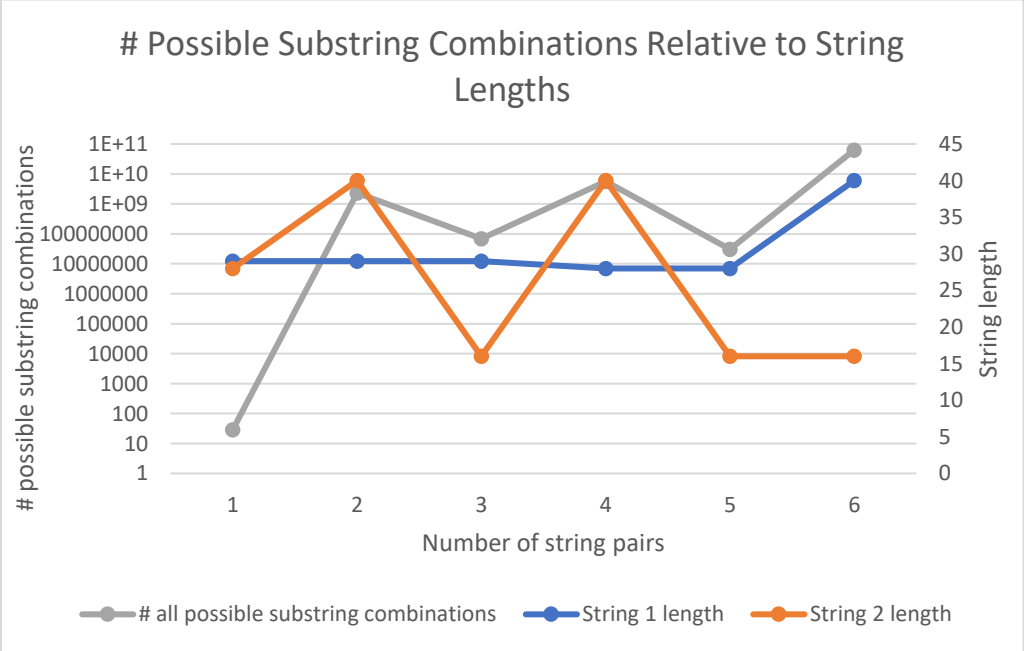
2 and 3	145	200	7.72×10^{49}
2 and 4	145	178	8.92×10^{35}
3 and 4	200	178	1.13×10^{29}

Table 7: Results of third dataset showing lengths of the strings in each pair and the calculated total number of all possible substring combinations for each pair.

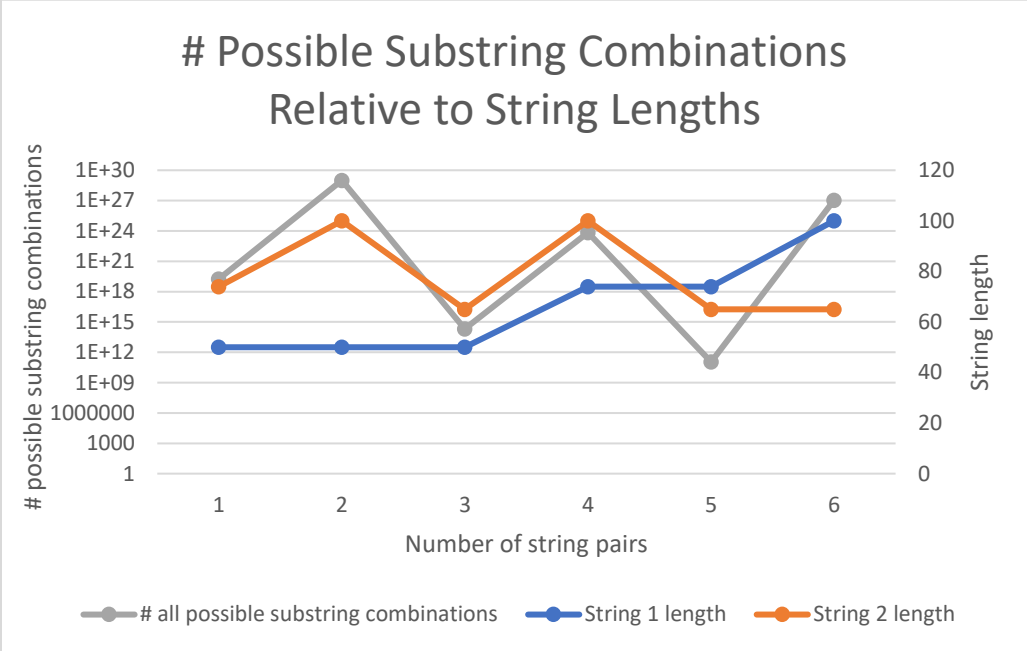
Strings compared:	**1st string length**	**2nd string length**	**# all possible substring combinations**
1 and 2	201	300	2.07×10^{81}
1 and 3	201	400	1.02×10^{119}
1 and 4	201	267	3.97×10^{63}
2 and 3	300	400	2.24×10^{96}
2 and 4	300	267	1.03×10^{44}
3 and 4	400	267	1.26×10^{109}

Table 8: Results of fourth dataset showing lengths of the strings in each pair and the calculated total number of all possible substring combinations for each pair.

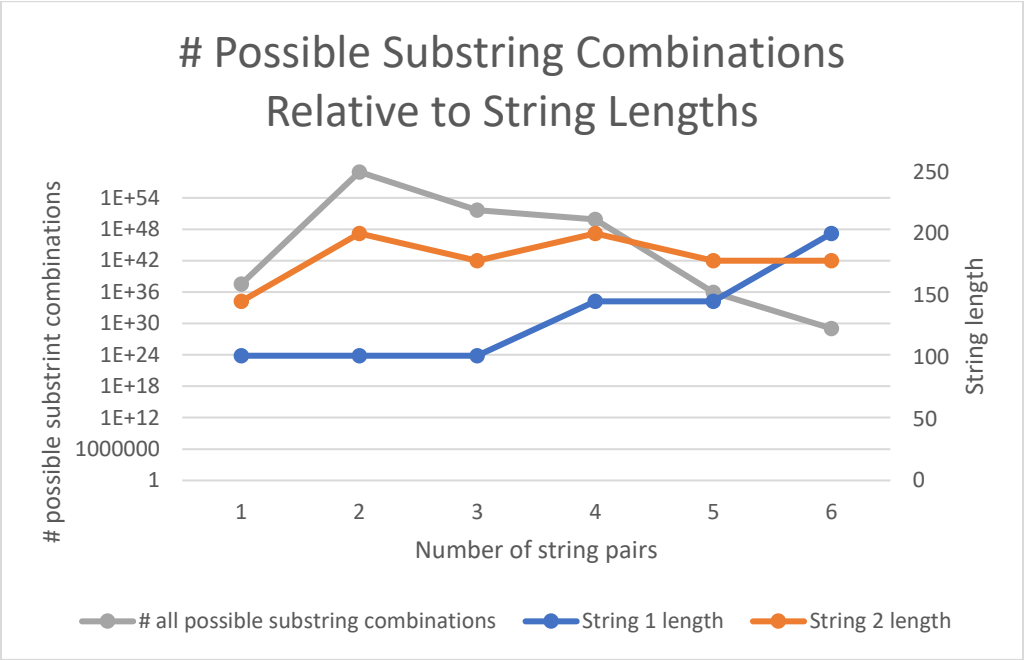
The relationship between string length and all possible substring combinations shown in tables 5-8 is further visualized in the following graphs:



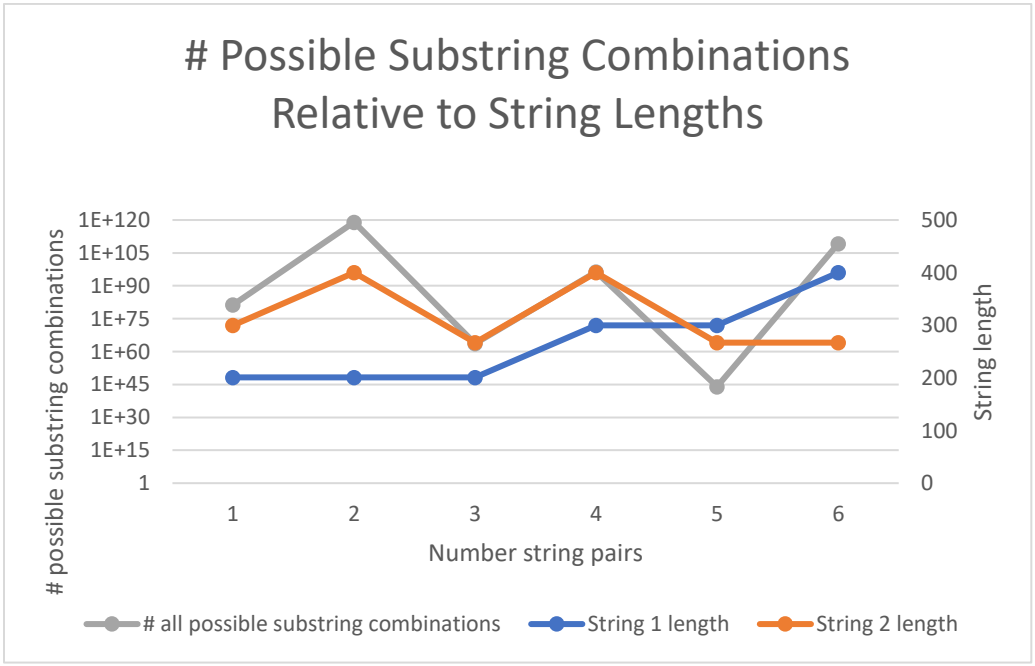
Graph 5: Shows the relationship of the strings in table 5 (blue and orange lines) and the number of all possible substring combinations between each pair of strings (gray line). *Note: the primary y-axis is on a logarithmic scale.*



Graph 6: Shows the relationship of the strings in table 6 (blue and orange lines) and the number of all possible substring combinations between each pair of strings (gray line). *Note: the primary y-axis is on a logarithmic scale.*



Graph 7: Shows the relationship of the strings in table 7 (blue and orange lines) and the number of all possible substring combinations between each pair of strings (gray line). *Note: the primary y-axis is on a logarithmic scale.*



Graph 8: Shows the relationship of the strings in table 8 (blue and orange lines) and the number of all possible substring combinations between each pair of strings (gray line). *Note: the primary y-axis is on a logarithmic scale.*

As can be seen in tables 5-8 and better visualized in graphs 5-8, the closer the strings are in length, the smaller the number of all possible substring combinations. The spikes in the number of possible substring combinations seen in graphs 6 through 8 correlate with the string pairs that have the largest difference in length (see tables 5-8 for string lengths corresponding to each dataset).

It is important to note that when interpreting these results in tables 5-8 and graphs 5-8, the total number of all possible substrings is irrespective of whether there is matching between the strings. That is, as stated above, these results show how many substrings of size m can be generated from a string of size n. Therefore, these results do not provide any information about efficiency, but rather provide a visual on how string length impacts how many substring combinations there theoretically can be.

Bioinformatics Application

The longest common substring (LCS) problem has a variety of applications to bioinformatics, including sequence alignment and pattern identifications, which in turn can help determine how DNA sequences are related, which could give hints into evolutionary history and biological functionality (Shikder et al, 2019). And in the case of protein sequences, it could give hints about protein function, location within the cell, whether it's hydrophobic or hydrophilic, etc.

In a paper from 2019, as the authors Shikder et al explained, as sequence length increases, the execution time of algorithms to find the longest common subsequence between two given DNA sequences also increases (2019). With algorithms traditionally running in O(n*m) time, where n and m are the lengths of the respective sequences, the resulting runtimes can eventually become unfeasible and unrealistic (Shikder et al, 2019). In an attempt to improve and potentially rectify this limiting aspect, Shikder et al introduced the development of three parallel LCS algorithms (2019). For background, a parallel algorithm is one that can implement multiple operations at the same time on different processors, and then combine the results of each individual processors to generate a final solution ("Parallel Algorithm", n.d.). The first algorithm presented by the authors was developed using message passing interface (MPI) on a distributed memory machine (i.e. each processor has its own memory), the second algorithm was developed on a shared memory machine (i.e. memories of each processor can be accessed at the same time) through using OpenMP (which an is an application programming interface), and the third introduced uses both shared and distributed memory (Shikder et al, 2019; Pardo et al, 2021; Chakraborty, 2019; Onsman, 2020).

Overall, Shikder et al found that the OpenMP-based approaches resulted in running a minimum of twice the speed as the best sequential LCS algorithm approaches and a maximum of seven times faster for both simulated and real DNA sequences of varying length (2019). The algorithm using both

shared and distributed memory had the worst performance regarding runtime and the MPI-based algorithm was also found to produce suboptimal results (Skikder et al, 2019).

Interestingly, string matching for DNA sequences can also be done using a hashing approach. A paper from 2021 by Karcioglu and Bulut describes two string matching algorithms, HqUF and HqUF-BM that expand upon an existing hashing algorithm called Hash-q. Generally speaking, string matching via hashing is a faster application, where the matching is based on how similar a pattern of interest is given the input sequence, and the pattern is processed through a function h prior to being aligned to the sequence of interest (Karcioglu & Bulut, 2021). The result of processing through function h is a hash value, also called a fingerprint (Karcioglu & Bulut, 2021). If the same hash value gets calculated more than once, the associated characters in the pattern are compared, meaning there is a match. Contrarily, if the hash value does not match one of those already calculated, it indicates a mismatch and the next character in the pattern is evaluated (Karcioglu & Bulut, 2021). The following image from Karcioglu and Bulut shows their approach:

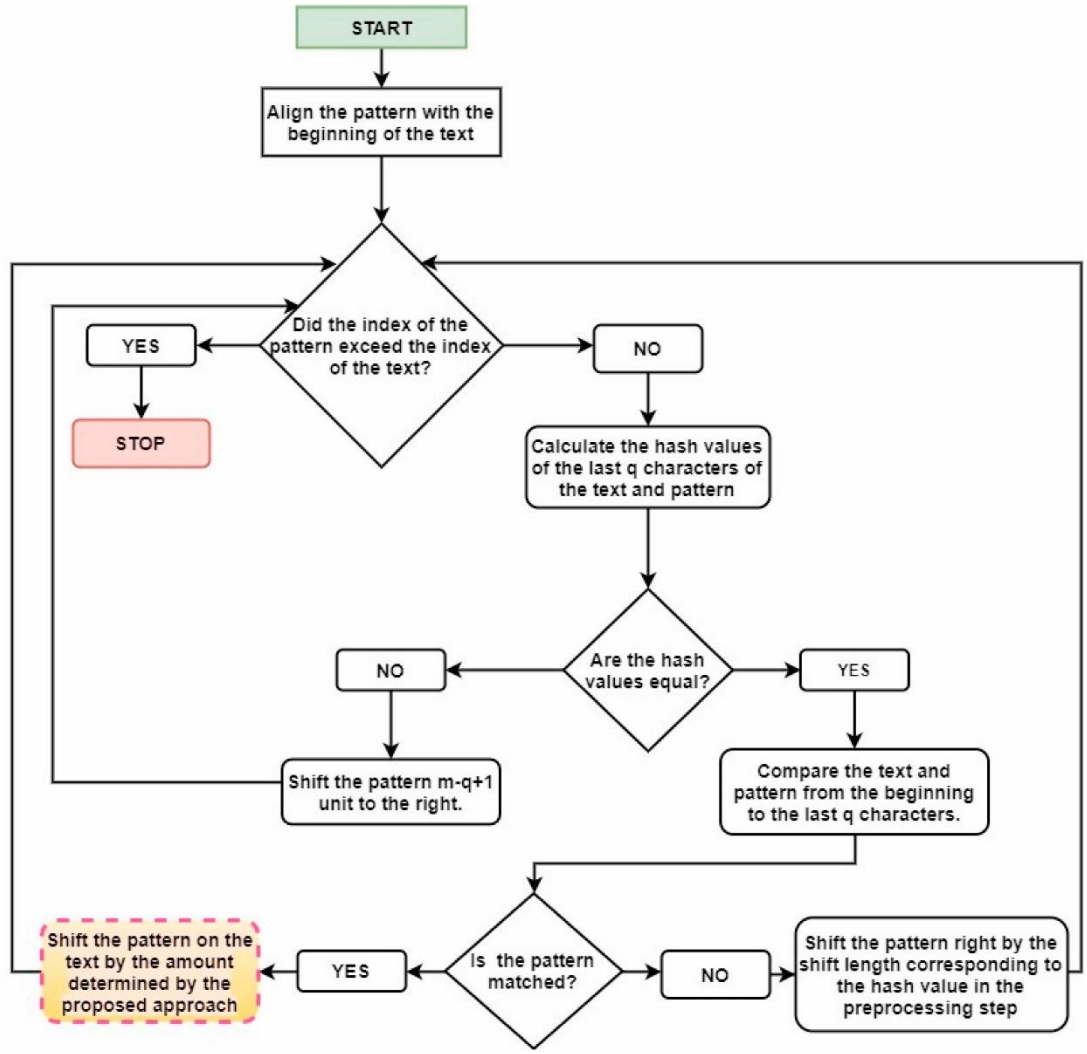


Fig. 2 from Karcioglu & Bulut (2021) showing the system of the proposed string-matching hashing algorithms.

Karcioglu & Bulut report that upon running the HqUF-BM algorithms on an E. coli dataset, human chromosome 1 dataset, and a synthetically developed dataset, a 83.11%, 72.85%, and 87.17% increase (respectively) in run time was achieved compared to the Hash-q algorithm (2021). The authors also note significant improvements in decreasing the number of character and hash comparisons using HqUF-BM versus Hash-q (Karcioglu & Bulut, 2021).

Given the above, as the number of sequences grows, so does the demand for efficient and accurate alignment algorithms, which stem from finding the longest common substring between DNA sequences. It is easy to see the importance of the studies described above, as sequence alignments have implications in organism classification, disease identification (i.e. genetic diseases or cancer), and phylogenetic and other evolutionary studies.

Reflection

This project was very helpful in furthering my understanding of dynamic programming and its implementation and benefits, namely in efficiency. In the case of this project, being able to reduce the time complexity to O(m * n) with respect to the string length is substantial, given that the brute force method takes $O(n * m^2)$ and the recursive method takes exponential time of $O(2^n)$ (see section "Efficiency and Analysis").

It took some time and reading to understand how the dynamic programming table was built and what the values in it meant, as well as how to traverse the table to determine the longest common substring. When writing the program, the tasks of building the DP table and then finding the longest common substring via finding the longest common suffix were split into two separate functions to improve readability and exactly differentiate which tasks were being carried out and when. Given this, I did not run in to any issues with efficiency. Once again, with the file input, I was given another opportunity to use regex to only extract the sequence and store those sequences into a single array. This project also allowed me to maintain my familiarity with the time and math libraries.

Overall, I would not make any changes to the program or the function implementation, as the DP table is successfully and correctly built, and the output of the longest common substring is as expected by finding the longest common suffix. Ultimately, this project served as an excellent example of dynamic programming, giving the challenge of implementing a concept that originally seemed abstract and less than tangible into working code.

Hazelyn Cates
11/27/22
EN.605.620.81.FA22

References:

Efficiency and Analysis:

DP Table. (n.d.). CODEPATH. https://guides.codepath.com/compsci/DP-Table#:~:text=PagePage%20History-,Dynamic%20Programming%20Table,suffix%20or%20subsequence%20problem%20type

Dynamic Programming. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/dynamic-programming/

Furey, Edward. (n.d.). Combinations Calculator (nCr). *CalculatorSoup.* https://www.calculatorsoup.com/calculators/discretemathematics/combinations.php#:~:text=Combinations%20Formula%3A,n%20%E2%89%A5%20r%20%E2%89%A5%200.&text=Also%20referred%20toz%20as%20r,or%20%22n%20choose%20k.%22

Longest Common Subsequence. (n.d.). OpenGenus IQ. https://iq.opengenus.org/longest-common-subsequence/

Longest Common Substring | DP-29. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/longest-common-substring-dp-29/

M, Prajwal. (2018). Demystifying Dynamic Programming. *freeCodeCamp*. https://www.freecodecamp.org/news/demystifying-dynamic-programming-24fbdb831d3a/#:~:text=In%20Dynamic%20programming%20problems%2C%20Time,O(n%20*%201)

n Choose k Formula. (n.d.). Cuemath. https://www.cuemath.com/n-choose-k-formula/

Results:

Random DNA Sequence Generator. (n.d.). UC Riverside. http://www.faculty.ucr.edu/~mmaduro/random.htm

*Used to generate the random DNA sequences for datasets 2 through 4.

Bioinformatics Application:

Chakraborty, Arnab. (2019). What is OpenMP? *TutorialsPoint*. https://www.tutorialspoint.com/what-is-openmp

Karcioglu, Abdullah Ammar & Hasan Bulut. (2021). Improving hash-q exact string matching algorithm with perfect hashing for DNA sequences. *Computers in Biology and Medicine*, 131, 104292. https://doi.org/10.1016/j.compbiomed.2021.104292

Onsman, Alex. (2020). Shared Memory Model of Process Communication. *TutorialsPoint*. https://www.tutorialspoint.com/shared-memory-model-of-process-communication

Parallel Algorithm – Introduction. (n.d.). TutorialsPoint. https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_introduction.htm

Pardo, David et al. (2021). Chapter 9 - Parallel implementation. In Pardo et al (Eds.), *Modeling of Resistivity and Acoustic Borehole Logging Measurements Using Finite Element Methods.* (pp. 257-264). Elsevier. https://doi.org/10.1016/B978-0-12-821454-1.00017-0

Shikder, Rayhan et al. (2019). An OpenMP-based tool for finding longest common subsequence in bioinformatics. *BMC Research Notes, 12*(220). https://doi.org/10.1186/s13104-019-4256-6

Other programming resources used but not cited in paper directly:

Iyer, J. Varun. (n.d.). OpenGenus IQ. https://iq.opengenus.org/longest-common-suffix/

Longest Common Subsequence. (n.d.). Programiz. https://www.programiz.com/dsa/longest-common-subsequence

Longest Common Substring. (2022). InterviewBit. https://www.interviewbit.com/blog/longest-common-substring/