Programming Assignment 3: Analysis
Kevin Feng Yong Chen
Algorithms for Bioinformatics

Description of the Program

This program solves the longest common subsequence (LCS) problem. Given 2 sequences, this program outputs a LCS of the 2 sequences to the output file. The input file must start with 4 required sequences that are prefixed with "S1/S2/S3/S4 =" followed by the test sequences. The program outputs the LCS between each of the required sequences. The LCSs between the test cases and each of the required sequences were also computed and printed to the output file. The LCS problem was solved using a dynamic programming approach. Dynamic programming can be used because of the existence of optimal substructure and overlapping subproblems in the LCS problem. The LCS between 2 sequences were computed using a bottom-up approach. That means the LCS between 2 smaller subsequences are solved before using it to solve larger subsequences. A recursive algorithm could be used to solve the LCS problem as well. However, the runtime of a recursive algorithm would be exponential. Thus, a bottom up approach was chosen because of its faster runtime. More on the efficiency of dynamic programming will be discussed in later sections. The dynamic programming algorithm was implemented using 2 functions: lcs_length and print_lcs. lcs_length computes a 2D array, c, that records the length of the common subsequences between 2 input sequences. The 2D array was implemented using a 2D list in python. In other words, it is a list of lists. This data structure was chosen because it was the most similar to a table and therefore it was the most suitable data structure to implement a 2D array. An alternative to a 2D list would be a NumPy array. The print_lcs function was implemented to print out the LCS between 2 sequences using recursion. Recursion was used because the problem is naturally recursive. It is natural to break down the task of constructing a LCS into smaller tasks of constructing its subsequences. Further, recursion usually allows coders to write concise and elegant code. Both functions were implementations of the algorithms described in 15.4 of the text book. However, instead of maintaining an extra table, b, to construct the LCS, I have decided to just use the c table from lcs_length to print the LCS. Although this does not change the auxiliary space complexity, it optimizes the use of space. For details on the implementation, please refer to the source code in this assignment as well as section 15.4 of the text book.

The main function takes care of the input/output stream. It takes the sequences from the input file and computes the LCS between the sequences using 2 auxiliary functions, compare_S and compare_T. Both compare_S and compare_T utilize the lcs_length function and the print_lcs function to find and print out LCSs between sequences. The function compare_S compares each required sequence with each of the other required sequences and outputs the LCS for each pair. The function compare_T compares a test case with each of the required inputs and outputs the LCS for each pair. I chose to use auxiliary functions instead of writing the code directly in the main function because it makes my main function more concise and easier to follow for the readers. Modularizing my code also allows me to test each module extensively to ensure that they have no bugs. Both upper and lower cases are allowed in the input sequences. This allows for some flexibility in the inputs.

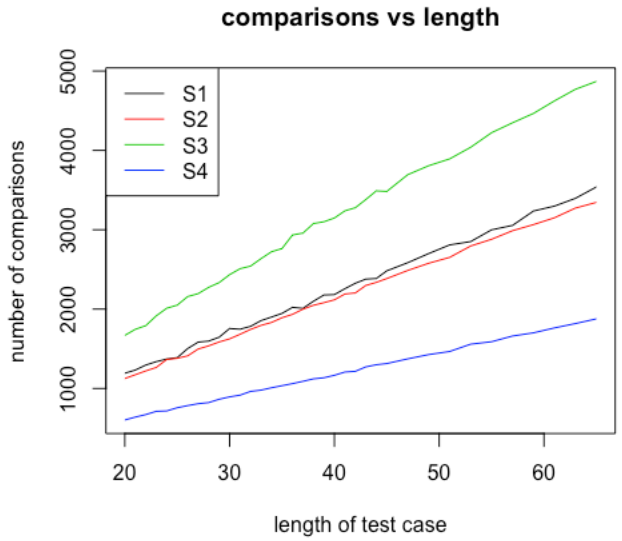
Theoretical Efficiency of the Dynamic Programming Approach

Based on 15.4 of the textbook, the runtime complexity of lcs_length is $\Theta(mn)$, m and n are the lengths of the 2 sequences that are being compared. In the algorithm, I created a 2D array with the dimension m*n and filling in each cell in the array takes O(1) time. We need to fill in each cell of the array regardless of the value of the inputs. Thus, we use big Theta as our asymptotic notation. The auxiliary space requirement of the algorithm is also $\Theta(mn)$ since we are creating a 2D array with dimension m*n. In addition, we need space to store the pointers and integers in the function but the asymptotic space

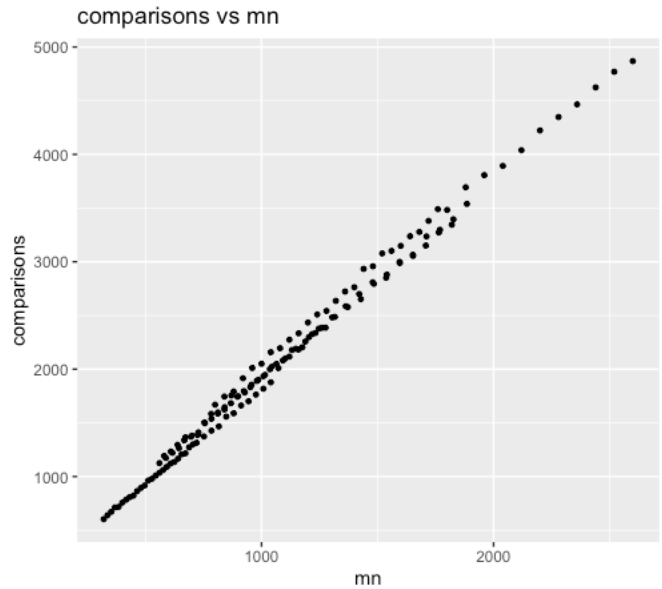
complexity should be dominated by the use of the 2D array. The runtime complexity for printing out the sequence of the LCS is O(m+n). This is because the algorithm starts with i=m and j=n and stops when either i or j is equal to 0. In each recursive call, at least one of i and j decreases by 1.

A Comparison of the Theoretical Efficiency to the Observed Efficiency

2 tables were made based on the observed values in this lab. They are in the supplemental material section. The 2 graphs shown below were generated based on the data from table 2 in the supplemental material section.



Graph 1. This graph is a visualization of table 2. S1-4 are the 4 required sequences



Graph 2. This graph plots the number of comparisons against the mn values which were calculated from multiplying the lengths of both sequences in each of the pairs from table 2.

The tables in the supplemental material section and the graphs above show the number of comparisons counted in the lcs_length function given the various pairs of input sequences. As shown in graph 1, for each of the 4 required sequences, the number of comparisons increases in a linear fashion as the length

of the test case it is compared to increases. This observation is consistent with the theoretical runtime of the algorithm which is $\Theta(mn)$, m and n are the lengths of the 2 sequences that are being compared. Based on the theoretical runtime, as the length of one of the 2 sequences increases, one would expect that the runtime increases proportionately. This was, in fact, observed in this lab. To visualize the observed value in another way, I have computed the theoretical value for each sequence pair from table 2 (see supplemental material) and compared them with their corresponding observed values. For example, for row 22 column S2, I would multiply 22 (the length of the test case) by 28 (the length of S2) to get mn. I then plotted the number of comparisons against the mn values (graph 2). Graph 2 showed that the number of comparisons is linear to mn. Therefore, the observed efficiency of the lcs_length algorithm is very consistent with the theoretical efficiency $\Theta(mn)$.

What I Learned

This programming assignment is very relevant to bioinformatics as pairwise sequence alignment is often conducted in the field to compare similarity between 2 sequences. From this assignment, I have learned how these alignments can be done efficiently. I have learned how dynamic programming can be applied in sequence alignment. I have also learned how to use regular expression in python to find a particular pattern in a string. For example, in my main function, I used regular expression to determine whether a sequence is a required sequence or a test case based on whether it has the prefix "S\d=". I have also used regular expression to check whether a sequence has any characters other than ATCG. The ability to use regular expression will be helpful for me in the future because bioinformaticians have to process genetic information in various file formats on a regular basis. Thus, being able to find a specific sequence in a file using regular expression will be a useful skill for me. This assignment has also greatly improved my understanding of dynamic programming. By implementing an algorithm from scratch and recording the efficiency of the algorithm while it is executing, I was able to learn and understand how dynamic programming works as well as why it is faster than a recursive approach. This assignment has also helped me understand what kind of problems are suitable for a dynamic programming approach. It has improved my ability to identify optimal substructure and overlapping subproblems in a problem.

What I Might Do Differently

If I could redo this assignment, I might choose to implement the print_lcs function using an iterative approach. As mentioned earlier, the task of printing the LCS is naturally recursive and so it is natural to implement this function using recursion. However, recursion tend to take up more computational resources and so an iterative approach may be more efficient. The print_lcs function from 15.4 of the text book can be easily converted to a while loop that terminates at i=0 or j=0. The function could start from i=m and j=n. Within each iteration, 3 if-else statements are tested just like in the original print_lcs function and at least one of i or j will decrease by 1 at the end of each iteration. Thus, the runtime would still be O(m+n). Another change I could make is to combine the lcs_length function with the print_lcs function. Since both functions are short, the combined function will not be too long. Since the lcs_length and the print_lcs functions were always executed together in this assignment, combining them into one would save me from having to execute each function separately. Thus, it allows me to write fewer lines of code.

Enhancement

As mentioned earlier, the use of space was reduced in this program. The algorithm described in 15.4 of the textbook uses 2 tables. One is used to record the length of the LCSs and the other was used to reconstruct the sequence of the LCSs. In my program, I have only used 1 table which records the length of the LCSs. Reconstruction of the LCS was done based on the data recorded in this table. By doing so, I have reduced the auxiliary space requirement by half. Although the asymptotic space complexity was still $\Theta(mn)$, the use of space was greatly reduced.

Another enhancement I did in this assignment was that I generated an extensive set of input, each with a different length. I collected a large amount of data to demonstrate the cost efficiency of the dynamic programming algorithm. I've also generated different graphs to help visualize the cost of the algorithm.

Application of Dynamic Programming to Bioinformatics

People in the field of bioinformatics often need to compare 2 DNA sequences to determine their similarity. There are many reasons why bioinformaticians want to determine the similarity between 2 sequences. Firstly, if an unknown sequence is similar to a gene with a known function, then that gives scientists information on the potential functionality of the unknown sequence. Secondly, 2 sequences that are similar may be close in evolutionary distance. Thus, sequence similarity provides hints on the phylogeny between organisms. One way to determine similarity between 2 sequences is by looking at the LCS between the 2. The longer the LCS between 2 sequences, the more similar they are. In a brute force approach to finding the LCS, we can enumerate all possible subsequence of one sequence and check if each subsequence is also a subsequence of the other sequence. We also have to keep track of the longest common subsequence we find. However, this approach is exponential. Suppose we have a sequence with 500 base pairs, then the number of subsequences in this sequence is 2^500. Given that bioinformaticians often have to align sequences that are over 500 base pairs, this approach is not practical because the runtime would be too long. By using dynamic programming to solve this problem bottom up, we could greatly reduce the runtime. Thus, dynamic programming algorithms are very important in sequence alignment because it allows us to solve the problem efficiently.

Another way to measure similarity between 2 sequences is by counting the number of changes needed to convert one sequence into another. We call this the editing distance. When aligning 1 sequence to another, we are most likely going to see gaps and mismatches in the alignment. The alignment below is an example of such mismatches and gaps. Both sequences were randomly generated by me.

```
          50         40                          30
EMBOSS TTACATG-ATTACAGTTCT---GG--------------TGTAGGT---GGT-AGC
       ::::: : :::     :::   ::               :::: ::   ::: :::
EMBOSS TTACAAGTATT-----TCTCACGGACCGCGTGCGTCAATGTATGTTACGGTGAGC
              20             30        40        50         60
```

The double dots between the 2 sequences represent identical bases. The hyphens represent gaps in the alignment. Gaps in an alignment indicates the occurrence of an insertion/deletion event in one of the 2 sequences. A mismatch in the alignment could mean that there was a substitution mutation in one of the 2 sequences. If we were to give a separate penalty score for each gap and mismatch, then we could define similarity in terms of scores. The lower the penalty score, the higher the similarity. We could then find an alignment that minimizes the penalty score. In other words, we can try to find an alignment that minimizes the cost of editing one sequence into another. Alternatively, we could also give rewards (i.e. positive scores) to matches and try to find the alignment with the maximum score. The scoring function we use is dependent on the alignment strategy we decide to adopt. Regardless of the strategy and the scoring system, if we used the brute force approach to find the alignment with the optimal alignment score, then it would take exponential time in terms of the length of the input sequences. Again, we could use dynamic programming to find the optimal alignment score between 2 sequences using a bottom-up approach. The algorithm to find the optimal alignment score would be similar to the algorithm we used in this programming assignment. The runtime would, again, be greatly reduced using a dynamic programming algorithm.

Supplemental Materials

	S1 (L=29)	S2 (L=28)	S3 (L=40)	S4 (L=16)
S1 (L=29)	na	1631	2358	875
S2 (L=28)		na	2235	837
S3 (L=40)			na	1169
S4 (L=16)				na

Table 1. This table shows the number of comparisons between each of the 4 required inputs to each other. The length of each of the required sequences are indicated (L).

test cases (in sequence length)	S1 (L=29)	S2 (L=28)	S3 (L=40)	S4 (L=16)
20	1192	1125	1669	603
21	1232	1173	1745	639
22	1295	1221	1792	671
23	1338	1265	1915	712
24	1371	1365	2011	717
25	1388	1380	2050	757
26	1503	1411	2158	784
27	1584	1497	2195	808
28	1597	1537	2274	822
29	1646	1586	2334	864
30	1755	1625	2434	894
31	1747	1683	2509	916
32	1783	1744	2542	963
33	1855	1795	2636	979
34	1900	1832	2723	1009
35	1946	1891	2763	1035
36	2022	1934	2934	1061
37	2010	1999	2958	1090
38	2099	2048	3078	1121
39	2178	2081	3101	1135
40	2183	2117	3148	1166
41	2257	2190	3238	1207
42	2326	2204	3278	1216
43	2378	2298	3381	1272
44	2386	2337	3490	1299
45	2482	2385	3483	1314
47	2585	2487	3694	1373
49	2700	2578	3807	1427

51	2809	2652	3893	1466
53	2851	2795	4039	1558
55	2999	2880	4223	1589
57	3055	2988	4348	1661
59	3236	3065	4466	1701
61	3298	3151	4624	1763
63	3396	3273	4770	1818
65	3539	3345	4869	1877

Table 2. This table shows the number of comparisons between each of the test cases and each of the required sequences (i.e. S1, S2, S3 and S4). The length of each of the required sequences are indicated (L).