Homework Assignment 4

Abstract

Smith-Waterman algorithm is a dynamic programming algorithm that performs a local sequence alignment. It determines similar regions between two nucleotide or protein sequences. The main goal of this report is to analyze the working and running time of the Smith-Waterman algorithm using the bottom up and top-down approach. We'll also be using the memoization technique for the top-down approach and check the performance of different approaches with different input sizes.

Introduction:

Dynamic Programming is an algorithmic technique for solving a problem by recursively breaking it down into simpler subproblems and using the fact that the optimal solution to the overall problem depends upon the optimal solution to its individual subproblems. The fundamental issue here is that certain subproblems are computed multiple times. Dynamic programming helps us solve recursive problems with a highly overlapping subproblem structure. We use memoization to avoid repeating subproblems. One way to avoid re-calculating the same subproblems again and again is to cache the results of those subproblems. The technique is:

- If the cache contains the result corresponding to a certain input, return the value from the cache.
- Otherwise, compute the result and store it in the cache, associating the result with the input that generated it. The next time the same subproblem needs to be solved, the corresponding result will already be in the cache.

We'll present and analyze the results of Smith-Waterman algorithm for different input sizes for both bottom up and top down with memoization approach in the next section and check their performance.

Results and Evaluation:

The average runtime of Smith-Waterman algorithm for different combination of input sizes with bottom-up and top-down with memoization approach is described in the table below.

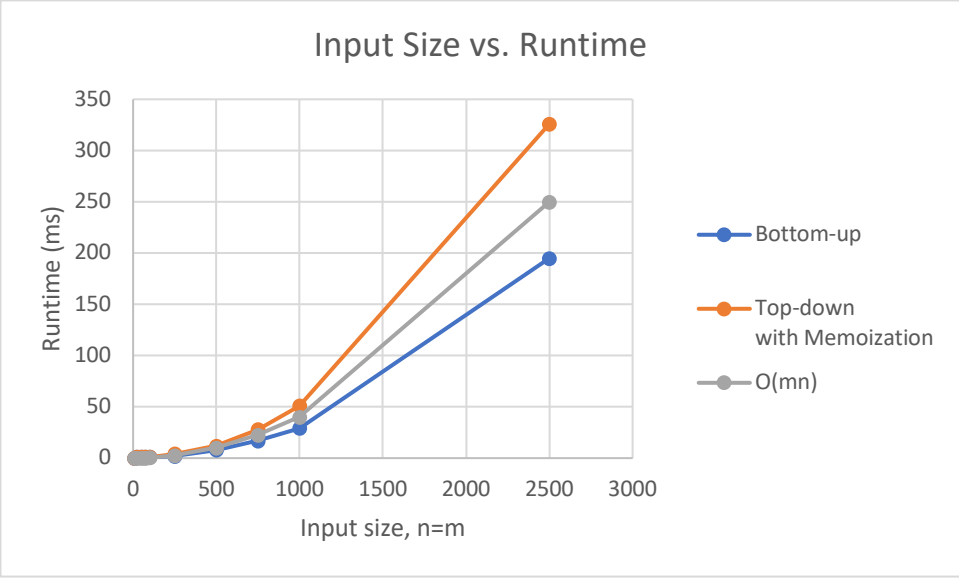
n=m	Bottom-up	Top-down with Memoization
10	0	0
25	1	1
50	1	1
75	1	1
100	1	1
250	2	4
500	8	12
750	17	28
1000	29	51

Smith-Waterman Algorithm Runtime in ms (Average Summary)

We can see that the bottom-up and top-down with memoization take almost the same amount of time to perform local sequence alignment for small input sizes. However, as the input size increases, the bottom-up approach is faster than the top-down approach. This is because the bottom-up approach includes the fact that iteration is usually faster than recursion and we don't need to initialize the matrix and test whether the sub-problems have already been computed. One disadvantage of bottom-up approach over memoizing is that it fills in the entire array even when it might be possible to solve the problem by looking at only a fraction of the array's cells.

The problem with the recursive solution is that the same subproblems get called many different times. There are exactly (m+1) (n+1) possible subproblems. If there are nearly 2^n recursive calls, some of these subproblems must be being solved over and over.
Using memoization, each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array. There are (m+1) (n+1) entries, so the total number of calls is at most 2(m+1) (n+1) +1 and the time is O(mn).

Below is the Input Size vs. Runtime comparison for different input sizes and dynamic programming strategies.



We can see that the runtime for both approaches increase exponentially with increase in the input size. The bottom-up approach, however, outperforms the top-down with memoization as the input size becomes large. We can also see that the top-down approach with memoization is comparable to the worst-case time complexity of O(mn).

Conclusion:

We're successfully able to test the runtime performance of Smith-Waterman algorithm using the bottom-up and top-down approach for different input sizes. From the above results and analysis, we can conclude that the bottom-up approach is faster than the top-down approach since its iterative and doesn't need to check if the subproblems are already computed. For the top-down approach, use of memoization decreases the runtime since we don't recompute the subproblems which have already been computed. This approach, however, is still less efficient than the bottom-up approach as the input size increases.

Dynamic Programming:

4. X = dcdcbacbbb

Y = acdccabdbb

H =

	-	a	c	d	c	c	a	b	d	b	b
-	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	2	1	0	0	0	2	1	0
c	0	0	2	1	4	3	2	1	1	1	0
d	0	0	1	4	3	3	2	1	3	2	1
c	0	0	2	3	6	5	4	3	2	2	1
b	0	0	1	2	5	5	4	6	5	4	4
a	0	2	1	1	4	4	7	6	5	4	3
c	0	1	4	3	3	6	6	6	5	4	3
b	0	0	3	3	2	5	5	8	7	7	6
b	0	0	2	2	2	4	4	7	7	9	9
b	0	0	1	1	1	3	3	6	6	9	11

P =

	-	a	c	d	c	c	a	b	d	b	b
-	0	0	0	0	0	0	0	0	0	0	0
d	0	↗	↗	↗	←	↗	↗	↗	↗	←	↗
c	0	↗	↗	↗	↗	↗	←	←	←	←	←
d	0	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗
c	0	↗	↗	↗	↗	↗	←	←	←	↗	↗
b	0	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗
a	0	↗	↗	←	↗	↗	↗	↗	←	↗	↗
c	0	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗
b	0	↗	↗	↗	↗	↗	←	↗	↗	↗	↗
b	0	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗
b	0	↗	↗	↗	↗	↗	↗	↗	↗	↗	↗

X = dcdc bac bbb
X' = cdcbacb-bb
Y' = cdcca-bdbb
Y = acdccabdbb

M(n,m) = 11

Homework 4

Exercise 15.1-2

Let us consider the prices up to length 4

Length i	1	2	3	4
Price p_i	1	5	8	9
Density p_i/i	1	2.5	2.66	2.25

The greedy strategy for a rod of length 4 cuts off a first piece of length 3 having maximum density. It will then cut off a piece of length 1.

$$\text{Total revenue} = 8 + 1 = 9$$

However, if we are given a rod of length 4, the optimal way of cutting the rod to maximize revenue is 2 pieces of length 2.

$$\text{Total revenue} = 5 + 5 = 10$$

This shows that the greedy strategy does not always determine an optimal way to cut rods.

Exercise 15.1-5

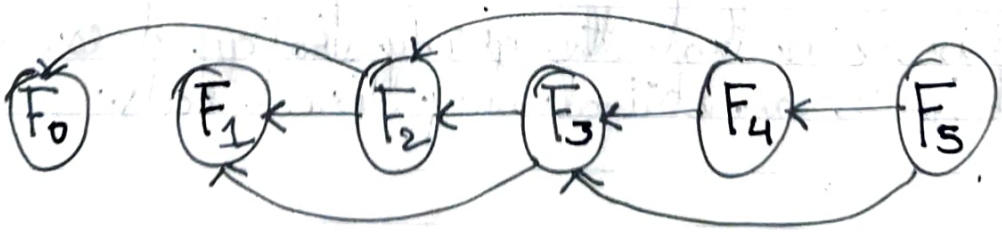
The recursive formula for the n^{th} element in the Fibonacci series is

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Dynamic Programming helps us solve recursive problems with a highly overlapping subproblem structure. One way to avoid re-calculating the same subproblems again and again is to cache the results of those subproblems. This technique is called memoization.

Store the sub-problems result so that we don't have to calculate again. So first check if solution is already available, if yes then use it else calculate and store it for future. The time complexity is $O(n)$.

The subproblem graph for Fibonacci series with $n=5$ is given below



The no. of vertices = $n+1$
The no. of edges = $2(n-1)$

Exercise 15.4-1

LCS of {1,0,0,1,0,1,0,1} and {0,1,0,1,1,0,1,1,0}

		0	1	0	1	1	0	1	1	0
	0	0	0	0	0	0	0	0	0	0
1	0	0	①	1	1	1	1	1	1	1
0	0	1	1	②	2	2	2	2	2	2
0	0	1	1	2	2	2	③	3	3	3
1	0	1	2	2	3	3	3	④	4	4
0	0	1	2	3	3	3	4	4	4	5
1	0	1	2	3	4	4	4	5	⑤	5
0	0	1	2	3	4	4	5	5	5	⑥
1	0	1	2	3	4	5	5	6	6	6

1 0 0 1 1 0

LCS = 1 0 0 1 1 0