SEQUENCE ALIGNMENT
CSCI-570 FALL 2021 FINAL PROJECT

Abid Hassan & Suvimal Yashraj

5414938221 3705389357

Table of Contents

1.Introduction

Strings arise very naturally in biology: an organism's genome—its full set of genetic material—is divided up into giant linear DNA molecules known as chromosomes, each of which serves conceptually as a one-dimensional chemical storage device. Indeed, it does not obscure reality very much to think of it as an enormous linear tape, containing a string over the alphabet {A, C, G, T}.

Sequence alignment is a fundamental problem that arises in comparing strings. In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. Sequence alignments are also used for non-biological sequences, such as calculating the distance cost between strings in a natural language or in financial data.

Suppose we are given two strings X and Y, where X consists of the sequence of symbols x1x2 ... xm and Y consists of the sequence of symbols y1y2 ... yn. Consider the sets {1, 2, . . . , m} and {1, 2, . . . , n} as representing the different positions in the strings X and Y. The matching of these strings is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an alignment if there are no "crossing" pairs i.e if (i, j), (i , j) ∈ M and i < i , then j < j . Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another.

Now the question is how to check the similarity between strings. The definition of similarity is based on finding the optimal alignment between strings X and Y, according to the following criteria.

- Suppose M is a given alignment between X and Y.
- First, there is a parameter δ > 0 that defines a gap penalty. For each position of X or Y that is not matched in M—it is a gap—we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a mismatch cost of αpq for lining up p with q. Thus, for each (i, j) ∈ M, we pay the appropriate mismatch cost αXiYj for lining up Xi with Yj. One generally assumes that αpp = 0 for each letter p—there is no mismatch cost to line up a letter with another copy of itself—although this will not be necessary in anything that follows.
- The cost of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

Why sequence alignment is useful?
The sequence of symbols in an organism's genome can be viewed as determining the properties of the organism. So, suppose we have two strains of bacteria, X and Y, which are closely related evolutionarily. Suppose further that we've determined that a certain substring in the DNA of X codes for a certain kind of toxin. Then, if we discover a very "similar" substring in the DNA of Y, we might be able to hypothesize, before performing any experiments at all, that this portion of the DNA in Y codes for a similar kind of toxin. This use of computation to guide decisions about biological experiments is one of the hallmarks of the field of computational biology.

2.Two Approaches

2.1 Dynamic Programming

Basic Intuition

In the optimal alignment M, either (m, n) is in M or (m, n) is not in M. (That is, either the last symbols in the two strings are matched to each other, or they aren’t.)

In an optimal alignment M, at least one of the following is true:

(i) (m, n) ∈ M

(ii) the mth position of X is not matched

(iii) the nth position of Y is not matched

Sub-problems

Now, let OPT(i, j) denote the minimum cost of an alignment between x1x2 ... xi and y1y2 ... yj. So for the three mentioned cases we get the following value for the alignment cost: -

(i) we pay αxmyn and then align x1x2 ... xm−1 as well as possible with y1y2 ... yn−1.
OPT(m, n) = αxmyn + OPT(m − 1, n − 1).

(ii) we pay a gap cost of δ since the mth position of X is not matched, and then we align x1x2 ... xm−1 as well as possible with y1y2 ... yn.
OPT(m, n) = δ + OPT(m − 1, n).

(iii) OPT(m, n) = δ + OPT(m, n − 1).

Recurrence Formula

The minimum alignment costs satisfy the following recurrence for i ≥ 1 and j ≥ 1: OPT(i, j) = min[αxiyj + OPT(i − 1, j − 1), δ + OPT(i − 1, j), δ + OPT(i, j − 1)].

For purposes of initialization, we note that OPT(i, 0) = OPT(0, i) = iδ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

Algorithm

```
Alignment(X,Y)
    Array A[0 . . . m,0... n]
    Initialize A[i, 0]= iδ for each i
    Initialize A[0, j]= jδ for each j
    For j = 1, . . . , n
        For i = 1, . . . , m
            A[i, j] = min[αxiyj + OPT(i − 1, j − 1), δ + OPT(i − 1, j), δ + OPT(i, j − 1)].
        Endfor
    Endfor
    Return A[m, n]
```

This gives us the optimal cost. To find the optimal alignment we just use the A matrix and retrace the steps from A[m, n] back to initial characters.

Time and Space Complexity
The running time is O(mn), because it takes constant time to determine the value in each of the mn cells of the array OPT; and the space requirement is O(mn) as well, since it was dominated by the cost of storing the array

Drawback
In biological applications of sequence alignment, however, one often compares very long strings against one another; and in these cases, the O(mn) space requirement can potentially be a more severe problem than the O(mn) time requirement. Suppose, for example, that we are comparing two strings of 100,000 symbols each. Depending on the underlying processor, the prospect of performing roughly 10 billion primitive operations might be less cause for worry than the prospect of working with a single 10-gigabyte array.

2.2 Divide and Conquer

The crucial observation is that to fill in an entry of the array A, the recurrence formula mentioned above only needs information from the current column of A and the previous column of A. Thus we will "collapse" the array A to an m × 2 array B: as the algorithm iterates through values of j, entries of the form B[i, 0] will hold the "previous" column's value A[i, j − 1], while entries of the form B[i, 1]will hold the "current" column's value A[i, j].

Space-Efficient-Alignment(X,Y)

```
    Array B[0 . . . m, 0 . . . 1]
    Initialize B[i, 0]= iδ for each i (just as in column 0 of A)
    For j = 1, . . . , n
        B[0, 1]= jδ (since this corresponds to entry A[0, j])
        For i = 1, . . . , m
            B[i, 1]= min[αxiyj + B[i − 1, 0], δ + B[i − 1, 1], δ + B[i, 0]]
        Endfor
        Move column 1 of B to column 0 to make room for next iteration:
            Update B[i, 0]= B[i, 1] for each i
    Endfor
```

It is easy to verify that when this algorithm completes, the array entry B[i, 1]holds the value of OPT(i, n) for i = 0, 1, . . . , m. Moreover, it uses O(mn) time and O(m) space. The problem is: where is the alignment itself? Since B at the end of the algorithm only contains the last two columns of the original dynamic programming array A, if we were to try tracing back to get the path, we'd run out of information after just these two columns.

For this we will be using a Divide and Conquer approach.

1. **Divide Step** – divide the string X in two, Left_X and Right_X by splitting in the middle. To divide the string Y we will use Dynamic Programming to determine the optimal split point. For this we will need to compute the Alignment cost between Left_X and each point in Y from left to right. And similarly compute the Alignment cost between Right_X and each point in Y from right to left.

Then we will compare and select the split point in Y having the least Alignment cost from both the X splits.

2. **Conquer Step** – Two recursive calls to the divide and conquer function, one having X_Left and Y from start till the break point and second one having X_Right and Y from split point to end.
3. **Combine Step** – We just need to concatenate the alignments found from the recursive calls.

Divide-and-Conquer-Alignment(X,Y)

```
    Let m be the number of symbols in X
    Let n be the number of symbols in Y
    If m ≤ 2 or n ≤ 2 then
        Compute optimal alignment using Alignment(X,Y)
    Call Space-Efficient-Alignment(X,Y[1 : n/2])
    Call Backward-Space-Efficient-Alignment(X,Y[n/2 + 1 : n])
    Let q be the index minimizing f(q, n/2) + g(q, n/2)
    Add (q, n/2) to global list P
    Divide-and-Conquer-Alignment(X[1 : q],Y[1 : n/2])
    Divide-and-Conquer-Alignment(X[q + 1 : n],Y[n/2 + 1 : n])
    Return P
```

Time and Space Complexity

For the combine step it just a string concatenate, hence this takes linear time. For the divide step let cmn denote the maximum running time of the algorithm on strings of length m and n. Now at each step the search space reduces by half so the total time is equal to:

$$cmn + .5cmn + .25cmn + 1.25cmn + \ldots. = 2cmn$$

Hence the we get O(mn).

As we seen above the space complexity = O(m)

Hence with the same time complexity we are saving huge amount on the space required in the Divide and Conquer approach.

3.Analysis & Plots

3.1 Dynamic Programming

The run time and space complexity of dynamic programming solution is bounded by O(m*n) where m & n (here m = n to plot graph) are the length of strings. We ran dynamic programming solution and plotted the time and space complexity graph as input of string length as follows:

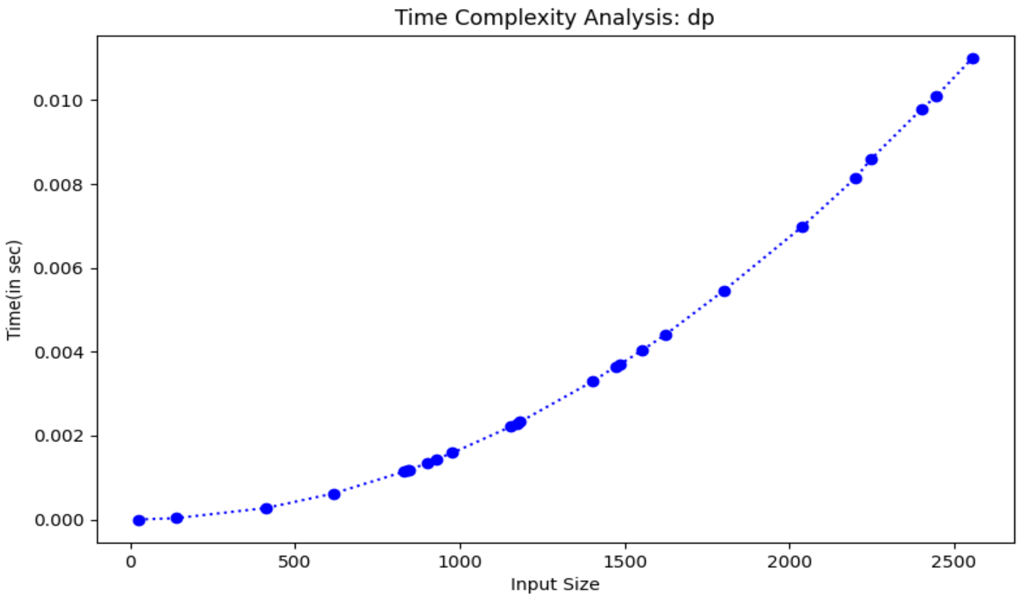

Figure1

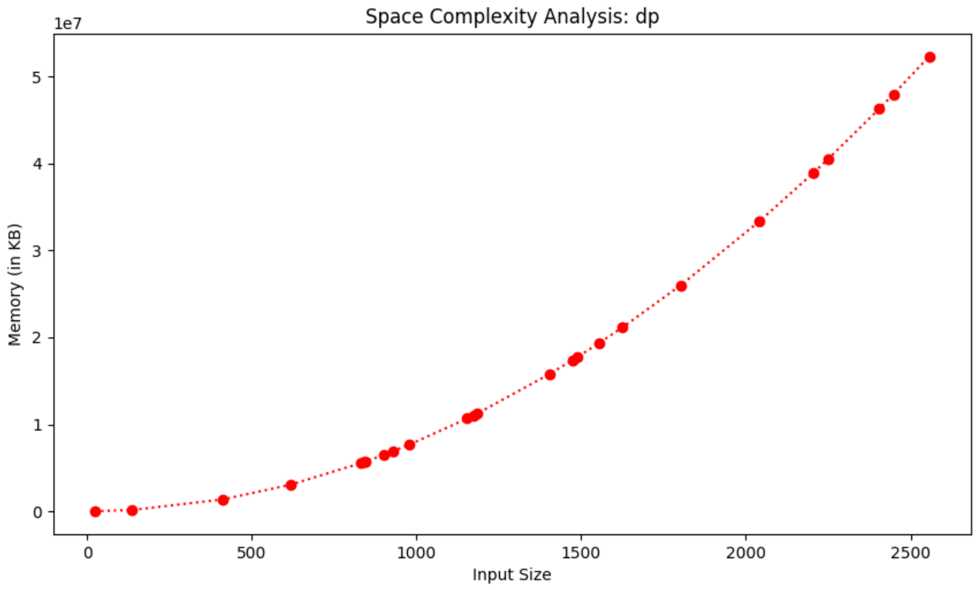

Figure2

3.2 Memory Efficient (DP and D&C)

Memory efficient run time complexity of dynamic programming solution is bounded by O(m*n) and space complexity is bounded by O(2*m) where m and n are the length of string X and Y respectively. We ran dynamic programming and divide & conquer strategy solution. The plotted graph of time and space complexity as input of string length as follows (to plot the graph, we have used m = n):

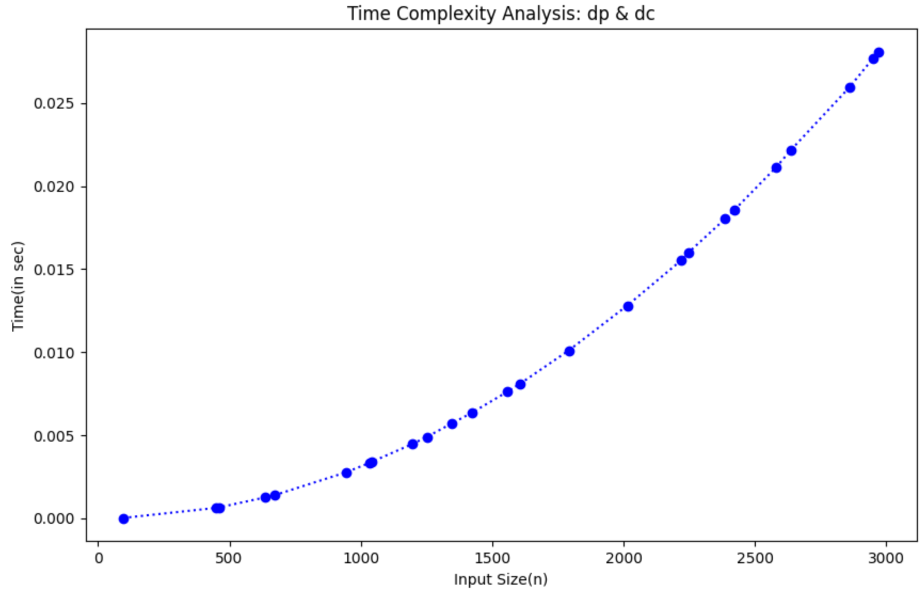

Figure3

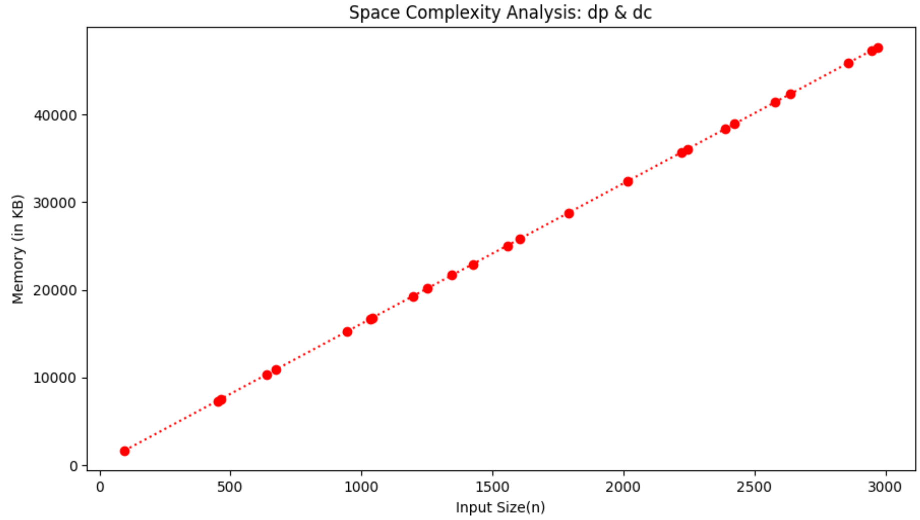

Figure4

3.3 Memory Efficient for Large Inputs

We ran our memory efficient algorithms on feasible large inputs of size 5000. The plotted graph of time and space complexity as input of string length as follows (to plot the graph, we have used m = n):

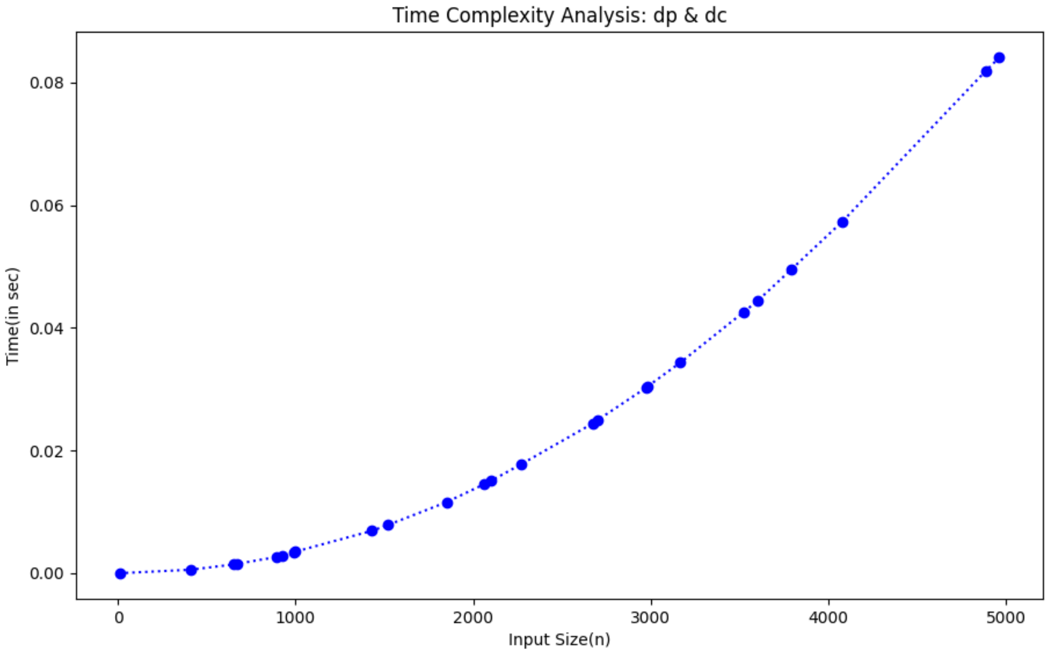

Figure5

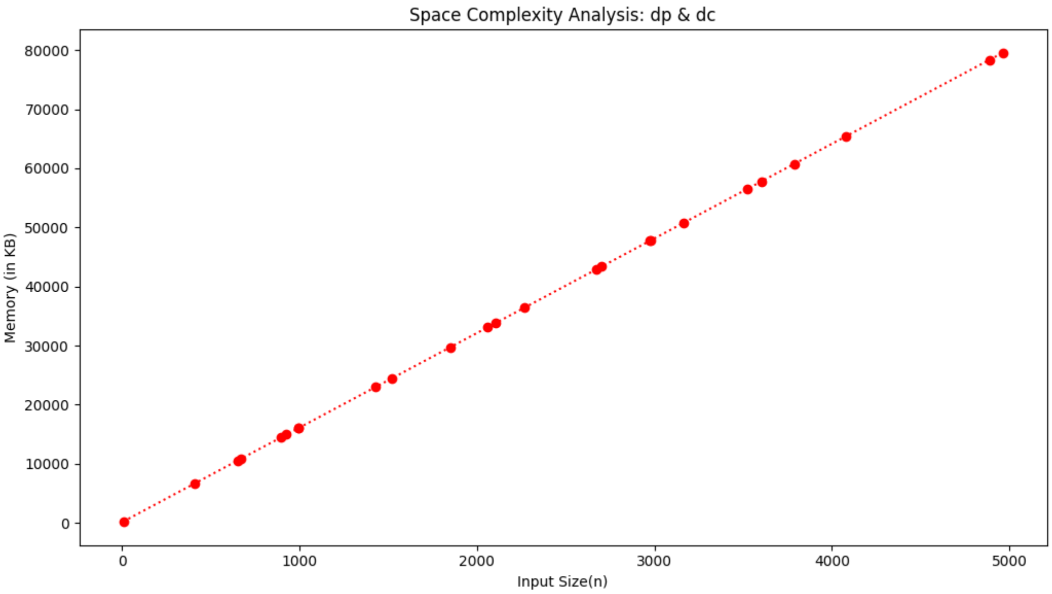

Figure6

4.Insights and Observation

After running both version of sequence alignment and analyzing the time and space complexity graph, we have following insights:

Run Time Comparison

In both simple dynamic programming approach and the divide and conquer approach, the run time is bounded by O(m*n) where m and n are length of strings respectively. But the divide and conquer approach almost has twice as many as computational steps as compared to the dynamic programming. This is quite visible in the Figure1 and Figure3. If we look at the input size of n=2000, then from Figure1 we can see the time taken for the dynamic programming is around 0.007 seconds. And if we see for the same input in Figure3, we can see the time taken for divide and conquer is around 0.013 seconds.
Hence, run time of the dynamic programming approach is better than the divide and conquer approach. This observation in line with the fact that we are running dynamic programming to find the split point for the Y string at every divide step.

Memory Utilization Comparison

In dynamic programming approach the space complexity is bounded by O(m*n). But in the divide & conquer approach, the space complexity is bounded by O(m). We can certainly infer this from the Figure2 and Figure4. If we consider the input n=2000, then from Figure2 we observe that the memory utilized is about $3.4*10^7$ KB. Now for the same input size we can observe from Figure4 that the memory utilized is around $5*10^4$ KB.
Hence, divide and conquer approach has huge memory savings as compared to the dynamic programming approach. This is in accordance with the fact that instead of maintaining m*n array in the memory in dynamic programming approach we are just maintaining an array of size 2*m in the divide and conquer approach.

Large Input Sizes

For large input sizes such as n=5000, we can perform the sequence alignment using the divide and conquer approach but for this input size we cannot find the solution using the dynamic programming approach. This is because normal computer architecture only supports 10^7 contiguous memory allocation, so the dynamic programming approach will exceed this limit whereas in the divide and conquer approach we were able to find the solution as we can see in Figure5 and Figure6.

5.Contributions

Coding the of the 2 approaches: -
Abid – 50%
Suvimal – 50%

Plots: -
Abid – 50%
Suvimal – 50%

Report: -
Abid – 50%
Suvimal – 50%