COT 6417 - Algorithms on Strings and Sequences
Fall 2020
Homework 3

Name: Nabila Shahnaz Khan

PID: 5067496

#Question 1:

Solution:

Here, F[n, m] denote the total number of alignments between two strings X and Y, of lengths n and m respectively. Let, F[i, j] denote the total number of alignments between the substring X[1,i] and Y[1,j], where 1 <= i <= n and 1 <= j <= m.

While aligning two strings X and Y using dynamic programming matrix, at position [i, j], the alignment cost will depend on the positions [i-1, j-1], [i-1, j], [i, j-1]. The number of possible path to access entry [i, j] is equal to the sum of the number of paths to access entries [i-1, j-1], [i-1, j], [i, j-1].
So, total possible alignments for entry [i, j] = total possible alignments for entry [i-1, j-1] + total possible alignments for entry [i-1, j] + total possible alignments for entry [i, j-1].

- **Recurrence Relation:**

Base Case: $F[i, 0] = 1$, *for* $0 \leq i \leq n$ *and* $F[0, j] = 1$, *for* $1 \leq j \leq m$
Then, for $1 <= i <= n$ *and* $1 <= j <= m$
$F[i, j] = F[i - 1, j - 1] + F[i - 1, j] + F[i, j - 1]$

- **Calculating F[3, 4] using the Recurrence Relation:**

m

n

	-	Y_1	Y_2	Y_3	Y_4
-	1	1	1	1	1
X_1	1	3	5	7	9
X_2	1	5	13	25	41
X_3	1	7	25	63	129

Using recurrence relation,
F[3, 4] = F[2, 3] + F[2, 4] + F[3, 3]
= (F[1, 2] + F[1, 3] + F[2, 2])
+ (F[1, 3] + F[1, 4] + F[2, 3])
+ (F[2, 2] + F[2, 3] + F[3, 2])
= (5 + 7 + 13) + (7 + 9 + 25) + (13 + 25 + 25)
= 129

#Question2:

Solution:

Here, suppose X and Y are two strings with length n and m respectively. In a unit cost model, the cost of a match is 0, and the cost of a substitution and a gap are both 1. Using the unit cost model, the recurrence relation for calculating the optimal edit distance between two strings is given below:

- **Recurrence Relation:**

Let M[i, j] denote the minimum cost of aligning X[1,i] and Y[1,j] using the unit cost model where 1 <= i <= n and 1 <= j <= m.

$$M[i, j] = \text{minimum} \begin{cases} M[i-1, j-1] + cost(x_i, y_j) \\ M[i-1, j] + 1 \\ M[i, j-1] + 1 \end{cases}$$

$$cost(x_i, y_j) = \begin{cases} 0, & x_i = y_j \\ 1, & otherwise \end{cases}$$

Base case:

*M[i, 0] = i * 1, for 0 ≤ i ≤ n and M[0, j] = j * 1, for 1 ≤ j ≤ m*

Here, M[n, m] is the optimal edit distance for X and Y.

- **Calculation of Edit Distance:**

Input: X = acagatta, Y = tagctta

Dynamic Programming Matrix:

	-	**t**	**a**	**g**	**c**	**t**	**t**	**a**
-	0	1	2	3	4	5	6	7
a	1	1	1	2	3	4	5	6
c	2	2	2	2	2	3	4	5
a	3	3	2	3	3	3	4	4
g	4	4	3	2	3	4	4	5
a	5	5	4	3	3	4	5	4
t	6	5	5	4	4	3	4	5
t	7	6	6	5	5	4	3	4
a	8	7	6	6	6	5	4	3

Backtracking Matrix:

Let, U = Up, L = Left, D = Diagonal

	-	t	a	g	c	t	t	a
-	**0**	L	L	L	L	L	L	L
a	**U**	D	D	L	L	L	L	D
c	U	**D**	D	D	D	L	L	L
a	U	U	**D**	D	D	D	D	D
g	U	D	U	**D**	L	D	D	D
a	U	D	D	U	**D**	D	D	D
t	U	D	U	U	D	**D**	D	L
t	U	D	D	U	D	D	**D**	L
a	U	U	D	U	D	U	U	**D**

Alignment:

g = gap, s = substitute, m = match

a	**c**	**a**	**g**	**a**	**t**	**t**	**a**
-	**t**	**a**	**g**	**c**	**t**	**t**	**a**
g	s	m	m	s	m	m	m

#Question 3:

Solution:

Here, S is a string of length n generated from a constant size alphabet. Query Q(i, j) should return the length of the longest common prefix of S[i, n] and S[j, n].

- **Construction of Suffix Tree:**

In order to preprocess string S, first we have to build a suffix tree T for string S. This will require O(n) runtime. In the suffix tree, the paths from leaf nodes i to root (path P_i) and from leaf j to root (path P_j) will represent the suffixes S[i, n] and S[j ,n] respectively. The longest common prefix (LCP) of substrings S[i, n] and S[j ,n] is same as the Longest Common Ancestor (LCA) of paths P_i and P_j in tree T.

- **Calculating LCA:**

LCA of nodes i and j, known as lca(i, j), can be calculated in constant time using the concept of Euler Tour of tree T and RMQ (Range Minimum Queries). For that, we will need to use three additional arrays known as E, L and R.

Array E holds listing of the nodes visited in a DFS traversal of T and array L contains the tree-depth of each nodes stored in E. Size of both E and L is 2n – 1. Array R is of size n and contains the index of the first appearance of a node in array E. Together, these three arrays require O($n \log n$) bits. For nodes i and j, their lowest common ancestor is the node at the smallest tree depth that is visited between an occurrence of i and an occurrence of j in the Euler Tour. So,

$$\mathrm{lca(i, j)} \ = \ \mathrm{E}[RMQ_L(\mathrm{R[i], R[j]})]$$

- **RMQ on Array L:**

RMQ_L can be done in O(1) time with the help of preprocessing of the array L. For preprocessing, we have to divide the array L into $\frac{2n}{\log n}$ blocks where size of each block is $\frac{1}{2}\log n$. Then we have to form two arrays, B and C, both of size $\frac{2n}{\log n}$. B contains the minimum element from each of the blocks in A and C contains the locations of the minimum in each block of A. Suppose, $\frac{2n}{\log n} = b$. While calculating RMQ_L(i, j), there will be two possible cases:

<u>Case 1:</u> i and j are not in the same block
<u>Case 2:</u> i and j are in the same block

- <u>Handling Case 1:</u> We have to find location of the minimum element in the range from i to the end of the block and in the range from the beginning of the block containing j to index j. We will also need to consider the location of the minimum elements in the range of blocks completely contained between i and j.
  Then from all these minimum values of the blocks, we finally have to select the position of the smallest minimum block value. In order to do that, array B is preprocessed. We constructed $\lfloor logb \rfloor$ + 1 arrays B_0, B_1, …., $B_{\lfloor logb \rfloor}$ such that B_j[i] contains RMQA(i, i + 2^j), provided i + $2^j \leq$ n. Required time and

space for preprocessing will be $O(b\ logb)$ = $O\left(\frac{n}{\log n}\ log\frac{n}{\log n}\right) = O(n)$. So, number of bits required for preprocessing will be $O(n\ logn)$.

- <u>Handling Case 2:</u> To handle this case, we have to pre-compute and store corresponding RMQ for each pair (i, j) of indices that fall in a block. As consecutive entries in L differ by +1 or -1, preprocessing of the distinct blocks will take $\frac{1}{2}\sqrt{n}\ \times O(log^2 n) = O(n)$ time.

So, overall, query Q(i, j) will return LCP of S[i, n] and S[j, n] in O(1) time using preprocessing. The preprocessing will require linear time in total and uses no more than $O(n\ logn)$ bits.

#Question 4:

<u>Solution:</u>

Jaccard similarity of two sets measures how similar the two sets are. The Jaccard similarity of two sets x and y is, S(x, y) = $\frac{|\,x \cap y\,|}{|\,x \cup y\,|}$.

Jaccard distance is defined as 1 minus the Jaccard Similarity. So, Jaccard distance, D(x, y) = 1 – S(x, y).

Now, the **Locality-Sensitive Functions** take two items and makes a decision about whether these items should be a candidate pair. Let $d_1 < d_2$ be two distances according to some distance measure d. A family F of functions is said to be **(d_1, d_2, p_1, p_2)-sensitive** if for every f in F:

1. If d(x, y) ≤ d_1, then the probability that f(x) = f(y) is at least p_1.
2. If d(x, y) ≥ d_2, then the probability that f(x) = f(y) is at most p_2.

Family of minhash function is an example of sensitive function. This means, for any distance d_1 and d_2, minhash function family is (d_1, d_2, p_1, p_2)-sensitive if D(x, y) ≤ d_1 for the probability that f(x) = f(y) is at least p_1 and D(x, y) ≥ d_2 for the probability that f(x) = f(y) is at most p_2, where $0 \le d_1 < d_2 \le 1$.

Now, Jaccard similarity of sets x and y, $S(x, y) = 1 - D(x, y) \geq 1 - d_1$ [as $D(x, y) \leq d_1$]. Similarly we can show, $S(x, y) = 1 - D(x, y) \leq 1 - d_2$. But we know that the Jaccard similarity of x and y, $S(x, y)$, is equal to the probability that a minhash function will hash x and y to the same value. As $1 - d_1 \leq S(x, y) \leq 1 - d_2$, we can say $p_1 = 1 - d_1$ and $p_2 = 1 - d_2$.

So, the family of minhash functions is a $(d_1, d_2, 1\text{-}d_1, 1\text{-}d_2)$-sensitive family for any d_1 and d_2, where $0 \leq d_1 < d_2 \leq 1$.