

# **ECS 222A: Assignment #1**

Due on Tuesday, January 13, 2015

*Daniel Gusfield TR 4:40pm-6:00pm*

**Wenhao Wu**

## Contents

<b>Problem 1</b>	<b>3</b>
Problem 1(a) . . . . .	3
Problem 1(b) . . . . .	3
Problem 1(c) . . . . .	3
<b>Problem 2</b>	<b>4</b>
Problem 2(a) . . . . .	4
Problem 2(b) . . . . .	5
Problem 2(c) . . . . .	6
Problem 2(d) . . . . .	6
Problem 2(e) . . . . .	6
<b>Problem 3</b>	<b>6</b>
<b>Problem 4</b>	<b>7</b>
Problem 4(a) . . . . .	7
Problem 4(b) . . . . .	7
Problem 4(c) . . . . .	8
Problem 4(d) . . . . .	8

## Problem 1

Given two strings  $L$  and  $S$  that have an equal number of occurrences of each specific character, we define  $D(S_1, S_2)$  as the minimum number of transpositions of adjacent characters needed to convert  $S_1$  into  $S_2$ . For example,  $S_1 = CBADA$  can be converted into  $S_2 = ABDC A$  using exactly four transpositions. Notice that all the transpositions are done on  $S_1$ .

### Problem 1(a)

Assume that  $S_1$  and  $S_2$  each have exactly one occurrence of each character, for example  $S_1 = ACBD$  and  $S_2 = DCAB$ . Develop an efficient algorithm to compute the number  $D(S_1, S_2)$  given any input strings  $S_1$  and  $S_2$  that obey the stated assumption. Argue that your algorithm is correct (try to find a rigorous yet simple argument) and discuss how efficient it is in terms of the number of operations it does. (What are the primitive operations in your algorithm?)

**Answer:**

**Lemma 1.1**  $D(S_1, S_2)$  equals to the Kendall tau distance between  $S_1$  and  $S_2$ , i.e.

$$\begin{aligned} D(S_1, S_2) &= K(S_1, S_2) \\ &= |\{(i, j) | i < j, (\tau_1[i] < \tau_1[j] \text{ AND } \tau_2[i] > \tau_2[j]) \text{ OR } (\tau_1[i] > \tau_1[j] \text{ AND } \tau_2[i] < \tau_2[j])\}| \end{aligned}$$

where  $\tau_1[i]$  and  $\tau_2[i]$  denotes the index/ranking of character  $i$  in  $S_1$  and  $S_2$ , respectively.

**Proof** Firstly, each transpositions of adjacent characters in  $S_1$  can at most reduce  $K(S_1, S_2)$  by 1, and when  $S_1$  is eventually converted to  $S_2$  with transpositions  $K(S_1, S_2) = 0$ . As a result,  $K(S_1, S_2)$  is a lower bound of  $D(S_1, S_2)$ .

On the other hand, bubble sort uses exactly  $K(S_1, S_2)$  adjacent transpositions to convert  $S_1$  to  $S_2$ , therefore  $K(S_1, S_2)$  is achievable. Consequently,  $D(S_1, S_2) = K(S_1, S_2)$

A classical algorithm to compute  $K(S_1, S_2)$  is merge-sort, has  $\mathcal{O}(n \log n)$  complexity. Define index sequence  $Q$  such that

$$Q[k] = \tau_1[S_2[k]], k = 1, \dots, n$$

the constructing of which has  $\mathcal{O}(n)$  complexity. Then we can simply count the number of inverted pairs in  $Q$  using merge sort to compute  $D(S_1, S_2)$ .

### Problem 1(b)

Develop an efficient algorithm to actually transform  $S_1$  into  $S_2$  using exactly  $D(S_1, S_2)$  transpositions. Argue that your algorithm is correct and discuss how efficient it is in terms of the number of operations it does. The operations are the operations done by the algorithm, not the number of transpositions needed.

**Answer:** As stated in the proof to Lemma 1.1, we can bubble sort  $S_1$  based on the pairwise ranking defined by  $Q$ , which uses exactly  $D(S_1, S_2)$  transpositions to convert  $S_1$  to  $S_2$ , since each transposition reduces  $D(S_1, S_2)$  exactly by 1. It has  $\mathcal{O}(n^2)$  complexity.

### Problem 1(c)

Now explain how to handle the case of computing  $D(S_1, S_2)$  when those strings may have more than one occurrence of any character. Assume that both strings have the same number of occurrences of any particular character. Hint: One approach is to find a simple reduction of this problem to the previous one. Of course, give a proof of correctness and an analysis of the number of operations needed.

**Answer:**

**Lemma 1.2** *The transpositions that achieve  $D(S_1, S_2)$  does not change the order of multiple occurrences of a same character. In other words, assume there are  $n_i$  occurrences of character  $i$  in  $S_1$ . After  $S_1$  being converted to  $S_2$ , each of this occurrence appears in a new position denoted as  $c_1^{(i)}, c_2^{(i)}, \dots, c_{n_i}^{(i)}$ , where the subscripts denotes the order of their original appearance in  $S_1$ , then a series of  $D(S_1, S_2)$ -achieving transpositions must satisfy*

$$c_1^{(i)} < c_2^{(i)} < \dots < c_{n_i}^{(i)}$$

**Proof** If there exists  $c_l^{(i)} > c_{l+1}^{(i)}$ , then the  $l$ -th and the  $l + 1$ -th occurrence of character  $i$  must have been transposed somewhere, which is totally unnecessary. This is in contradict with the assumption of minimum number of transposition.

According to Lemma 1.2, we can redefine the index/ranking for each occurrence of each character. Denote  $\tau_1[i, m]$  as the index of the  $m$ -th occurrence of character  $i$  in  $S_1$ , then the index sequence  $Q$  is also redefined as

$$Q[k_2[i, m]] = \tau_1[i, m], k = 1, \dots, n$$

where  $k_2[i, m]$  is the index of the  $m$ -th occurrence of character  $i$  in  $S_2$ . Then counting the number of inverted pairs in  $Q$  using merge sort results in  $D(S_1, S_2)$ .

The algorithm to compute  $D(S_1, S_2)$  and actually transform  $S_1$  into  $S_2$  is implemented in `hw_1_1.py`.

## Problem 2

The following algorithmic problem arose in the field of Sociology. You are given two strings, for example  $L = ABCCBCD$  and  $S = AQCBA D$ . For each character  $X$  in the alphabet, define  $M(X)$  as the minimum number of times  $X$  appears in either  $L$  or  $S$ . For example,  $M(A) = 1$  and  $M(Q) = 0$  in the example above.

The problem requires us to remove characters from  $L$  and  $S$  so that for each character  $X$ , the number of remaining occurrence of  $X$  in each of  $L$  and  $S$  is exactly  $M(X)$ . So far there is no real problem since we know exactly how many of each character must be removed from each string.

However, for some character(s)  $X$  there may be choices for which specific occurrences of  $X$  should be removed. Hence there are choices for what the resulting strings (call them  $S_1$  and  $S_2$ ) will be.

Now comes the problem: Among all possible ways to choose the required removals, we want to create resulting strings  $S_1$  and  $S_2$  to minimize  $D(S_1, S_2)$ . We call this the *MinDistRem* problem. For example, with  $L$  and  $S$  above, we can create  $S_1 = ABCD$  and  $S_2 = CBAD$  with  $D(S_1, S_2) = 3$ , or we could create  $S_1 = S_2 = ACBD$  with  $D(S_1, S_2) = 0$ , and so we choose the latter as the solution to the *MinDistRem* problem.

### Problem 2(a)

Solve the MinDistRem problem for  $L = ACDQDCGFDERAE$  and  $S = EEC DACW ERGARF$ .

**Answer:** Denote  $H_L[X]$  and  $H_S[X]$  as the number of occurrences of character  $X$  in string  $L$  and  $S$ , respectively. We have We first note that “Q” must be removed from  $L$  and “W” must be removed from  $S$ , while “A”, “C”, “F” and “G” should not be removed. 2 “D” must be removed from “L”, 1 “E” must be removed from  $S$  and 1 “R” must be removed from  $S$ . We remove the characters that are surely to be

$X$	$H_L[X]$	$H_S[X]$	$M[X]$
A	2	2	2
C	2	2	2
D	3	1	1
E	2	3	2
F	1	1	1
G	1	1	1
Q	1	0	0
R	1	2	1
W	0	1	0

removed and highlight the characters that could be removed as

$$L = ACDDCGFDERAE$$

$$S = EECDACERGAF$$

There are a 3 ways of removing 2 “D”s from  $L$  and  $2 \times 2$  ways to remove a “E” and “R” from  $S$ . We list the  $D(S_1, S_2)$  for each possible pair of  $S_1$  and  $S_2$  in Table 1

Table 1: All possible  $D(S_1, S_2)$  for exhaustive search.

$S_2$	$S_1 = ACCGFDERAE$	$S_1 = ACD CGFERAE$
<i>ECDACEGARF</i>	18	15
<i>ECDACERGAF</i>	18	15
<i>EECDACGARF</i>	22	19
<i>EECDACRGAF</i>	22	19

Using an exhaustive search, we have  $S_1 = ACCGFDERAE$  and  $S_2 = ECDACERGAF$  or  $S_2 = ECDACEGARF$ .

## Problem 2(b)

We would like to find an efficient algorithm for the MinDistRem problem. The following algorithm was proposed:

1. Let  $L$  be the longer string and  $S$  the shorter string. If the two strings are the same length, choose  $L$  and  $S$  arbitrarily.
2. For string  $L$  make a list (called LIST, duh!) of the characters (but not their positions) that will be removed. For each such character  $X$  include on LIST a number of copies of  $X$  equal to the number of occurrences of  $X$  that must be removed.
3. Set pointer  $p$  to 1 and pointers  $q$  and  $q'$  to 0.
4. Let  $X$  be the character in position  $p$  of  $S$ .
5. Scan  $L$  for the first occurrence (if any) of character  $X$  from position  $p$  forward in  $L$ . If such an  $X$  is found, set  $q'$  to the position of that found  $X$ .

6. If such an  $X$  was found in  $L$ , scan the characters from  $q + 1$  to  $q' - 1$  (inclusive) for any characters on the LIST. As each such character is found (if any are) remove it from  $L$  and remove one copy of it from the LIST.
7. Set  $q$  to  $q'$ .
8. Increment  $p$  by one.
9. Repeat steps 5 through 8 until  $p$  is at the end of  $S$  or until LIST is empty.
10. If LIST is not empty, scan  $L$  from its right end to find occurrences of characters on LIST. As each such character is found (if any are), remove it from  $L$  and remove one copy of it from LIST.
11. If sequence  $L$  is now shorter than sequence  $S$ , exchange the labels  $L$  and  $S$ , and repeat steps 2 to 10 on these two current strings  $L$  and  $S$ . Else terminate.
12. At termination, one the remaining string  $L$  is called  $S_1$  and the remaining string  $S$  is called  $S_2$ .

**Answer:** This algorithm is implemented in *hw\_1\_2.py*.

### Problem 2(c)

Execute the above algorithm on  $L = ABCCBCD$  and  $S = ACBAD$  and then on  $L = ACDQDCGFDERAE$  and  $S = EECDACWERGARF$ .

**Answer:** On  $L = ABCCBCD$  and  $S = ACBAD$  the algorithm results in  $S_1 = S_2 = ACBD$ . On  $L = ACDQDCGFDERAE$  and  $S = EECDACWERGARF$  the algorithm results in  $S_1 = ECDACEGARF$  and  $S_2 = ACCGFDERAE$ .

### Problem 2(d)

Argue that the algorithm always terminates within two executions of steps 2 to 10.

**Answer:** Steps 2 to 10 removes characters from  $L$ . Since step 10 ensures all characters in LIST are removed, and from the definition of LIST in step 2, the occurrence of each character  $X$  in  $L$  is exactly  $M[X]$  now.

Since  $S$  has not been altered yet, if  $S$  is in the same length as  $L$ , then the occurrence of each character  $X$  in  $S$  is exactly  $M[X]$  now and there is nothing to remove from it, the algorithm terminates in one pass of step 2 to 10. Otherwise the by swapping  $L$  and  $S$  the second pass will ensure so. Consequently, the algorithm always terminates within two executions of steps 2 to 10.

### Problem 2(e)

Is it true that the algorithm always solves the MinDistRem problem? Justify.

**Answer:** Comparing the results from Problem 2(c) and Problem 2(a), we can see that this algorithm does not always solves the MinDist Rem problem.

## Problem 3

If you think the above algorithm does not solve the MinDistRem problem, can you propose an efficient algorithm that does always solve it? (this may be a very hard problem)

**Answer:** Too hard for me ...

## Problem 4

A “prototein” is a binary string that we embed on a two-dimensional grid. A legal embedding must satisfy the following rules:

1. Each character in the string gets placed on one point of the grid.
2. Each point of the grid gets at most one character of the string placed on it.
3. Two adjacent characters in the string must be placed on two points that are neighbors on the grid in either the horizontal or vertical direction, but not both. That is, two points across a diagonal are not neighbors. (Note that an interior point on the grid has four neighbors on the grid, and that an outside point has three neighbors, and that a corner point has only two neighbors).

Rules 1,2,3 mean that we are embedding the string onto the grid as a self-avoiding walk, without deforming the string. See the figure below. A contact is formed for every pair of 1's that are not adjacent in the string,

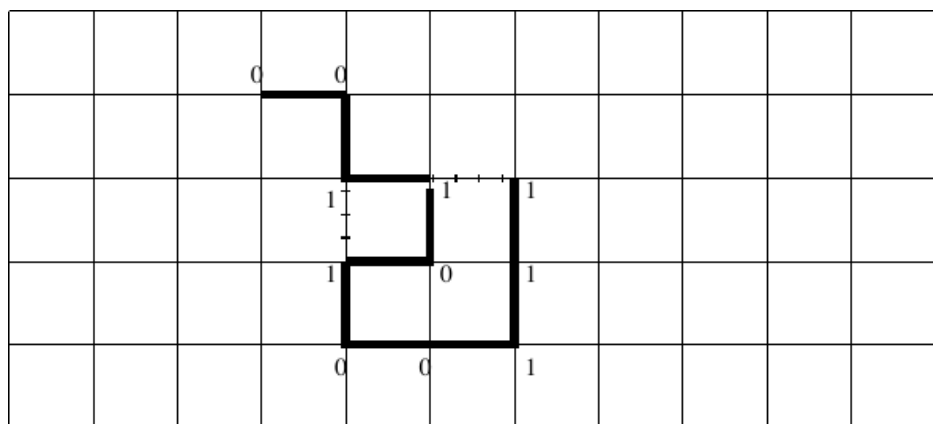


Figure 1: One possible embedding of the string 00110100111. The hatched lines show the contacts, two in this embedding. Can you find an embedding with more contacts?

but are placed on neighboring points in the grid. Such a pair of 1's is called a “contact” in the embedding.

Given an input string, the general problem is to find an embedding of the string on the grid so as to maximize the number of contacts.

Your Problems:

### Problem 4(a)

Prove (that is give a clear explanation for) the following claim: For any string, and any legal embedding of the string, the characters in positions  $i$  and  $j$  in the string can form a contact only if  $|i-j|$  is odd. Try playing with some examples first to convince yourself that this is true, and then try to find a concise way to prove it.

**Answer:**

### Problem 4(b)

For any given string  $S$ , let  $E(S)$  be the number of 1's in even positions in the string, and let  $O(S)$  be the number of 1's in odd positions in the string. Let  $C(S)$  be the minimum of  $E(S)$  and  $O(S)$ . Prove that the

number of contacts, in any legal embedding of  $S$ , cannot exceed  $2(C(S) + 1)$ .

*Answer:*

### Problem 4(c)

If we can invent the string  $S$ , as well as decide on how to embed it, we could get a lot of contacts, but that is cheating. Still if we can invent a string  $S$ , and embed it, so that the ratio of the number of contacts to the number of 1's in the string is high, that would be a good challenge. Find a string  $S$ , and an embedding of it, that has a ratio of contacts to 1's of  $7/6$ . Hint: you will need a string of length more than 25 (I think). So you will have to discover an idea, rather than just playing around.

*Answer:*

### Problem 4(d)

Prove that, over all possible strings and embeddings of those strings, the highest possible ratio of contacts to 1's is  $7/6$ . Hint: the handshake lemma from graph theory (remember graphs from cs100 or cs20?) is helpful here. The rest is just case analysis.

*Answer:*