# ECS 222A: Assignment #1

Due on Tusday, January 13, 2015

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

**Wenhao Wu**

# Contents

# 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 $S_2 = ABDCA$ 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.*

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

*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

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

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.

Problem 1 [Problem 1(c)] continued on next page... 

***Answer:***

**Lemma 1.2** *The transpositions that achieve $D(S_1, S_2)$ does not change the order of multiple occurences of a same character. In other words, assume there are $n_i$ occurences of character $i$ in $S_1$. After $S_1$ being converted to $S_2$, each of this occurence appears in a new position denoted as $c_1^{(i)}, c_2^{(i)}, \ldots, c_{n_i}^{(i)}$, 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*

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

**Proof** If there exists $c_l^{(i)} > c_{l+1}^{(i)}$, then the $l$-th and the $l+1$-th occurence 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 occurence of each character. Denote $\tau_1[i, m]$ as the index of the $m$-th occurence of character $i$ in $S_1$, then the index sequence $Q$ is also redefined as

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

where $k_2[i, m]$ is the index of the $m$-th occurence 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 $S = AQCBAD$. 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 $S = EECDACWERGARF$.

***Answer:*** Denote $H_L[X]$ and $H_S[X]$ as the number of occurences 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

$$L = ACDDCGFDERAE$$
$$S = EECDACERGARF$$

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.

| $S_2$ | $S_1 = ACCGFDERAE$ | $S_1 = ACDCGFERAE$ |
|---|---|---|
| $ECDACEGARF$ | 18 | 15 |
| $ECDACERGAF$ | 18 | 15 |
| $EECDACGARF$ | 22 | 19 |
| $EECDACRGAF$ | 22 | 19 |

Using an exhaustive search, we have $S_1 = ACCGFDERAE$ and $S_2 = ECDACERGAF$ or $S2 = ECDACEGARF$.

## 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 glorithm 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 occurence 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 occurence of each character $X$ in $S$ is exactly $M[X]$ now and there is nothing to remove from it, the algotithm 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 MinDistRem 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 . . . I just implemented a brute force solution to exhaust all possible $(S_1, S_2)$ and find the one with the minimum distance in ***hw_1_2.py***. Surely this is not considered "efficient".

# 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 adjacnet 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 neighbor 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,

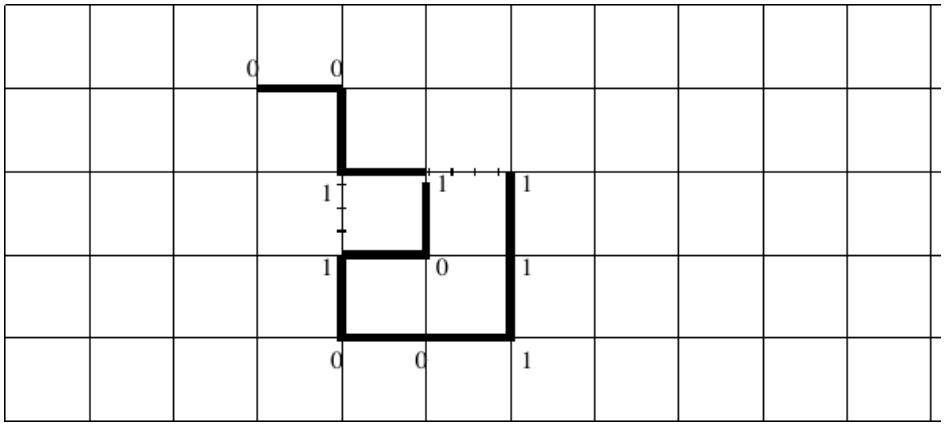


Figure 1: One possible embedding of the string 00110100111. The hatched lines show the contacts, two in this embedding. Can you find an embedding with more contacts?

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 $|ij|$ 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:*** We first color all grid points with 2 colors white and black so that the neighboring points have different color (as in a chess board).

Consider two points $a$ and $b$ on the grid, if they have the same color, then when $b$ moves to one of its neighboring points, they have different colors; if they have different colors, then when $b$ move to one of its

neighboring points, they have the same color. Consequently, two points of the same color must be separated by even steps and two points of different colors must be separated by odd steps. Here one step is defined as one move towards one of the neighboring points.

Due to the rule 3 the number of steps just defined equals to the distance between two characters in an embedded string. If characters on $i$ and $j$ forms a contact, their corresponding grid points must be neighbors so that they have different colors, therefore these two points must be separated by an odd number of steps. As a result $|i - j|$ is also odd.

## 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:*** According to problem 4(a), one character in a contact must be on an odd position and the other must be on an even position.

If $C(S) = O(S)$, we can count the number of contacts according to the 1's on the odd positions. For each of these 1's that is not at the beginning or end of the string, its corresponding grid points have at most 4 neighbors, but 2 of these 4 points must correspond to the 2 adjacent characters in the string so they can not form contact pairs. Consequently each 1 on the odd positions but not the beginning or end of the string can have at most 2 contacts. For the beginning 1 and the ending 1, a similar analysis applies and both of them have at most 3 contacts. As a result, the maximum number of contact is $2(C(S) + 1)$.

If $C(S) = E(S)$, we can count the number of contacts according to the 1's on the even positions. Similarly, each 1 that is not at the end of the string (it cannot be at the beginning!) can have at most 2 contacts. And the 1 at the end of the string (if any) can have at most 3 contacts. the maximum number of contact is $2C(S) + 1$.

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

## Problem 4(c)

If we can invent the string $S$, as well as decide on how to embed it, we could get alot 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:*** One simple example is $S = 1001000000010010000000 1001$, and the embedding is shown in Figure 2. This embedding has exactly 7 contacts and 6 1's, so its ratio of contacts to 1's is 7/6.

## 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:*** Consider only the 1's in the string. According to how many neighboring 1's they have in the embedding, they can be categorized as follows:

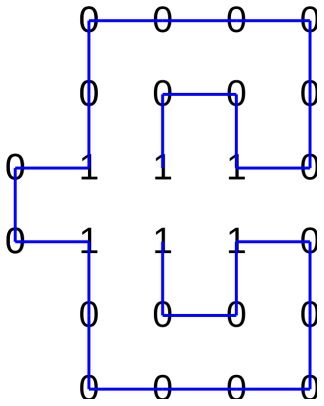


Figure 2: Embedding of $S = 1001000000010010000000 1001$.

1. The 1 with 4 neighboring 1's:
   - If this 1 is at the end or beginning of the string, it can form at most 3 contacts.
   - If this 1 is in the middle of the string, it can form at most 2 contacts.
2. The 1 with 3 neighboring 1's:
   - If this 1 is at the end or beginning of the string, it can form at most 3 contacts.
   - If this 1 is in the middle of the string, it can form at most 2 contacts.
3. The 1 with 2 neighboring 1's, it can form at most 2 contacts.
4. The 1 with 1 neighboring 1's, it can form at most 1 contacts.
5. The 1 with 0 neighboring 1's, it can form 0 contact.

According to the proof of the handshake lemma, we can count the total number of contacts as the summation of the number of contacts each 1 forms divided by 2. Then we sould like to prove that the average number of contacts each 1 can form is at most 7/3.

From the above categorization, we can see that if neither the end nor beginning of the string is a 1 with 3 neighboring 1's or 4 neighboring 1's, each 1 can form at most $2 < 7/3$ contacts.

If there is a 1 either at the begining or the end point of the string have 4 or 3 neighboring 1's, we would like to have as few 1's that is in the middle of the string as possible in order to possibly make the average number of contacts each 1 can form larger, since each of them can form at most 2 contacts. According to the number of neighboring 1's that the beginning and ending 1's have, we can count the minimum number of middle 1's in the string (a very loss lower bound though) and therefore the corresponding ratio of contacts to 1's. Note that if this ratio is smaller than 7/6, then having any more number of middle points will also result in a ratio smaller than 7/6, because each middle point account for at most 1 to this ratio. The results are shown in Table 2. Some examples of each cases are plotted in Figure 3 and Figure 4.

From the table we can see that the maximum raio of contacts to 1's is 7/6.

Table 2: Enumeration of cases where the string has the minimum number of 1's in the middle.

| case | beginning of the string | end of the string | min. # 1's in the middle | ratio of contacts to 1's |
|---|---|---|---|---|
| 1 | 1 with 4 neighbor 1's | 1 with 4 neighbor 1's | 6 | $\frac{6\cdot2+2\cdot3}{2\cdot(6+2)}=\frac{9}{8}$ |
| 2 | 1 with 4 neighbor 1's | 1 with 3 neighbor 1's | 5 | $\frac{5\cdot2+2\cdot3}{2\cdot(5+2)}=\frac{8}{7}$ |
| 3 | 1 with 4 neighbor 1's | 1 with 2 neighbor 1's | 4 | $\frac{4\cdot2+3+2}{2\cdot(4+2)}=\frac{13}{12}$ |
| 4 | 1 with 4 neighbor 1's | 1 with 1 neighbor 1's | 3 | $\frac{3\cdot2+3+1}{2\cdot(3+2)}=1$ |
| 5 | 1 with 4 neighbor 1's | 1 with 0 neighbor 1's | 4 | $\frac{4\cdot2+3+0}{2\cdot(4+2)}=\frac{11}{12}$ |
| 6 | 1 with 4 neighbor 1's | 0 | 4 | $\frac{4\cdot2+1\cdot3}{2\cdot(4+1)}=\frac{11}{10}$ |
| 7 | 1 with 3 neighbor 1's | 1 with 3 neighbor 1's | 4 | $\frac{4\cdot2+2\cdot3}{2\cdot(4+2)}=\frac{7}{6}$ |
| 8 | 1 with 3 neighbor 1's | 1 with 2 neighbor 1's | 3 | $\frac{3\cdot2+3+2}{2\cdot(3+2)}=\frac{11}{10}$ |
| 9 | 1 with 3 neighbor 1's | 1 with 1 neighbor 1's | 2 | $\frac{2\cdot2+3+1}{2\cdot(2+2)}=1$ |
| 10 | 1 with 3 neighbor 1's | 1 with 0 neighbor 1's | 3 | $\frac{3\cdot2+3+0}{2\cdot(3+2)}=\frac{9}{10}$ |
| 11 | 1 with 3 neighbor 1's | 0 | 3 | $\frac{3\cdot2+1\cdot3}{2\cdot(3+1)}=\frac{9}{8}$ |

# Appendices

1. ***hw_1_1.py***

```
#! /usr/bin/env python3
import sys
INF = sys.maxsize

S1 = "CBADA"
S2 = "ABDCA"

def get_hist(S):
    """
    Function used to count the number of appearances of each character in
    ↪ the string S, return a dictionary with characters as keys and count
    ↪ as values.
    """
    hist = dict();
    for c in S:
        hist[c] = hist.get(c, 0) + 1
    return hist

class UnequalOccurence(Exception):
    pass

def count_inversion(Q, start, end):
    """
    Basically merge sort plus counting inversion across the left and the
    ↪ right segment.
    """
    if end - start > 1:
```

```
        mid = int((start + end) / 2)
        countInvLeft = count_inversion(Q, start, mid)
        countInvRight = count_inversion(Q, mid, end)
        QLeft, QRight = (Q[start : mid], Q[mid : end])
        idxLeft, idxRight = (0, 0)
        countInvCross = 0
        QLeft, QRight = (QLeft + [INF], QRight + [INF])
        for idx in range(start, end):
            if QLeft[idxLeft] < QRight[idxRight]:
                Q[idx] = QLeft[idxLeft]
                idxLeft += 1
                countInvCross += idxRight
            else:
                Q[idx] = QRight[idxRight]
                idxRight += 1
        return countInvLeft + countInvRight + countInvCross
    else:
        return 0

def get_kendall_tau_dist(str1, str2):
    """
    get kendall tau distance between str1 and str2
    """
    h = get_hist(str1)
    n = len(str1)
    if h != get_hist(str2):
        raise UnequalOccurence("Not every character has the same number of
            ↪ occurence in both strings!")

    # Construct index sequence Q
    tau2_dict = dict()
    for idx, c in enumerate(str2):
        tau2_dict.setdefault(c, list()).append(idx)
    Q = [0] * n
    for idx, c in enumerate(str1):
        Q[idx] = tau2_dict[c].pop(0)
    Q0 = Q[:]
    #print(Q)

    # Count the inversion in Q
    return (count_inversion(Q, 0, len(Q)), Q0)

def transform_by_transpose(str1, Q):
    """
    Transform str1 to str2 according to index sequence Q computed by
    ↪ get_kendall_tau_dist() function
    """
    n = len(str1)
    assert n == len(Q), "str1 and Q must have the same length!"
```

```
    count_transpose = 0
    print("{0}: {1}".format(count_transpose, tuple(str1)))
    str_zipped = list(zip(list(str1), Q))
    for i in range(n - 1, -1, -1):
        for j in range(0, i):
            if str_zipped[j][1] > str_zipped[j + 1][1]:
                str_zipped[j + 1], str_zipped[j] = (str_zipped[j],
                    ↪ str_zipped[j + 1])
                count_transpose += 1
                print("{0}: {1}".format(count_transpose, list(zip(*
                    ↪ str_zipped))[0]))

    str_sorted = list(zip(str_zipped))
    return str(str_sorted[0])

if __name__ == "__main__":
    print("S1 is: {0}".format(S1))
    print("S2 is: {0}".format(S2))
    (D, Q) = get_kendall_tau_dist(S1, S2)
    print("Q is: {0}".format(Q))
    print("D(S1, S2) is: {0}".format(D))
    transform_by_transpose(S1, Q)
```

2. ***hw_1_2.py***

```
#! /usr/bin/env python3
import hw_1_1
import itertools

def get_LIST(L, S):
    """
    Function that computes M and return a LIST of the characters must be
    ↪ removed from L
    """

    occurence_L, occurence_S = (hw_1_1.get_hist(L), hw_1_1.get_hist(S))
    chars = set(occurence_L.keys()).union(set(occurence_S.keys()))
    M = {c: min([occurence_L.get(c, 0), occurence_S.get(c, 0)]) for c in
        ↪ chars}
    LIST = []
    for c in set(L):
        n_remove = occurence_L.get(c, 0) - M[c]
        if n_remove > 0:
            LIST.extend([c] * n_remove)
    return (M, LIST)


def min_dist_rem(L, S):
    """
```

```
    Functions to solve the MinDistRem problem and return a tuple of the 2
    ↪ resulting strings.
    """

    if len(L) < len(S):
        L, S = (S, L)
    L = remove_from_L(L, S)

    if len(L) < len(S):
        L, S = (S, L)
        L = remove_from_L(L, S)

    return (L, S)

def remove_from_L(L, S):
    """
    Functions representing a pass of step 2 to 10
    """
    M, LIST = get_LIST(L, S)

    L_list, S_list = (list(L), list(S))
    p, q, q_prime = (0, -1, -1)
    while p < len(S) and len(LIST) > 0:
        X = S[p]
        try:
            q_prime = L_list.index(X, p)
        except ValueError:
            pass
        else:
            for idx in range(q + 1, q_prime):
                if L_list[idx] in LIST:
                    LIST.remove(L_list[idx])
                    L_list[idx] = None
            q = q_prime
        finally:
            p += 1
    if len(LIST) > 0:
        for idx in range(len(L_list), -1, -1):
            if L_list[idx] in LIST:
                LIST.remove(L_list[idx])
                L_list[idx] = None

    return ''.join([c for c in L_list if c is not None])

def min_dist_rem_BF(L, S):
    """
    Brute force to solve the minDistRem problem
    """
    S1s, S2s = get_all_S12(L, S)
```

```
    D = {(S1, S2): hw_1_1.get_kendall_tau_dist(S1, S2)[0] for S1 in S1s
        ↪ for S2 in S2s}
    #print(D)
    D_min = min(D.values())
    R = [key for key, value in D.items() if value == D_min]
    return R

def get_all_S12(L, S):
    """
    Return a tuple of all possible S1 and a tuple of all possible S2
    """

    chars_to_remove_L, chars_to_remove_S = get_char_to_remove(L, S)

    wheres_to_remove_L = itertools.product(*(itertools.combinations(
        ↪ chars_to_remove_L[c][1], chars_to_remove_L[c][0]) for c in
        ↪ chars_to_remove_L.keys()))
    S1s = set()
    for where_to_remove_L in wheres_to_remove_L:
        idxs_to_remove = [idx for idxs in where_to_remove_L for idx in
            ↪ idxs]
        S1s.add("".join([L[idx] for idx in range(len(L)) if idx not in
            ↪ idxs_to_remove]))

    wheres_to_remove_S = itertools.product(*(itertools.combinations(
        ↪ chars_to_remove_S[c][1], chars_to_remove_S[c][0]) for c in
        ↪ chars_to_remove_S.keys()))
    S2s = set()
    for where_to_remove_S in wheres_to_remove_S:
        idxs_to_remove = [idx for idxs in where_to_remove_S for idx in
            ↪ idxs]
        S2s.add("".join([S[idx] for idx in range(len(S)) if idx not in
            ↪ idxs_to_remove]))

    return S1s, S2s


def get_char_to_remove(L, S):
    """
    Get 2 dict of characters, numbers to remove and possible possitions to
        ↪ remove for L and S respectively
    """

    occurence_L, occurence_S = (hw_1_1.get_hist(L), hw_1_1.get_hist(S))
    chars = set(occurence_L.keys()).union(set(occurence_S.keys()))
    M = {c: min([occurence_L.get(c, 0), occurence_S.get(c, 0)]) for c in
        ↪ chars}

    chars_to_remove_L = dict()
```

```
    for c in set(L):
        n_remove = occurence_L.get(c, 0) - M[c]
        if n_remove > 0:
            chars_to_remove_L[c] = (n_remove, find(L, c))

    chars_to_remove_S = dict()
    for c in set(S):
        n_remove = occurence_S.get(c, 0) - M[c]
        if n_remove > 0:
            chars_to_remove_S[c] = (n_remove, find(S, c))

    return (chars_to_remove_L, chars_to_remove_S)

def find(s, c):
    """
    Generalization of the index function: find all indices of occurences
    ↪ of c in string s
    """
    return [i for i, ltr in enumerate(s) if ltr == c]

if __name__ == "__main__":
    for (L, S) in [("ABCCBCD", "ACBAD"), ("ACDQDCGFDERAE", "EECDACWERGARF"
    ↪ )]:
        print("L = {L}\nS = {S}".format(**locals()))
        print("(S1, S2) = {0}".format(min_dist_rem(L, S)))

    L = "ACDQDCGFDERAE"
    S = "EECDACWERGARF"

    R_BF = min_dist_rem_BF(L, S)
    R = min_dist_rem(L, S)

    print("By the proposed algorithm: (S1, S2) = {0}, D = {1}".format(R,
    ↪ hw_1_1.get_kendall_tau_dist(*R)[0]))
    print("By exhaustive search: (S1, S2) = {0}, D = {1}".format(R_BF,
    ↪ hw_1_1.get_kendall_tau_dist(*(R_BF[0]))[0]))
```

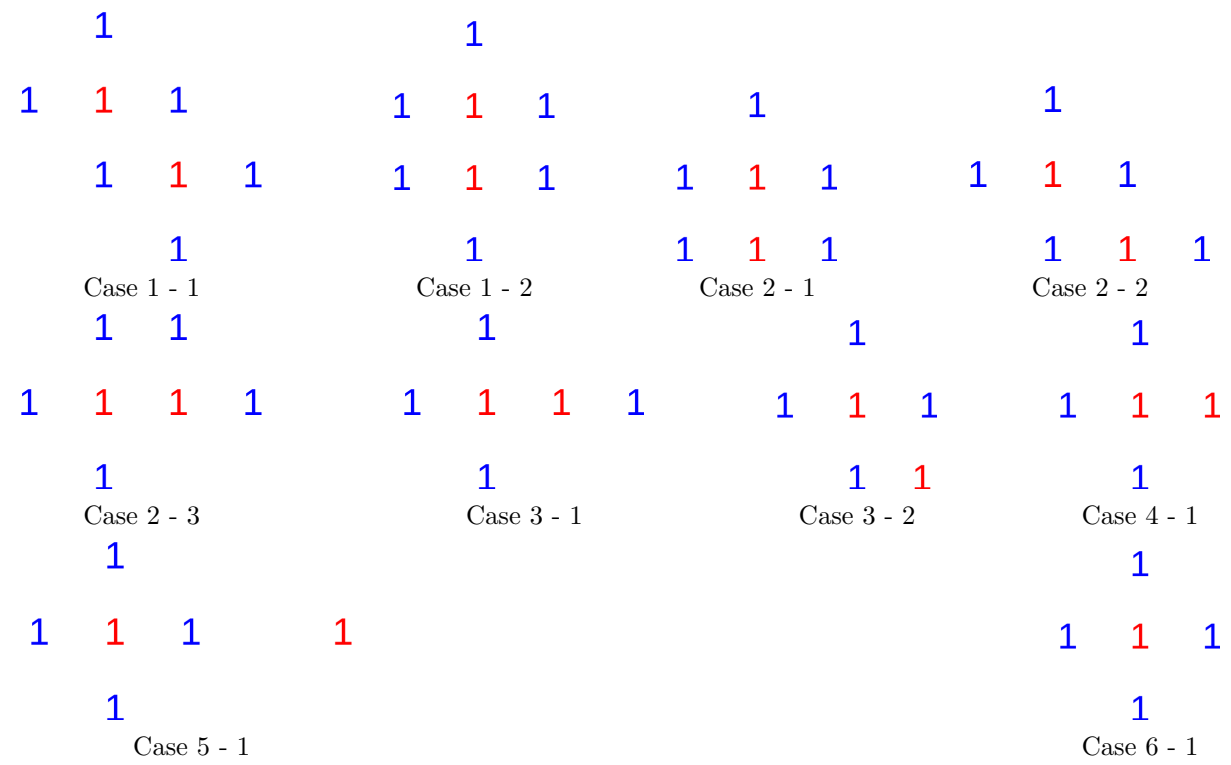


Figure 3: Different cases where the string has the minimum number of 1s in the middle. Red represents beginning and ending 1's and blue represents middle 1's. Symmetrical cases are not plotted.

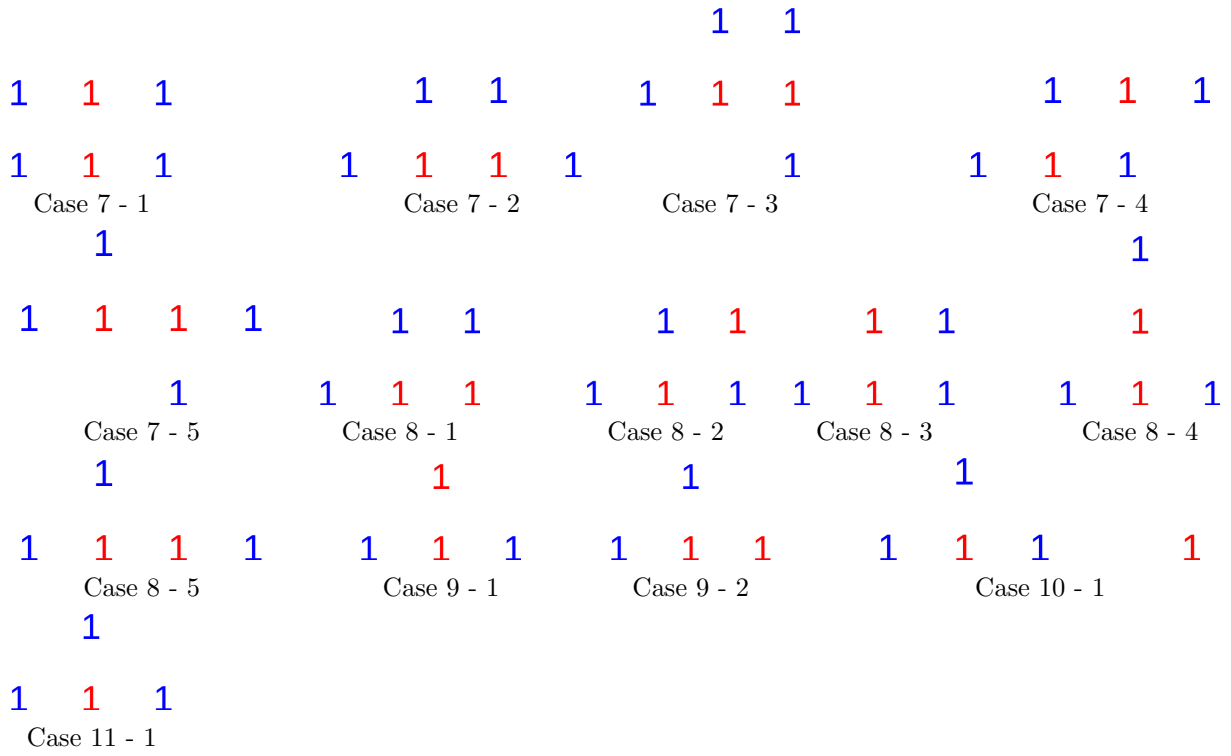


Figure 4: Different cases where the string has the minimum number of 1s in the middle (continued).