Prefix Transpositions on Binary and Ternary Strings

Amit Kumar Dutta[1], **Masud Hasan**[2], **and M. Sohel Rahman**[2]
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh
[1]amit@csebuet.org
[2]{masudhasan,msrahman}@cse.buet.ac.bd

Abstract—*The problem sorting by Prefix Transpositions asks for the minimum number of prefix transpositions required to sort the elements of a given permutation. In this paper, we study a variant of this problem where the prefix transpositions act not on permutations but on strings over a fixed size alphabet. Here, we determine the minimum number of prefix transpositions required to sort the binary and ternary strings, with polynomial time algorithms for these sorting problems.*

Keywords: strings, sorting, permutation, genome comparison, prefix transposition

1. Introduction

The *transposition distance* between two permutations (and the related problem of *sorting by transposition*) is used to estimate the number of global mutations between genomes and can be used by molecular biologists to infer evolutionary and functional relationships. A transposition involves swapping two adjacent substrings of the permutation. In a prefix transposition, one of them must be a prefix. *Sorting by prefix transposition* is the problem of finding the minimum number of prefix transpositions needed to transform a given permutation into the identity permutation. In the literature, other interesting problems include sorting by other operations like reversals, prefix reversals, block interchange etc.

A natural variant of the aforementioned sorting problems is to consider them not on permutations but on strings over fixed size alphabets. This shift is inspired by the biological observation that multiple “copies" of the same gene can appear at various places along the genome [3].

Indeed, recent works by Christie and Irving [1], Radcliffe et al. [2] and Hurkens et al. [3] explore the consequences of switching from permutations to strings. Notably, such rearrangement operations on the strings have been found to be interesting and important in the study of orthologous gene assignment [4], especially if the strings have only low level of symbol repetition.

The work of Chen et al. [4], presented for both reversals and transpositions, polynomial-time algorithms for computing the minimum number of operations to sort a given binary string. They also gave exact constructive diameter results on binary strings. Radcliff et al. [2] on the other hand gave refined and generalized reversal diameter results for non fixed size alphabets. Additionally, they gave a polynomial-time algorithm for optimally sorting a ternary (3 letter alphabet) string with reversals. Finally, Hurkens et al. [3] introduced *grouping* (a weaker form of sorting), where identical symbols need only be grouped together, while a group can be in any order. In the sequel, they gave a complete characterization of the minimum number of prefix reversals required to group (and sort) binary and ternary strings.

In this paper, we follow up the work of [3] and consider prefix transposition to group and sort binary and ternary strings. Notably, as a future work in [3], the authors raised the issue of considering other genome arrangement operators. In particular, here, we find the minimum number of prefix transpositions required to group and sort binary or ternary strings. It may be noted that, apart from being an useful aid for sorting, grouping itself is a problem of interest in its own right [5].

The rest of the paper is organized as follows. In Section 2, we discuss the preliminary concepts and discuss some notations we use. Section 3 is devoted to grouping, where we present and prove the corresponding bounds. Then, in Section 4, we extend the results of grouping to get the corresponding bounds for sorting. In Section 5, we very briefly discuss about the higher-arity alphabets in the context of our results. Finally, we briefly conclude in Section 6.

2. Preliminaries

We follow the notations and definitions used in [3], which are briefly reviewed below for the sake of completeness. We use $[k]$ to denote the first k non-negative integers $\{0, 1, \ldots, k-1\}$. A k-ary string is a string over the alphabet $\Sigma = [k]$. Moreover, a string $s = s_1 s_2 \ldots s_n$ of length n is said to be *fully k-ary*, or to have *arity* k, if the set of symbols occurring in it is $[k]$.

A prefix transposition $f(1, x, y)$, where $1 < x < y \leq (n + 1)$, is an rearrangement event that transforms s into $f\{s\} = [s[x] \ldots s[y-1] s[1] \ldots s[x-1] s[y] \ldots s[n]]$. The prefix transposition distance $d_s(s)$ of a string s is defined as the number of prefix transpositions required to sort the string. Note that, after a transposition operation is performed, the two adjacent symbols of the corresponding string may be identical. We consider two strings to be equivalent if one can be transformed into the other by repeatedly duplicating (by transposing) symbols and eliminating adjacent identical symbols. This elimination of adjacent identical symbols gives us a *reduced string*, i.e., a string of reduced length and this process is referred to as *reduction*. As representatives of the *equivalence classes* we take the shortest string of each class. Clearly, these are the strings where adjacent symbols always differ. The process of transforming a string into the representative string of its equivalence class is sometimes referred to as *normalization*. Therefore, the process of normalization basically comprises of repeated transposition and reduction.

As an example, let $s = baaabbabaabaaab$ and we want to apply the operation $f(1, 3, 7)$. Now, $s[x] \dots s[y-1] = s[3] \dots s[6] = aabb$, $s[1] \dots s[x-1] = s[1] \dots s[2] = ba$, $s[y] \dots s[n] = s[7] \dots s[15] = abaabaaab$. Therefore, after applying the operation, we get, $s = s[3] \dots s[6] s[1] \dots s[2] s[7] \dots s[15] = \overline{aabb}\underline{ba}abaabaaab = aab$ **bbaa** $baabaaab = aab$ **ba** $baabaaab = aabbabaabaaab$.

A reduction that decreases the string length by x is called an x transposition. So, if $x = 0$, then we have a 0 transposition. The above example illustrates a 2 transposition.

3. Grouping

The task of sorting a string can be divided into two subproblems, namely, *grouping* the identical symbols together and then putting the groups of identical symbols in the right order. The grouping distance $d_g(s)$ of a fully k ary string s is defined as the minimum number of prefix transposition required to reduce the string to one of length k.

3.1 Grouping Binary Strings

As strings are normalized, only 2 kinds of binary strings are possible, namely, $010101 \dots 010$ and $101010 \dots 101$. The grouping of binary strings seems to be quite easy and obvious. The following bound is easily achieved.

Theorem 1: (Bound for Binary strings) Let s be a fully binary string. Then, $d_g(s) = \lceil \frac{n-k}{2} \rceil$.

Proof. We can always have a 2 transposition if $|s|$ is even. However, if $|s|$ is odd, we need an extra 1 transposition. So, the upper bound is $d_g(s) = \lceil \frac{n-k}{2} \rceil$.

We illustrate the above result with the help of an example. Let $s = ababab$. Then we can continue as follows: $s = ababab = \overline{ab}\underline{a}bab =$ **aabb**$ab = abab = \overline{ab}\underline{a}b =$ **aabb** $= ab$. Here, we need two 2 transpositions to group this string. So, $d_g(s) = 2$.

3.2 Grouping Ternary Strings

In this section, we focus on ternary strings. As it seems, grouping ternary strings is not as easy as grouping binary strings. We start with the following theorem.

Lemma 1: In a fully ternary string, we can always give a 1 transposition.

Proof. We take a ternary string s of length $n > 3$. Now, we take a prefix a of length k. Now if $a[1]$ occurs at the suffix at position i, we can transpose $a[1] \dots a[i-1]$ with $a[i]$. Then, a[1] and a[i] are adjacent and we can eliminate one of the two. Otherwise, if $a[k]$ occurs at the suffix at position i, then we can transpose $a[1] \dots a[k]$ with $a[k+1] \dots a[i-1]$. Then $a[k]$ and $a[i]$ are adjacent and as before, we can eliminate one of them. Since, one of the above cases always occurs for ternary strings, the result follows.

3.3 Grouping distance for Ternary strings

The lower bound for the *grouping* of a ternary string remains the same as that of binary strings; but, as can be seen from Theorem 2 below, the upper bound differs. We first give an easy but useful lemma.

Lemma 2: Suppose $s[1..n]$ is a fully ternary string. If we have a prefix $s[1..i], 1 < i \leq n-2$ such that $s[1] = s[n-1]$ and $s[i] = s[n]$, then we have a 2 transposition.

The proof of Lemma 2 is very easy and hence is omitted.

Theorem 2: (Bound for Ternary strings) Let s be a fully ternary string. Then, $\lceil \frac{n-k}{2} \rceil \leq d_g(s) \leq \lceil \frac{n-k}{2} \rceil + 1$ where n is the length of the string and k is the arity.

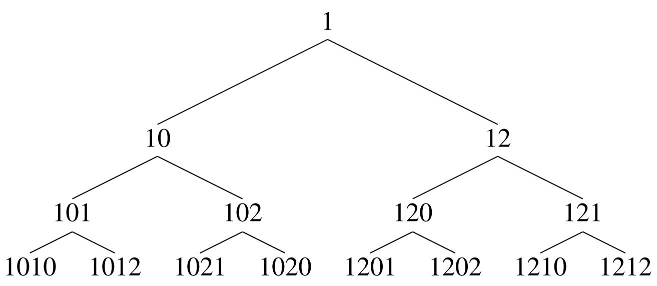
Proof. First we prove the lower bound. Here, $k = 3$. If we can always give a 2 transposition, then at each operation the string length is decremented by 2 and if $|s|$ is odd, we need an extra 1 transposition. Hence, we have $\lceil \frac{n-k}{2} \rceil \leq d_g(s)$.

Let us concentrate on the upper bound. As strings are fully ternary, we don't need to work with $n \leq 3$. Now, if we apply the upper bound for $n = 4, 5$ and 6, we have $d_g(s) = 2, 2$ and 3 respectively. It is easy to realize that, by Lemma 1, we can always satisfy the above upper bound. Thus the upper bound is proved for $n < 7$.

Now we consider $n \geq 7$. In what follows, we only consider strings starting with 1. This doesn't lose the generality since we can always use relabeling for strings starting with 0 or 2. Now, note carefully that for any string starting with 1, we can only have one of the following eight prefixes of length 4 :

$$1012, 1010, 1021, 1020, 1201, 1202, 1210 \text{ and } 1212. \quad (1)$$

Here we give the tree diagram of all strings starting with 1:



Now note that, the upper bound of Theorem 2 tells us that, we can give at most three 1 transpositions when n is even (i.e. $n-k$ is odd) and two 1 transpositions when n is odd (i.e. $n-k$ is even)[1]. For a n length string, if we can give a 2 transposition, the resulted reduced string may start with 1,0 or 2. For the latter two cases, i.e., 0 or 2, we can use relabeling as is mentioned before. Therefore we can safely state that the reduced string will have any of the 8 prefixes of List 1. Hence, it suffices to prove the bound considering each of the prefixes of List 1. We will now follow the following strategy:

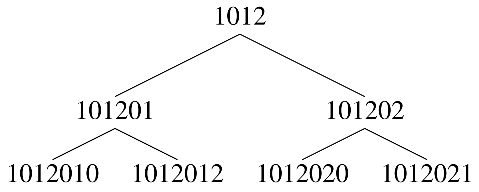
> We will take each of the prefixes of List 1 and expand it (by adding symbols) to construct all possible strings of length greater than or equal to 7. Strictly speaking, we will not consider all possible strings; rather we will continue to expand until we get a 2 transposition, since afterwards, any further expansion would also guaranty a 2-transposition. Suppose s is one such string. We will take s and try to give a 2 transposition with any of its prefix. If we succeed, then, clearly, we are moving towards the best case and we only need to work with the reduced string. If we can't give a 2 transposition, we specifically deal with s and show that the bound holds. Now if we can give a 2 transposition, the reduced string will have any of the 8 prefixes (using relabeling if needed) and we will show that all strings of these cases will follow the bound.

Firstly it is easy to note that, the prefixes 1010 and 1212 themselves have 2-transpositions (Lemma 2). Therefore, we can safely exclude them from the following discussion. In what follows, when we refer to the prefixes of List 1, we would actually mean all the prefixes excluding 1010 and 1212. Now, to expand the prefixes, if we add 10 or 12, all of them would be able to give a 2 transposition (Lemma 2). Therefore, in what follows, we consider the other cases. Now we analyze each of the prefixes below.

1012

If we add 01, we can only add[2] 0 or 2. The resulting expanded string becomes 1012010 or 1012012. In both cases, we can give a 2 transposition. On the other hand, if we add 02, we can add 0 or 1 next and the string becomes 1012020 or 1012021. The string $10120\underline{20}$ satisfies the bound as follows: $\overline{1}\underline{01}2020 \Rightarrow 012020 \Rightarrow \overline{012}\underline{0}20 \Rightarrow 0120 \Rightarrow \overline{0}\underline{12}0 \Rightarrow 120$. Here, $n=7$ and we need only 3 transpositions holding the bound true. Now if we add 1, the string becomes 10120201. For this one as well the bound holds as follows: $\overline{1}\underline{01}20201 \Rightarrow 0120201 \Rightarrow \overline{01}\underline{2020}1 \Rightarrow 20201 \Rightarrow \overline{20}\underline{2}01 \Rightarrow 201$. Here, n =8 and we needed 3 transpositions. Now, it can be easily checked that strings like $1012020(20)^*$ or $1012020(20)^*1$ the bound holds using the same strategy as shown above. Adding anything with $1012020(20)^*1$ will also give a 2 transposition as follows. Clearly, we first need to add either 0 or 2 and immediately Lemma 2 would apply.

Now we consider 1012021. The bound holds for this one as well as follows: $\overline{1}\underline{01}2021 \Rightarrow 012020 \Rightarrow \overline{012}\underline{0}21 \Rightarrow 0121 \Rightarrow \overline{01}\underline{2}1 \Rightarrow 201$. Now, strings like $1012(02)^*1$ can also be handled similarly hence the bound holds for them as well. Adding anything with $1012(02)^*1$ will also give a 2 transposition (Lemma 2). Figure 2 shows the tree diagram. For the other cases, tree diagram is almost similar and hence we will not include them in other cases.



1021

This prefix is ending with 1 and adding anything will give a 2 transposition (Lemma 2).

1020

We first add 21 with this prefix. Then, adding anything with 102021 will give a 2 transposition (Lemma 2). Now, for $102(02)^+1$, we need a 1 transposition to move the initial 1 at the end. Now the upper bound holds because the rest of the string is binary (Theorem 1). Also, adding anything with $102(02)^+1$ will give a 2 transposition (Lemma 2).

Now, if we add 20 with the prefix we get 102020. Next we add 1 or 2 and get 1020201 or 1020202 respectively. For 1020201, the bound holds as follows: $\overline{1}\underline{02020}1 \Rightarrow 020201 \Rightarrow \overline{02}\underline{0}201 \Rightarrow 0201 \Rightarrow \overline{0}\underline{2}01 \Rightarrow 201$. All strings like $1020(20)^+1$ can also be handled similarly and hence the bound holds for them as well. And adding anything with $1020(20)^+1$ will give a 2 transposition (Lemma 2).

Now for 1020202, the bound holds as follows: $\overline{10}\underline{2}0202 \Rightarrow 210202 \Rightarrow \overline{210}\underline{2}02 \Rightarrow 2102 \Rightarrow \overline{2}\underline{10}2 \Rightarrow 102$. Now, string like $10202(02)^+$ can be handled similarly and hence the bound holds. For $10202(02)^+1$, we need a 1 transposition to move the initial 1 at the end. Now the upper bound holds because the rest of the string is binary (Theorem 1).

[1]If we could give a 2 transposition at each step, we would get the bound of $\lceil \frac{n-k}{2} \rceil$.

[2]By 'add' we mean append.

Also adding anything with $10202(02)^{+}1$ will give a 2 transposition (Lemma 2).

1201

This prefix is ending with a 1 and adding anything will give a 2 transposition (Lemma 2).

1202

Here, we can employ relabeling and map 2 to 0 and 0 to 2 to get 1020. Now recall that we have already considered 1020 before and hence we are done.

1210

Here we can again employ relabeling and map 2 to 0 and 0 to 2 to get 1012. And since we have already considered 1012, we are done.

And this completes our proof.

3.4 Sorting Binary Strings

Theorem 3: (Bound for Binary strings) Let s be a fully binary string. Then, $d_s(s) \leq \lceil \frac{n-k}{2} \rceil$.

Proof. As binary strings have only 2 letters, after grouping they are already sorted (in ascending or descending order). So, the upper bound is $d_s(s) \leq \lceil \frac{n-k}{2} \rceil$.

4. Sorting

The sorting distance $d_s(s)$ of a fully k ary string s is defined as the minimum number of prefix transposition required to sort the string to one of length k. We again consider normalized strings.

4.1 Sorting Ternary Strings

Theorem 4: (Bound for Ternary strings) Let s be a fully Ternary string. Then, upper bound for sorting ternary string is $d_g(s) \leq \lceil \frac{n-k}{2} + 2 \rceil$.

Proof. After grouping a ternary string, we have the following grouped strings: $012, 021, 102, 120, 201$ and 210. Among these, 012 and 210 are already sorted. We need one more 0 transposition to sort $021, 102, 120$ and 201. Hence the result follows.

5. Higher-arity Alphabets

So far we have only focused on binary and ternary strings. Clearly, it would be interesting to explore higher-arity strings. And, in fact, we are currently working on that direction, although, complete characterizations for higher alphabets have so far eluded us. However, the following lemma is easy:

Lemma 3: In a fully string of any arity, we can always give a 1 transposition.

Proof. It is easy to say that Theorem 1 will also work for fully strings of arbitary arity. If a string has arity k, it is fully and it's length is greater than k, at least one symbol must be repeated and hence we can always give a 1 transposition.

From Lemma 3, for strings of length n, we can easily state that at most $n-k$ prefix transpositions are required (i.e. we are giving 1 transposition at each step) for grouping. But certainly, some 2 transpositions will be available. However, the question of getting any tight upper or lower bound still remains unsettled.

Note that, as was argued in [3], the shift from permutations to strings alters the problem universe somewhat. With permutations, for example, the distance problem between two permutations π_1 and π_2 for prefix transpositions (or any other genome rearrangement operations), is equivalent to sorting, because the symbols can simply be relabeled to make either permutation equal to the identity permutation. However for strings when symbol repetition is allowed, such a relabeling is not possible. Also, it seems that as the alphabet size increases, the difficulty level also increases.

6. Conclusion

In this paper, we have discussed grouping and sorting of fully binary and ternary strings when the allowed operation is prefix transposition. Following the work of [3], we have handled grouping by prefix transpositions of binary and ternary strings first and extended the results for sorting. In particular we have proved that, for binary strings the grouping distance $d_g(s) = \lceil \frac{n-k}{2} \rceil$ and for ternary string we have $\lceil \frac{n-k}{2} \rceil \leq d_g(s) \leq \lceil \frac{n-k}{2} \rceil + 1$, where n is the length of the string and k is the arity. On the other hand, for sorting binary and ternary strings the sorting distance $d_s(s)$ is upper bounded by $\lceil \frac{n-k}{2} \rceil$ and $\lceil \frac{n-k}{2} \rceil + 2$ respectively. As has already been mentioned, we are now considering the higher-arity alphabets as an extension of our work.

Acknowledgement

This research work was done as part of the undergraduate thesis work of Amit KUmar Dutta at CSE, BUET.

References

[1] David A. Christie, Robert W. Irving, "Sorting Strings by Reversals and by Transpositions," *SIAM J. Discrete Math*, vol. 14, num. 2, pp. 193–206, 2001.

[2] A. J. Radcliffe, A. D. Scott, E. L. Wilmer,"Reversals and transpositions over finite alphabets," *SIAM J. Discrete Math*, 2006.

[3] Cor A. J. Hurkens, Leo van Iersel, Judith Keijsper, Steven Kelk, Leen Stougie, and John Tromp, "Prefix Reversals on Binary and Ternary Strings," *SIAM J. Discrete Math*, vol. 21, num. 3, pp. 592–611, 2007.

[4] Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang, "Assignment of Orthologous Genes via Genome Rearrangement," *IEEE/ACM Trans. Comput. Biology Bioinform*, vol. 2, num. 4, pp. 302–315, 2005.

[5] Henrik Eriksson, Kimmo Eriksson, Johan Karlander, Lars J. Svensson, and Johan Wästlund, "Sorting a bridge hand," *Discrete Mathematics*, vol. 241, num. 1-3, pp. 289–300, 2001.