The string matching problem is a classic computer science problem. Here, we are given a text $T$ (with $n$ symbols). We want to determine whether $T$ contains a specific pattern $P$ (with $m$ symbols). Note that $n$ and $m$ can be large. This is a classical CS subject with many nice results. I will only give a quick sampling of what I think are most relevant to Bioinformatics. Refer to Gusfield’s book for much more discussions on string algorithms.

1 Suffix Tree

The concept of suffix is fundamental to string algorithms covered in this class. A suffix is a portion of string (including the last character).

A fundamental data structure in string matching is the suffix tree. A suffix tree for a string $T$ is leaf-labeled by suffix number at leaves. Edges are labeled with substrings of $T$. There are two key properties

- From root to a leaf, the concatenation of the substrings on the visited edges is equal to the suffix labeled on that leaf.
- No two edges leaving a node has the same starting character.

Let us look at an example. We let $T = tartar\$, where \$ is the smallest symbol that does not appear in the rest of $T$. Then we have 7 suffixes: $suff_7 = \$, $suff_6 = r\$, $suff_5 = ar\$, $suff_4 = tar\$, $suff_3 = rtar\$, $suff_2 = artar\$ and $suff_1 = tartar\$. The suffix tree is shown in the following figure (with $suff_7$ omitted).

![Suffix Tree Example](image)

Figure 1: Suffix tree for $T = tartar$.

We first show that a suffix tree can greatly speed up string matching. For simplicity, we assume constant-sized alphabet. This is the case when we are working with English or DNA sequences (which has a alphabet size of only 4). Suffix tree allows very efficient string matching. Even when text is very long (i.e. $m$ is very large), the time taken to matching only depends on the length of pattern (i.e. $n$, which is usually much smaller than $m$). To see how this is done, first convince yourself that we only need to find whether there is some suffixes of $T$ whose prefix matches $P$. This implies that any pattern contained in $T$ can be found from a path from the root down to somewhere in the tree; and position of any suffix (leaf) under that point is a match for the pattern. Then we just follow the suffix tree from the root and try to match each character of $P$ against the outgoing branches. Since
each branch begins with a distinct symbol, there is at most one branch we need to follow. Since the alphabet size is constant, such a branch (if exists) can be found in constant time.

An important result in string algorithm appears in 1973, which showed that we can build a suffix tree in $O(n)$ time (where $n$ is the length of the string). This gives us an $O(n)$ time algorithm (which needs $O(m)$ preprocessing time) for matching $P$ on $T$: first construct a suffix tree for $T$ (which takes $O(m)$ time); then matching $P$ on the suffix tree can be done in $O(n)$ time. The linear time algorithm for constructing suffix tree, however, is fairly involved. Thus we will not go over that here in this class. Refer to Gusfield’s book for more details on that.

## 2 Suffix Array

Linear-time string matching algorithm has been around for over 30 years. But the algorithms known till recently are too complicated, even after much efforts in simplification attempts. Recently, a data structure called suffix array becomes popular. Initially proposed by Mamber and Myers as an space-efficient alternative to suffix trees, suffix array has some very interesting results. It has very simple linear-time algorithm to construct directly (which we will illustrate below), and suffix array is also easily converted to suffix tree. Also, it motivates more interesting study related to Burrows-Wheeler Transform (BWT, which we will discuss soon). Therefore, nowadays, suffix array is used more often than suffix tree in string matching. Suffix tree, however, is still useful in term of getting intuition: you can think in term of suffix tree, and implementation is done by suffix array.

Suffix array is a (lexicographically) sorted list of suffixes. That is, each cell of a suffix array stores a suffix (the starting position of the suffix), which appears in lexicographical order. For example, let $T = tartar$. Then the suffix array for $T$ is: $[7, 5, 2, 6, 3, 4, 1]$

A main benefit of suffix tree is the memory reduction. Suppose the string length $n$ is within 32-bit limit (i.e. no more than 4 Gb). Then each suffix can fit into one integer (32 bits each). As a comparison, on average, suffix tree needs 100 bits per character. How can we store string matching with suffix array? Here is a property of suffix array: suffixes with same prefix will be grouped together in suffix array. Thus, we can perform binary search on suffix array. Naive implementation will lead to $O(n \log m)$ time algorithm for string matching, which appears to be slower than string matching with a suffix tree. The time bound, however, can be improved. We will omit details here (see Gusfield’s book).

Moreover, suffix array can be constructed from a given suffix tree. In fact, we simply perform a depth-first search by first picking branches with alphabetically smaller starting characters. The DFS will list all the suffixes lexicographically. Thus, we have a simple linear time algorithm for building suffix array: first construct a suffix tree and then perform DFS lexicographically. However, this is not so ideal: we want to construct suffix array directly without going through suffix tree construction to fully achieve the benefits of suffix arrays.

In the following, I describe a simple direct linear-time algorithm for constructing suffix array. Construct suffix array directly from text in linear time is a major open problem in string matching until 2003, where there are three linear-time algorithms published about the same time. This has puzzled for ten years, and three groups of people solved it (a little different methods) in 2003. All three algorithms have some similar structure. We will focus on one algorithm that is particularly simple. In the following, I focus on the basic ideas. For the ease of understanding, I omit some technical details.

The basic idea is divide and conquer. How are we going to divide the problem into subproblems? One may say do first half and second half. That is, you first sort the suffixes in the first half and then sort the second half. But this does not really work well. A better way is to divide suffixes into odd position or even position. Why is this useful? Note that sorting suffix is different from sorting a list of numbers. That is, $suffix_i$ and $suffix_{i+1}$ are not independent: $suffix_i$ is the concatenation of character $T[i]$ and $suffix_{i+1}$. This simple fact is used frequently in the following.
We consider two suffixes \( \text{suffix}_{i+1} \) and \( \text{suffix}_j \). If we know which one is lexicographically larger, then we can decide in constant time which of \( \text{suffix}_i \) and \( \text{suffix}_j \) bigger. Why? We first compare the first character \( T[i] \) and \( T[j] \). If we have a mismatch, we already know which suffix is larger. What if we have a match? In this case, that depends on which of \( \text{suffix}_{i+1} \) and \( \text{suffix}_{j+1} \) is larger (which we already know). This is roughly the high-level idea this approach takes, but with an important twist: instead of do odd-even, we do tri-partition: \( 0,1,2 \mod 3 \). We now sketch the high-level idea of this tri-partition algorithm. You should think about why odd-even partition does not work.

Let us assume for now that the first suffix starts at position 0 (instead of 1). Suppose we recursively sort all mod-1 and 2 suffixes. Let us look at one example: \( T = \text{yabbadabbado} \). Let the set of suffixes at mod-i positions be \( S_i \). So, \( S_1 = [1, 4, 7, 10] \) and \( S_2 = [2, 5, 8, 11] \). Suppose recursively, we sort \( S_1 \cup S_2 \) (this is done by the conquer step). This gives a sorted list: \( [1, 4, 8, 2, 7, 5, 10, 11] \). The key is that we can use this sorted list \( S_1 \cup S_2 \) as the base to sort the entire suffix array. In particular, we will first sort suffixes in \( S_0 = [0, 3, 6, 9] \). We first observe that it is easy to know which one in \([0, 3, 6, 9]\) is larger or smaller. Why? Suppose we want to know which of \( \text{suffix}_0 \) and \( \text{suffix}_3 \) is smaller. This is immediate since \( \text{suffix}_0 \) begins with \( y \) and \( \text{suffix}_3 \) begins with \( b \) (and \( \text{suffix}_3 \) is smaller). Now we look at \( \text{suffix}_0 \) and \( \text{suffix}_3 \), which both begin with \( a \). In this case, we look at \( \text{suffix}_7 \) and \( \text{suffix}_1 \). This is also easy by checking \( S_1 \cup S_2 \) (these two following suffixes must be in this set; think about why this is the case). So we have \( \text{suffix}_0 < \text{suffix}_3 \). Clearly a standard sorting \( O(n \log n) \) algorithm works. But sorting \( S_0 \) can be done faster with radix sort: basically each of the suffix in \( S_0 \) can be coded with two symbols: the first being the starting character and the second being the suffix adjacent to the right. The radix sort needs only \( O(n) \) time to sort the \( S_0 \) list. In this example, the sorted \( S_0 = [6, 9, 3, 0] \).

Now we have two sorted lists of suffixes: \( S_1 \cup S_2 \) and \( S_0 \). To come up with the complete sorted list, we only need to \( merge \) the two sorted lists. This can be done with an approach similar to the merge sort: maintain one pointer per list which is initialized at the left end; compare the suffixes pointed by the two pointers and put the smaller suffix into the complete list. The key is to efficiently compare two suffixes. In worst case, \( O(n) \) time seems to need since you may need to compare every letter in the two suffixes. But this is not necessary. Note that \( S_1 \cup S_2 \) has 1 or 2 (mod 3) suffixes and \( S_0 \) has 0 (mod 3) suffixes. Suppose we compare 1 (mod 3) suffix with a 0 (mod 3) suffix. We exam the first letter of the two suffixes; if mismatch, done; if match, then we end up with 1 (mod 3) and 2 (mod 3) suffixes, whose relative magnitude is known! For example, we compare \( \text{suffix}_0 \) and \( \text{suffix}_1 \). They match (with \( a \)) at the first position. So we look at \( \text{suffix}_7 \) and \( \text{suffix}_2 \), and \( \text{suffix}_2 \) is smaller. So we put \( \text{suffix}_1 \) into the final list. What if we need to compare a 2 (mod 3) suffix with a 0 (mod 3) suffix? If the first character matches, we have a 0 (mod 3) suffix and a 1 (mod 3) suffix, which are not in the same sorted list. This seems to be a problem. However, if we compare one more character and that leads to a 1 (mod 3) suffix and a 2 (mod 3) suffix which are in the same sorted list \( S_1 \cup S_2 \). The bottom line is that at most 3 comparisons are needed to determine which suffix of two is smaller so as to put in the sorted list.

Finally, we perform time analysis. We are doing recursion over a sub-problem of size \( \frac{2n}{3} \). So we have the running time \( T(n) = T(2n/3) + cn \). This gives \( O(n) \) running time (by the Master theorem of recurrences). Some details are not discussed. For example, how exactly divide and conquer are performed (they are suffixes, not just unrelated numbers)? Well this can be done by properly maintain the data structure (omitted and you should refer the paper for more details).