1 Introduction of suffix tree and suffix array

Building a suffix tree is an important result of 1973, which builds suffix tree in $O(n)$ time for a string $S$ with $n$ symbols. Suffix tree is leaf-labeled by suffix number at leaves. Edges are labeled with substrings of $S$. A key property is, from root to leaf, the substrings along the path spell out the suffix on that leaf. Also, no two edges out a node have the same starting symbol. Note that every substring of $S$ can be found on the tree, starting from the root. Now try to build suffix yourself for $S=\text{tartar}$ and get the first hand taste of what a suffix tree is.

Why is suffix tree useful? Suppose we want to find whether pattern $P=\text{tar}$ is in $S=\text{tartar}$. We first build a ST. Then search for tar in it. How to proceed? We just match $P$ with the unique path that agree with $P$. The first thing to note the path is indeed unique: that is, at each node, we will have at most one branch to follow. This is because in ST, each branch starts with different char. If we can not continue matching (while still something left in $P$), we conclude it is not in. Otherwise, ALL leaves under the current ending position has an exact match of $P$. Why? The crucial point is that if pattern $P$ starting in $T[j]$, iff suffix $T[j..m]$ starts with $P$. Thus all positions of $T$ matching $P$ must have the same $T$. When alphabet size is constant, the search for path to go can be done in constant time. Thus, pattern matching can be done in linear time.

Linear-time pattern matching has been around for over 30 years. But the algorithms known till very recently are too complicated, even after much efforts in trying to simplify. Fortunately, something called suffix array comes to rescue. Initially proposed by Manber and Myers as an space-efficient alternative to suffix trees, suffix array has some very interesting results. In particular, it has very simple linear-time algorithm to construct directly, and suffix array is also easily converted to suffix tree. Therefore, nowadays, suffix array is used more often than suffix tree in pattern 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.

So what is: suffix array introduced by Manber and Myers? We start with the same string $\text{tartar}$.

POS is an array of suffix; each cell stores a suffix, and listed in lexicographic order regarding to the corresponding suffix. Here, POS = [ 7, 5, 2, 6, 3, 4, 1].

A main advantage of suffix array is the memory reduction. But I still think suffix tree is more conceptually clearer when thinking of things. First, suffix array can be easily built from suffix tree: just a depth-first search in a lexicographically (well, when the alphabet is constant size). How to build suffix tree from array? We need one additional piece of data: LCP array (also called depth array). The LCP array stores the length of the longest common string for two suffix, next to each other in POS. For our example here $S=\text{tartar}$: LCP = [0, 0, 2, 0, 1, 0, 3].

Here is how you will generate suffix tree from POS and LCP arrays. Essentially, start from smallest (first) suffix, and when move to the next, create a new branch on the tree, from the current
tree. Where to branch out? Well, depends on the depth array. From the last added suffix si (leaf), backup till depth of LCP(i). O(1) per edge when backing up. Create a new internal node if needed and then branching off an edge. Question: will this violate the property of suffix tree by creating an edge whose label has the same start symbol as another existing edge? Convince yourself it will not. Now how about running time? The main concern is that insertion of new branches leads to traversal of (sometimes multiple) edges. How to bound such traversal? The key is that only the path leading to the rightmost suffix is traversed. Once traversed, it is no longer part of the new rightmost path, and so it will not be traversed again. All other operations are only a single action per suffix. So entire conversion takes O(n).

How can do pattern matching in suffix array? The simplest thing is to do binary search. This gives \(O(n \log(m))\) time where m is the pattern size. But there are more advanced techniques, which gives \(O(n + \log(m))\). We omit such details.

See Gusfield’s notes on LCP arrays, which explain how to build the LCP array efficiently.

We now focus on the problem of building the POS array in O(n) time. This problem has puzzled people for ten years, and three groups of people solved it (a little different methods) in 2003. All the three methods use the same high-level idea: that is, first sort part of string (called a sample), and then sort the rest of string from the sorted sample. The three methods differ by the way of choosing samples and how to merge.

The main idea is divide and conquer. But how are we going to divide? One may say do first half and second half, but this does not seem to do much. Another is to say odd position or even position. Why is this useful? Sorting suffix is different from sorting a list of numbers. suffix i and i+1 is not independent (unlike sorting an array of numbers)! Look at two positions: i and j. If we know which one is (lexicographically) bigger, then we can decide in constant time which of suffix i+1, j+1 bigger? Why? Compare the first char, and if equal then (i+1,j+1) pair tells us the result. This is roughly the high-level idea this approach takes, but with an important twist: instead of doing odd-even, we do tri-partition: 0,1,2 mod 3. Let us give an example to see how this works.

Here we start numbering position from 0. As an example, consider T= y a b b a d a b b a d o. For this string, POS=[12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0]. We list suffix for 1 mod 3 (in an array called \(B_1\)) and list suffix for 2 mod 3 (in an array called \(B_2\)). For this example, \(B_1=[1,4,7,10]\), \(B_2=[2,5,8,11]\).

**Step 1: Sort suffix in \(B_1\) and \(B_2\)**

For this purpose, we will sort this string: [abb][ada][bba][do$] [bba][dab][bad][o$$]. Note each triple (encapsulated in []) becomes a new character. This string is longer than the original string, but we just replace the triple with the ranks of these triples in the matrix by their ranks. This is done by radix sort of the letters present in the samples. This turns to:

- abb 1
- ada 2
- bba 4
- do$ 6
- bba 4
- dab 5
- bad 3
- o$$ 7

So we have the new string (by plugging in the ranks) as \(R'=[1,2,4,6,4,5,3,7]\). Then we sort this recursively (and note \(R'\) is shorter than \(T\) by 1/3). And suppose we get \(SA(R')=[8, 0, 1, 6, 4,\)
2, 5, 3, 7 (recall suffix start from 0). Use radix sort is the simple way of converting the combo alphabet back to integer alphabet, so that we can use recursion!

Also, create a rank for every suffix for the sampled positions. Rank(R') = x, 1, 4, x, 2, 6, x, 5, 3, x, 7, 8, x, 0, 0 (note use 0 for the last two positions) Be careful: how do we know this? SA(R') = [8, 0, 1, 6, 4, 2, 5, 3, 7], so rank(R') = 1, 2, 5, 7, 4, 6, 3, 8, 0 (simply put the current rank in the corresponding position and increase the current rank by 1). What does this mean? Want to compare words: convert to the normal positions (because we let 1 mod 3 go first, then followed by 2 mod 3 go next). It now becomes: 1, 4, 2, 6, 5, 3, 7, 8, 0, 0 (fill 00 near the end).

**Step 2. Sort positions not sampled (i.e. mod 3 == 0)**

Now this becomes easy: For a pair for each of these position (S[i], rank(S[i + 1])), Then do radix sort of this pairs, and it takes only O(n) time! Just line them up like before, and sort it. In our example, we have: (y, 1), (b, 2), (a, 5), (a, 7), which gives (0, 0) < (a, 5) < (a, 7) < (b, 2) < (y, 1).

**Step 3: Merge**

Standard comparison merge! In more detail: we are merging one list contains 0 mod 3 suffix (called M0), and the other list contains 1 or 2 mod 3 suffix (called M1 and M2). So, we have two cases: compare M0 with M1, and M0 with M2. If we compare M0 with M1, (say s0 and s4) we first see if S[0] ≠ S[4]. If so, we are done (which suffix is larger is already known). Otherwise, we then look to see if s[1] and s[5], which one is larger (and this is already known in one of the list B1 and B2). If we compare M0 with M2 (say s0 with s5), we compare two characters first and then the case reduce to the original situation. Now we are done!

Finally, time analysis. We are doing recursion over 2/3n, so we have: T(n) = T(2n/3) + O(n). This gives O(n) running time.

### 2 Pattern matching using suffix array

We now consider the problem of searching for pattern string P in text T, whose suffix array POS and LCP array are given. We first review what we briefly discussed in class before. Since POS array is sorted, to find which suffix’s prefix matches P, we can perform a binary search as follows.

We maintain the current search interval L and R, which are initialized to 1 and m respectively. We compare P with the middle suffix \( M = [(L + R)/2] \) by a simple direct comparison of P and the suffix. If we see a match, we are done. Otherwise, if P is lexically less than this middle suffix, we know P has to appear in the first interval. We then change R to be one less than the middle position, and continue. The other case is similar. In general, this is the standard binary search procedure. As we all know, binary search runs in O(\( \log(m) \)) iterations. Since each iteration, we may need to compare the whole O(n) symbols, the search takes O(nlog(m)) time. This is not bad when m is not very large, but clearly its performance is worse than that of the suffix tree case. We now show some simple tricks to speed up the search procedure.

First, it seems wasteful to compare the whole P and a suffix each time. A simple observation can help. Suppose during search, we keep track the matched length of P and suffix POS(L) (denoted as l) and of P and suffix POS(R) (denoted as r). Then a moment of thought suggests that P must match the first \( mlr = \min(l, r) \) symbols. That is, we can skip the first \( mlr \) symbols in matching. Now try to think about it yourself to make sure you understand why this is the case. In practice, this simple trick alone can bring down the expected running time of the search to O(n + \( \log(m) \)).
We now take a closer look at the binary search. Note that to really reduce the running time of search, it is critical to reduce the number of redundant character comparison. Here, redundant comparison means a symbol in P is compared multiple times. Recall that this is not the case in suffix tree search, where we only match each symbol in P once. In the above speedup trick, we avoided mlr redundant search, but the comparison from mlr to max(l,r) is still redundant. To eliminate these redundant comparison, we need to find a way to start searching from max(l,r) instead. This needs the LCP array. Recall that the LCP array stores the length of the longest common prefix of two adjacent suffix in POS. For the purpose of matching, we need an extension of LCP. For any two positions i and j, we define LCE(i,j) as the length of longest common prefix of suffix POS(i) and POS(j). Note that this means LCP(i) = LCE(i-1, i) when \( i \geq 2 \). In the following, we will assume the LCE values for needed pairs of positions are known. We will address the issue of how to obtain LCE values later.

Now consider several cases. First, if \( l=r \), there is no extra redundancy, and so start matching from mlr + l. (Make sure you see why there is no redundancy.) We now assume \( l > r \). (Note the case \( l < r \) is similar.) That is, the left suffix matches more of P than the right suffix. We then consider three sub-cases.

1. \( LCE(L, M) > l \). This implies P can not occur between L and M. This is because P is lexically larger than the left suffix and the middle suffix matches at the position where P and the left suffix differ, which implies P is also lexically larger than the middle suffix. So in this case, we move L to M. No comparison is needed. And we keep l and r the same.

2. \( LCE(L, M) < l \). It is not hard to see that P must be lexically smaller than the middle suffix (why?). Then we move R to M, change r=LCE(L,M) (why?). No comparison is needed.

3. \( LCE(L,M)=l \). Then P can start from l+1.

It is now clear that with the help of LCE values, we can get rid of redundant character comparison since the above shows we can start comparing at max(l,r) instead of min(l,r). If you are still not convinced, you should check closely to see how values l, r change, where comparisons occur and so on.

The only remaining issue is on LCE values. There are two concerns: first, there may be too many LCE values we need; second, computing LCE values may be slow (not linear time in the length of T). Fortunately, it turns out here are only \( O(m) \) LCE pairs needed, and compute these LCE values takes only \( O(m) \) time. So we can just do that in the pre-processing stage and these LCE values can be used in later pattern matching.

To summarize, pattern matching in suffix array can be done in \( O(n+\log(m)) \) time (after some additional pre-processing, which takes \( O(m) \) time).