1 Definition of Suffix Arrays

Imagine that you write down all the suffixes of a string T of length t. The i^{th} suffix is the one that begins at position i. Now imagine that you sort all these suffixes. And you write down the indices of them in an array in their sorted order. This is the suffix array. For example, suppose $T =$ `banana$` and we've sorted the suffixes:

```
   0  1  2  3  4  5  6
   b  a  n  a  n  a  $

6:  $
5:  a$
3:  ana$
1:  anana$
0:  banana$
4:  na$
2:  nana$
```

The numbers to the left are the indices of these suffixes. So the *suffix array* is:

```
6 5 3 1 0 4 2
```

This array can be computed in a straightforward way by sorting the suffixes directly. This takes $O(t^2 \log t)$ time because each comparison of two strings takes $O(t)$ time. In fact, the suffix array can be constructed in $O(t)$ time. We will see faster construction algorithms in a minute.

It's often handy to have an auxiliary array called the "LCP" array around. Each successive suffix in this order matches the previous one in some number of letters. (Maybe zero letters.) This is recorded in the common prefix lengths array, or the LCP array. In this case we have:

```
suffix array is:               6 5 3 1 0 4 2
common prefix lengths array     0 1 3 0 0 2
```

2 Searching T using the suffix array

Consider the standard string search problem: we have the suffix array A for a string T and we want to find where pattern P occurs in A.

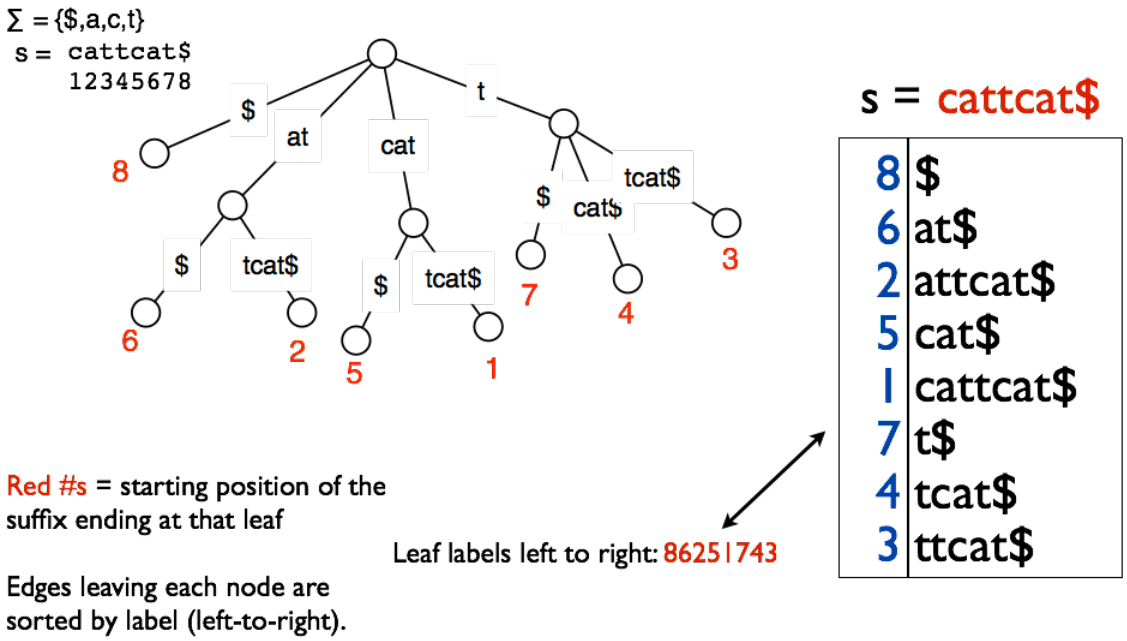
We can do this in $O(|P| \log |T|)$ time using binary search using the suffix array: Maintain a range $[U, D]$ of candidate positions in the array; initially the range is the entire suffix array. Let $M(U, D)$ be the midpoint of that range. Repeatedly check whether P comes before or after the suffix at position $M(U, D)$ of the array, and update U and D accordingly. This takes $O(|P| \log |T|)$ since we're doing a binary search over $|T|$ elements, and each comparison of P against the string at suffix $M(U, D)$ takes $O(P)$ time.

How do we find *all* the occurrences of P in T? The thing to note is that these will all be adjacent in the suffix array since all the suffixes that start with P obviously start with the same sequence. We can find the range that starts with P in two ways: The first way: we could do 2 binary searches: one that takes the "left" range when there is a tie — which will find the start of the range — and one which takes the "right" range when there is a tie — which will find the end of the range. These searches will give you the range which contains suffixes starting with P. The second way: using the LCP array, we can walk left and right from a suffix that starts with P, continuing as long as the LCP is $\geq |P|$.

Note that $O(|P| \log |T|)$ time to search is slower than we got with suffix trees — we can do better, however. In fact, there is an $O(|P|)$ algorithm to search, matching the time for suffix trees. If there's time, we will see an $O(|P| + \log |T|)$ time algorithm that is almost as good.

3 Suffix Array ↔ Suffix Tree

Suffix Tree → Suffix Array. The suffix array can be computed from the suffix tree by doing an in-order traversal of the tree, where in-order means that we visit children in lexicographic (alphabetical) order.



This takes $O(|T|)$ time.

Suffix Array → Suffix Tree. We add the suffixes one at a time into a partially built suffix tree in the order that they appear in the suffix array. At any point in time, we keep track of the sequence of nodes on the path from the most recently added leaf to the root. To add the next suffix, we find where this suffix's path deviates from the current path we're keeping track of. To do this, we just use the common prefix length value. We walk up the path until we pass this prefix length. This tells us where to add the new node.

A potential argument can be used to see that this process runs in linear time. Imagine a token on each of the edges on the path from the current leaf to the root. We use these tokens to pay for

walking up the tree until we find the branch point where a new child is added. The tokens on the path pay for the steps we take up the tree. We'll need a new token for the edge that connects to the new leaf. We may also need another token in case we have to split an edge. So in all, at most two new tokens are needed to pay for the work. This proves that the running time is linear.

4 Constructing Suffix Arrays

The key idea to improve on the simple $O(|T|^2 \log |T|)$ algorithm to construct suffix arrays is to use the fact that we are not sorting a collection of arbitrary strings, but rather strings that are related by the fact that they are suffixes of the same string. There are a number of ways to take advantage of this so that you can do the string comparisons required in the sorting in less than $O(|T|)$ time. In fact, it's possible to construct the array directly in $O(|T|)$ time. We will see a simpler $O(|T| \log |T|)$ algorithm.

The first idea behind the algorithm is that if we could sort the suffixes by their prefixes of length 2, then we would be making progress. The second idea reduces the problem of sorting by longer and longer prefixes to the problem of sorting by things of length 2. Let's start with the first step of the algorithm: Given a string $s = a_1 a_2 a_3 \ldots$ we sort its suffixes using only their first 2 characters as a key. This can be done by creating and sorting an array of triples:

$$\{(a_i, a_{i+1}, i)\}$$

We pretend the string is padded at the end with an infinite string of `$` characters.

Let's keep a running example of $s =$ `cattcat$`. Sorting the suffixes just by their first 2 characters gives us the following (don't worry about the numbers to the left yet):

```
0 $$
1 at tcat$
1 at $
2 ca ttcat$
2 ca t$
3 t$
4 tc at$
5 tt cat$
```

The main trick is that (1) there are at most n different 2-character prefixes in the string, and (2) we can use their sorted order as a *code* for 2-character groups in the string. The numbers to the left above give this code (which can be computed by walking down the sorted list comparing 2-character prefixes). Using this code, we can re-write the string:

```
cattcat$
21542130
```

But note: now comparing tuples (a_i, a_{i+2}) in this "coded" version of the string is the same as comparing prefixes of length 4 in the original string since the coding obeys the same order as the 2-character groups. So, e.g., the pair $(5, 2)$ represents the string `ttca`.

So we repeat the process, now with a new "alphabet" of $\{0, \ldots, |T|\}$. On the next round, the "super-characters" will represent 4 original characters, and then 8 and so on, doubling each time until we are sorting by all n characters on the $O(\log |T|)$th round.

There are $O(\log |T|)$ rounds. At each round we do linear work to create the code plus $O(|T| \log |T|)$ work to sort (now each sort need only compare tuples of 2 super-characters, which takes constant time). That gives us an $O(|T| \log^2 |T|)$ algorithm.

We can reduce this to $O(|T| \log |T|)$ by using a radix sort in all but the first round of sorting. Once we are sorting "coded" strings, we know the alphabet is the set $\{0, \ldots, |T|\}$, so we can sort in $O(|T|)$, removing one of the log factors.

5 Faster search

The simple binary search scheme described above to find a pattern P in a text T took $O(|P| \log |T|)$ time. But again, we did not use the fact that the things we were comparing were related strings rather than arbitrary entries in an array. Using this, one can get an $O(|P| + \log |T|)$-time search algorithm.

To explain the algorithm, we need to define $lcp(X, Y)$ as the length of the longest common prefix between suffix X and suffix Y. Note that our LCP array gives this value directly when $Y = X + 1$. And note that $lcp(X, Y)$ is exactly the depth of the LCA of suffix X and Y in the suffix tree. For now, assume we have easy access to these values.

Throughout the following, we will conflate the string at suffix x with the suffix number x and also the position of x in the suffix array.

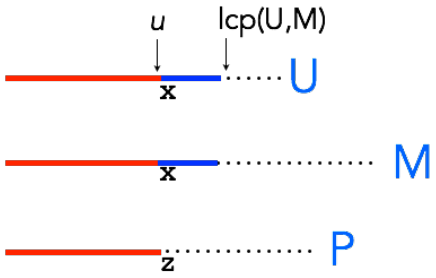
Using the $lcp(X, Y)$ values, we can avoid repeatedly comparing characters of P during our binary search. Recall that our binary search is maintaining a range $[U, D]$. Now we also maintain u and d:

$$u = \text{length of the longest prefix of } U \text{ that matches a prefix of } P$$
$$d = \text{length of the longest prefix of } D \text{ that matches a prefix of } P$$

At the start, these can be computed by directly comparing P to the first and last suffix. Our binary search will now update u and d as well as U and D.

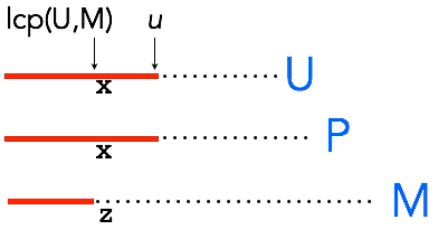
Suppose $u > d$. Let M be the midpoint of the range $[U, D]$. We have three cases that we can use to update the binary search range quickly:

Case #1: $lcp(U, M) > u = lcp(U, P)$. Then the situation looks like this:



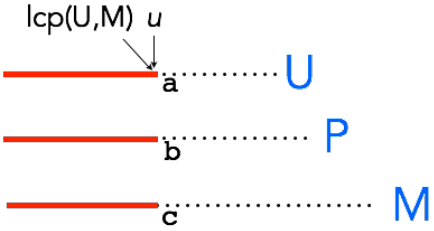
and we know that P comes after M. We need 0 new character comparisons to discover this. d and u are unchanged.

Case #2 $lcp(U, M) < u = lcp(U, P)$. Then the situation looks like this:



and we know that P must be before M. We need 0 new character comparisons to discovery this. u is unchanged and $d \leftarrow lcp(U, M)$.

Case #3 $lcp(U, M) = u = lcp(U, P)$. This looks like this:



Here, we have no information about where P should go, be we know we can start comparing at position u.

When $u < d$, we have the above cases, but with the roles of u and d and U and D swapped. We have one more case:

Case #0 $u = d$. In this case, we start comparing P to M from position $u + 1 = d + 1$.

5.1 Algorithm Summary

Set $[U, D]$ to be the entire suffix array. Compute u and d directly by looking at the first and last suffixes of the array. Repeatedly do the following:

If $u = d$, apply case 0
If $u > d$, apply case 1,2,3 as appropriate
If $u < d$, apply case 1,2,3 as appropriate, but using D and d in place of U and u.

5.2 Running time.

Only cases 0 and 3 actually compare any characters, and they always start comparing at $\max(u, d)$. If they match k characters of P, then one of u or d will be incremented by k and we will never look at those characters again. Thus, *matching* characters takes at most $O(|P|)$ time.

The *mismatching* characters might be compared more than once, but there is only 1 mismatch per iteration, since we stop comparing as soon as there is a mismatch. Since there are still $O(\log |T|)$ iterations in the binary search, we spend at most $O(\log |T|)$ time comparing mismatching characters.

This leads to an overall running time of $O(|P| + \log |T|)$, nearly matching the running time of suffix trees. (The additive $\log |T|$ factor can be removed with a more sophisticated algorithm.)

5.3 How do we get the $lcp(X, Y)$ values

First, even though it looks like there are $O(|T|^2)$ relevant lcp values, in fact this is not the case. The only values that are relevant are the ones where X and Y are the end points of a range encountered during a binary search. There are $O(n)$ of these for a binary search on a length-n array. (Exercise: prove this!).

Second, $lcp(x, y) = \min_{k=x,\ldots,y-1} lcp(k, k+1)$. Exercise: prove this! The values $lcp(k, k+1)$ are the values in our LCP array stored with the suffix array. So we need a way to find the minimum over ranges in a list of integers. That is the *range min query* problem. We don't have time to cover that in this lecture (but we'll find a way to get you information on that if you are interested.)

6 Summary

We saw different approaches to exact string matching problems: given a pattern P and a text T, find all occurrences of P within T. We saw:

- The randomized Karp-Rabin fingerprinting scheme which also has a similar running time. However, it is a versatile idea and extends to different settings (like 2-dimensional pattern matching).
- The suffix-tree construction. Here you preprocess the text in $O(t)$ time and space, and then you can perform many different operations (including searching for different patterns) in time $O(p + n_{p,t})$, where $n_{p,t}$ is the number of occurrences of pattern p in text t.

There are also deterministic searching algorithms that we did not have time to cover such as:

- the Knuth-Morris-Pratt algorithm which runs in time $O(t + p)$. Here if you have a pattern you want to find in many texts, you can preprocess the pattern in $O(p)$ time and space, and then search over multiple texts, the search in text T_i taking the time $O(|T_i|)$.

Each one its advantages.