15-451/651: Design & Analysis of Algorithms November 27, 2018
Lecture #24: Suffix Trees and Arrays last changed: November 26, 2018

We're shifting gears now to revisit string algorithms. One approach to string problems is to build a more general data structure that represents the strings in a way that allows for certain queries about the string to be answered quickly. There are a lot of such indexes (e.g. wavelet trees, FM-index, etc.) that make different tradeoffs and support different queries. Today, we're going to see two of the most common string index data structures: suffix trees and suffix arrays.

1 Suffix Trees

Consider a string T of length t (long). Our goal is to preprocess T and construct a data structure that will allow various kinds of queries on T to be answered efficiently. The most basic example is this: given a pattern P of length p, find all occurrences of P in the text T. What is the performance we aiming for?

- The time to find all occurrences of pattern P in T should be $O(p+k)$ where k is the number of occurrences of P in T.
- Moreover, ideally we would require $O(t)$ time to do the preprocessing, and $O(t)$ space to store the data structure.

Suffix trees are a solution to this problem, with all these ideal properties.[1] They can be used to solve many other problems as well. In this lecture, we'll consider the alphabet size to be $|\Sigma| = O(1)$.

1.1 Tries

The first piece of the puzzle is a *trie*, a data structure for storing a set of strings. This is a tree, where each edge of the tree is labeled with a character of the alphabet. Each node then implicitly represents a certain string of characters. Specifically, a node v represents the string of letters on the edges we follow to get from the root to v. (The root represents the empty string.) Each node has a bit in it that indicates whether the path from the root to this node is a member of the set—if the bit is set, we say the node is marked.

Since our alphabet is small, we can use an array of pointers at each node to point at the subtrees of it. So to determine if a pattern P occurs in our set we simply traverse down from the root of the tree one character at a time until we either (1) walk off the bottom of the tree, in which case P does not occur, or (2) we stop at some node v. We now know that P is a prefix of some string in our set. If v is marked, then P is in our set, otherwise it is not.

This search process takes $O(p)$ time because each step simply looks up the next character of P in an array of child pointers from the current node. (We used that $|\Sigma| = O(1)$ here.)

1.2 Tries $\rightarrow$ Suffix Trees

Our first attempt to build a data structure that solves this problem is to build a trie which stores all the strings that are suffixes of the given text T. It's going to be useful to avoid having one suffix match the beginning of another suffix. So in order to avoid this we will affix a special

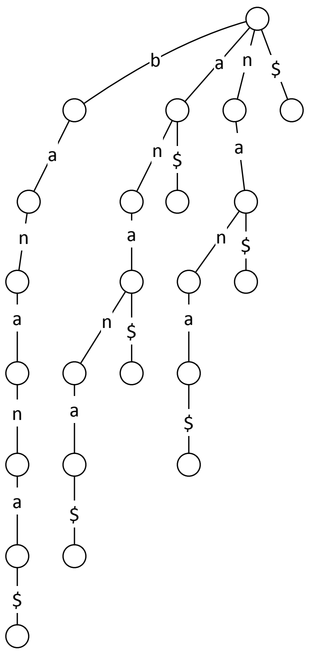
[1] Suffix trees were invented by Peter Wiener.

character denoted "\$" to the end of the text T, which occurs nowhere else in T. (This character is lexicographically less than any other character.)

For example if the text were $T =$ `banana$`, the suffixes of T are then

```
banana$
anana$
nana$
ana$
na$
a$
$
```

(Since we appended the \$ sign to the end, we're not including the empty suffix here.) And the trie of these suffixes would be:



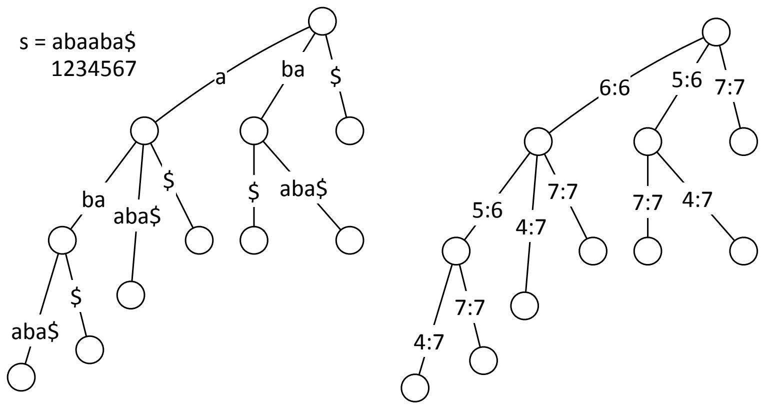
Suppose we have constructed the trie containing all these suffixes of T, and we store at each node the count of the number of leaves in the subtree rooted at that node. Now given a pattern P, we can count the number of occurrences of P in T in $O(|P|)$ time: we just walk down the trie and when we run out of P we look at the count of the node v we're sitting on. It's our answer.

But there are a number of problems with this solution. First of all, the space to store this trie could be as large as $\Theta(t^2)$. [Exercise: find an example with a constant-size alphabet where this is true.] Also, it's unsatisfactory in that it does not tell us where in s these patterns occur. Finally, it will also take too long to build it.

Shrinking the tree to $O(t)$ space. The first two issues are easy to handle: we can create a "compressed" tree of size only $O(t)$. Since no string occurs as a prefix of any other, we can divide the nodes of our trie into *internal* and *leaf* nodes.

The leaf nodes represent a suffix of T. We can have each leaf node point to the place in T where the given suffix begins.

If there is a long path in the trie with branching factor 1 at each node on that path. We can compress that path into a single edge that represents the entire path. Moreover, the string of characters corresponding to that path must occur in T, so we can represent it implicitly by a pair of pointers into the string T. So an edge is now labeled with a pair of indices into T instead of just a single character (or a string). Here's an example (with the substrings labeling the edges on the left, and the start-end pairs labeling them on the right):



This representation uses $O(t)$ space. (We count pointers as $O(1)$ space.) Why? Each internal node now has degree at least 2, hence the total number of nodes in the tree is at most twice the number of leaves. But each leaf corresponds to some suffix of T, and there are t suffixes.

Building the tree — sketch. What about the time to build the data structure? Let's first look at the naïve construction, by adding suffixes into it one at a time. To add a new suffix, we walk down the current tree until we come to a place where the path leads off of the current tree. (This must occur because the suffix is not already in the tree.) This could happen in the middle of an edge, or at an already existing node. In the former case, we split the edge in two and add a new node with a branching factor of 2 in the middle of it. In the latter case we simply add a new edge from an already existing node. In either case the process terminates with a tree containing $O(t)$ nodes, and the running time of this naive construction algorithm is $O(t^2)$.

In fact, it is possible to build a suffix tree on a string of length t in time $O(t)$. We will not have time to go over this construction algorithm in detail, but there are several algorithms to do it.

2 Applications of Suffix Trees

We've seen how suffix trees can do exact search in time proportional to query string, once the tree is built. There are many other applications of suffix trees to practical problems on strings. We'll just mention just a few here.

2.1 Simple Queries

Suffix trees make it easy to answer common (and less common) kinds of queries about strings. For example:

- Count the number of occurrences of P in T: follow the path for P; the number of leaves under the node you end up at is the number of occurrences of P. If you are going to answer this kind of query a lot, you can store the number of leaves under each node in the nodes.
- Output the locations of all the occurrences of P in T: follow the path in the tree that spells out P. Every leaf under this node (if it exists) gives an occurrence of T. This subtree can be traversed in $O(k)$ time, where k is the number of occurrences. This leads to a total time of $O(p+k)$ which is what we were aiming for.
- Check whether P is a suffix of T: follow the path for P starting from the root and check whether you end at a leaf.
- Find the lexicographically (alphabetically) first suffix: start at the root, repeatedly follow the edge labeled with the lexicographically (alphabetically) smallest letter.
- Find the longest repeat in T. That is, find the longest string r such r occurs at least twice in T: Find the deepest node that has ≥ 2 leaves under it.

2.2 Longest Common Substring of Two Strings

Given two strings S and T, what is the longest substring that occurs in both of them? For example if $S =$ `boogie` and $T =$ `ogre` then the answer is `og`. How can one compute this efficiently? The answer is to use suffix trees. Here's how.

Construct a new string $U = S\%T\$$. That is, concatenate S and T together with an intervening special character that occurs nowhere else (indicated here by "%"). Let n be the sum of the lengths of the two strings. Now construct the suffix tree for U. Every leaf of the suffix tree represents a suffix that begins in S or in T. Mark every internal node with two bits: one that indicates if this subtree contains a suffix of S, and another for T. These bits can be computed by depth first search in linear time. Now take the deepest node in the suffix tree (in the sense of the longest string in the suffix tree) that has both marks. This tells you the the longest common substring.

This is a very elegant solution to a natural problem. Before suffix trees, an algorithm of Karp, Miller, and Rosenberg gave an $O(n \log n)$ time solution, and Knuth had even conjectured a lower bound of $\Omega(n \log n)$.

2.3 Searching for Matching Strings in a Database

The above idea can be extended to more than 2 strings. This is called a *generalized suffix tree.* To represent a set of strings $D = \{S_1, \ldots, S_m\}$, you concatenate the strings together with $m-1$ unique characters (1 unique character between each pair) and label the leaves with the index of the string in which that suffix begins. To find which strings in D contain a query string q you follow the path for q and report the indices that occur in the subtree under the node at which you stopped.

3 Suffix Arrays

Suffix trees are a great way to think about many string problems, and they use $O(|T|)$ space, but the constant hidden inside the big-O notation is somewhat large (a pointer for each child). Suffix Arrays let you answer many of the same queries still using $O(|T|)$ space, but with a better constant. Suffix trees are almost never used in practice for large strings, but suffix arrays are often quite practical.

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 alphabetically, 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
```

3.1 Constructing the suffix array

So how do we compute the suffix array and the common prefix lengths array? There are linear time algorithms for this, but here we will describe a probabilistic method that is $O(n \log^2 n)$.

It's based on Karp-Rabin fingerprinting. If we could compare two suffixes in $O(1)$ time we could then just sort them in $O(n \log n)$ time. Instead we'll use a method for comparing two suffixes that works in $O(\log n)$ time.

Using Karp-Rabin fingerprinting we can compare two strings for equality in $O(1)$ time (with a tiny error probability).[2] To compare two suffixes a and b to determine their lexicographic ordering, we use binary search to find the shortest length r such that $a[0 \dots (r-1)] = b[0 \dots (r-1)]$, but $a[0 \dots r] \neq b[0 \dots r]$. Then $a < b$ exactly when $a[r] < b[r]$. Furthermore this also tells us the common prefix length between the two suffixes.

(Section 6 gives a deterministic $O(n \log n)$ algorithm for suffix array construction.)

[2]Really, we're counting the number of arithmetic operations modulo a prime p, where p is polynomial in t.

3.2 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.

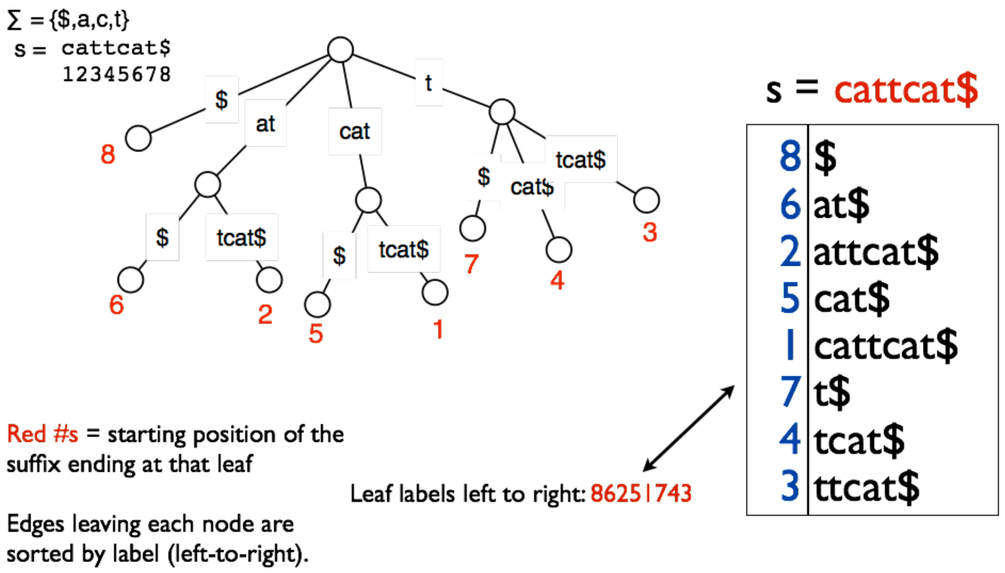
How do we find *all* the occurrences of P in T? The thing to note is that these will all be adjacent in the suffix array since all the suffixes that start with P obviously start with the same sequence. We can find the range that starts with P in two ways: The first way: we could do 2 binary searches: 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. In fact, there is an $O(|P|)$ algorithm to search, matching the time for suffix trees, which we will not have time to cover.

Optional section 7 gives an $O(|P| + \log |T|)$ time algorithm that is almost as good that uses the LCP array.

3.3 Suffix Array $\leftrightarrow$ Suffix Tree

Suffix Tree $\rightarrow$ 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:



Σ = {$,a,c,t}
s = cattcat$
12345678
$
at
cat
t
$
tcat$
$
tcat$
$
cat$
tcat$
8
6
2
5
1
7
4
3
s = cattcat$
8 $
6 at$
2 attcat$
5 cat$
1 cattcat$
7 t$
4 tcat$
3 ttcat$
Red #s = starting position of the suffix ending at that leaf
Leaf labels left to right: 86251743
Edges leaving each node are sorted by label (left-to-right).

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.

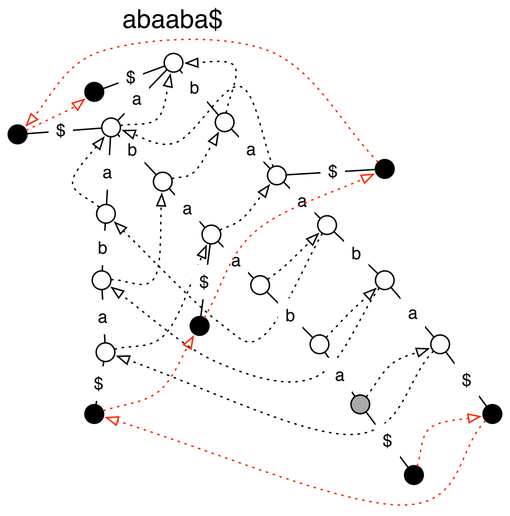
A potential argument can be used to see that this process runs in linear time. Imagine a token on each of the edges on the path from the current leaf to the root. We use these tokens to pay for walking up the tree until we find the branch point where a new child is added. The tokens on the path pay for 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 Another Application: Longest Common Substring

Suppose we can preprocess our large database text T in anticipation of many future (unknown) queries q where we want to find the longest common substring between T and q. To solve this problem, we need to introduce the concept of a *suffix link*:

Definition 1 *A* suffix link *is an extra pointer leaving each node in a suffix tree. If a node u represents string $x\alpha$ (where x is a character and α is a string) then u's suffix link connects from u to the node representing the string α.*

Every node has a suffix link. Why? Every node represents the prefix of some suffix. Suppose node u represents the prefix of suffix i of length m. Then u's suffix link should point to the node on the path representing the prefix of length $m-1$ of suffix $i+1$. A trie with all the suffix links shown as dashed arrows is given below:



If node representing string $x\alpha$ exists in our suffix tree even after compression, then a node representing α must exist. The reason for this is since the node $x\alpha$ exists, it has two children y and z. That means both the strings αy and αz must exist in the string. So the node representing by α must have 2 children too.

Using suffix links we can get an $O(|q|)$-time algorithm to find the longest common substring between T and q:

1. Walk down the tree, following q
2. If you hit a dead-end at node u,
   (a) save the depth of node u if it is deeper than the deepest dead end you found so far.
   (b) follow the suffix link out of u.
   (c) Go to step 1 (starting from your current place in q)
3. When you exhaust q, return the string represented by the deepest node you visited.

The first idea here is the longest common substring starts at some suffix — we just don't know which suffix, so we try them all starting with suffix 0. The second idea here is that following a suffix link chops off the first character of q taking you to the place in the tree that you would have been at if the query had started with the 2nd character of q.

5 Summary

We've seen several 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 has a linear running time. 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.
- Suffix arrays. Here, the space is still linear, the constants are smaller, but more clever algorithms are needed to get linear-time searching.

Another approach is to use a classical string search algorithm:

- the Knuth-Morris-Pratt algorithm, for example, 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 time $O(|T_i|)$.

Each one its advantages.

6 Faster Suffix Array Construction*

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.

Note that this algorithm doesn't directly compute the LCP array. However, the LCP array can be constructed from the suffix array using Kasai's algorithm (see `https://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/`)

Below is an Ocml implementation of the suffix array construction algorithm:

```
(*
   An Ocaml implementation of the "merging blocks" algorithm
   to compute the suffix array in O(n log^2 n).

   Actually it might better be called the "surrogate" algorithm.
   Because during each pass it replaces a pair (i,j) with a single
   integer, the surrogate.  Comparing the surrogates for two pairs
   gives the same result as comparing the original pairs.  Applying
   this replacement k times means that the surrogate at position i
   represents the string of length 2^k starting at i, for purposes of
   comparison with any other substring of length 2^k.  And it works
   in constant time.

   This algorithm's performance can be improved to O(n log n) by
   replacing the sorting algorithm (in all but the first pass) with a
   2-pass radix sort, since in that case we're sorting pairs (i,j)
   where 0 <= i,j < n.

                    Danny Sleator, October 13, 2016
*)

let suffix_array a_in n =
  (* input is an array of length n of non-negative integers *)
  let small = -1 in  (* less than any number in the input array *)
  let a = Array.copy a_in in
  let w = Array.make n ((0,0),0) in
  let x = Array.make n 0 in  (* surrogates *)
  let rec pass shift =
    let get i = if i+shift < n then a.(i+shift) else small in
    for i=0 to n-1 do
      w.(i) <- ((a.(i), get i), i)
    done;
    Array.fast_sort compare w;
    for j=1 to n-1 do
      x.(j) <- x.(j-1) + (if fst w.(j-1) = fst w.(j) then 0 else 1)
    done;
    if x.(n-1) < n-1 then (
      for j=0 to n-1 do
        a.(snd w.(j)) <- x.(j)
      done;
      pass (shift*2)
    )
  in
  pass 1;
  Array.init n (fun i -> snd w.(i))

(*
   Example:
   Input:  [1, 1, 2, 2, 2, 2, 1, 1, 1, 1]
   Output: [9, 8, 7, 6, 0, 1, 5, 4, 3, 2]
*)
```

7 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.

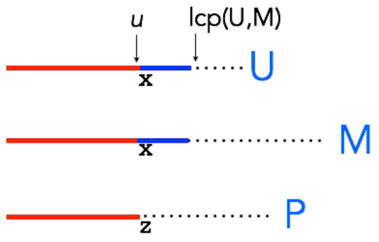
Using the $lcp(X, Y)$ values, we can avoid repeatedly comparing characters of P during our binary search. Recall that our binary search is maintaining a range $[U, D]$. Now we also maintain u and d:

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

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.

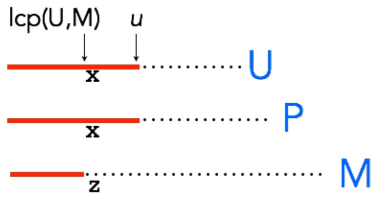
Suppose $u > d$. Let M be the midpoint of the range $[U, D]$. We have three cases that we can use to update the binary search range quickly:

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



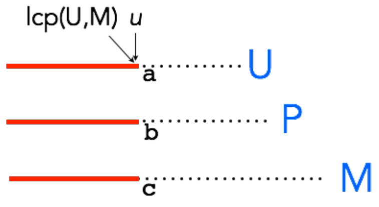
and we know that P comes after M. We need 0 new character comparisons to discover this. d and u are unchanged.

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 $u + 1 = d + 1$.

7.1 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.

7.2 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 of P 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.)

7.3 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, which you saw in an earlier homework.