In lecture 24 we introduced suffix arrays and the accompanying longest common prefix (LCP) array. In this lecture we give efficient algorithms for computing them.

1 Review: 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 n. 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 $O(n^2 \log n)$ time because each comparison of two strings takes $O(n)$ time. In fact, the suffix array can be constructed in $O(n)$ time. We will see faster construction algorithms in a minute.

Each successive suffix in sorted order matches the previous one in some number of (perhaps 0) letters. This is recorded in the longest common prefix lengths array, or the LCP array. In this case we have:

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

It's often very useful to have the LCP array available to solve string problems.

2 Computing the Suffix Array

The key idea to improve on the simple $O(n^2 \log n)$ 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 $O(n)$ time. In fact, it's possible to construct the array directly in $O(n)$ time. We will see a simpler $O(n \log n)$ 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 $s = a_0a_1a_3 \dots a_{n-1}$ 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 (the numbers on the the left will be explained in a moment):

```
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 as a sequence of digits:

```
cattcat$
21542130
```

But note: Comparing a pair of adjacent characters in the string to another adjacent pair can be done by comparing the digit codes of the two pairs. So, for example the code `at` is 1, the code for `ca` is 2, and `at` < `ca` and 1 < 2. So we say that 1 is a *surrogate* for `at`.

Therefore 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`. Using this we can proceed to build a code for 4-letter strings, etc.

So we repeat the process, now with a new "alphabet" of $\{0, \dots, n-1\}$. 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 $O(\log n)$th round.

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

We can reduce this to $O(n \log n)$ 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 $\{0, \ldots, n-1\}$, so we can sort in $O(n)$, removing one of the log factors.

Below is an Ocaml implementation of this suffix array construction algorithm.

```
let suffix_array a_in n =
  (* input is an array of length n of non-negative integers *)
  (* it returns (suffix_array, inverse_suffix_array) *)
  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 rec pass shift =
    (*  At this point a.(i) is a surrogate for a_in.(i, i+1, ... i+shift-1).
        That is, the result of comparing blocks of size shift in the input:

             a_in.(i,i+1,...i+shift-1)   to   a_in.(j,j+1,...j+shift-1)

        is the same as comparing a.(i) to a.(j)
    *)
    let geta j = if j < n then a.(j) else small in
    for i=0 to n-1 do
      w.(i) <- ((a.(i), geta (i+shift)), i)
    done;
    Array.fast_sort compare w;
    a.(snd w.(0)) <- 0;
    for j=1 to n-1 do
      a.(snd w.(j)) <- a.(snd w.(j-1)) + (if fst w.(j-1) = fst w.(j) then 0 else 1)
    done;
    if a.(snd w.(n-1)) < n-1 then pass (shift*2)
  in
  pass 1;
  (Array.init n (fun i -> snd w.(i)), a)
```

Here's a complete example of a run of this algorithm. Here's the input:

```
                     0  1  2  3  4  5  6
         a_in:      [3, 1, 8, 8, 3, 1, 8]
```

The algorithm forms the following sequence of triples. The first two elements of the triple are the elements from adjacent positions of `a_in`. The third element is the index from where the first one came from.

```
((3,1),0)  ((1,8),1)  ((8,8),2)  ((8,3),3)  ((3,1),4)  ((1,8),5)  ((8,-1),6)
```

Sorting these gives:

```
((1,8),1)  ((1,8),5)  ((3,1),0)  ((3,1),4)  ((8,-1),6)  ((8,3),3)  ((8,8),2)

    0          0          1          1           2           3          4
```

The line below shows the next surrogate numbering. It's obtained by starting with 0, and only incrementing when the pair is different from the previous one. Now we place these surrogate numbers into an array called `a` in the correct positions. This gives the following array:

```
    block size = 2  a = [1, 0, 4, 3, 1, 0, 2]
```

These are the surrogates for blocks of length 2. Now we repeat the same process again to make surrogates for blocks of length 4. The triples formed are:

```
  ((1,4),0)  ((0,3),1)  ((4,1),2)  ((3,0),3)  ((1,2),4)  ((0,-1),5)  ((2,-1),6)
```

Notice that the pairs are taken from positions 2 apart. Sorting and renumbering gives:

```
  ((0,-1),5) ((0,3),1)  ((1,2),4)  ((1,4),0)  ((2,-1),6) ((3,0),3)   ((4,1),2)

     0          1          2          3           4         5           6
```

Which leads to the following new surrogate array:

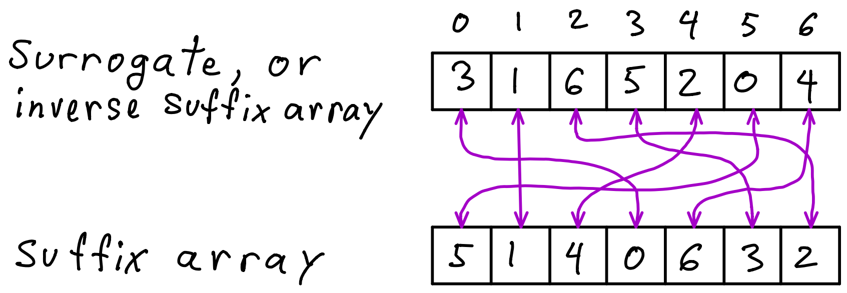
```
                        0  1  2  3  4  5  6
    block size = 4  a = [3, 1, 6, 5, 2, 0, 4]
```

This is a permutation of 0 through $n-1$ so there's no need for the algorithm to continue, Every suffix has now been uniquely placed in an ordering.

So these are the surrogates. To convert this into the suffix array we just rearrange it. So the suffix starting at index 5 is the smallest, so that becomes the first element of the suffix array. And the suffix starting at 1 is the second smallest, and the suffix starting at 4 is the third smallest, etc. So a linear scan gives us the suffix array:

```
      suffix array:     [5, 1, 4, 0, 6, 3, 2]
```

The following figure shows the relationship between the last surrogate array and the suffix array, both of which are output by the algorithm.

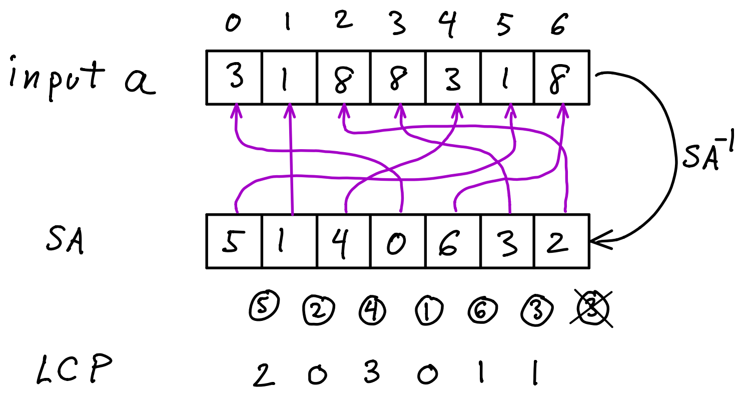


3 Computing the LCP array

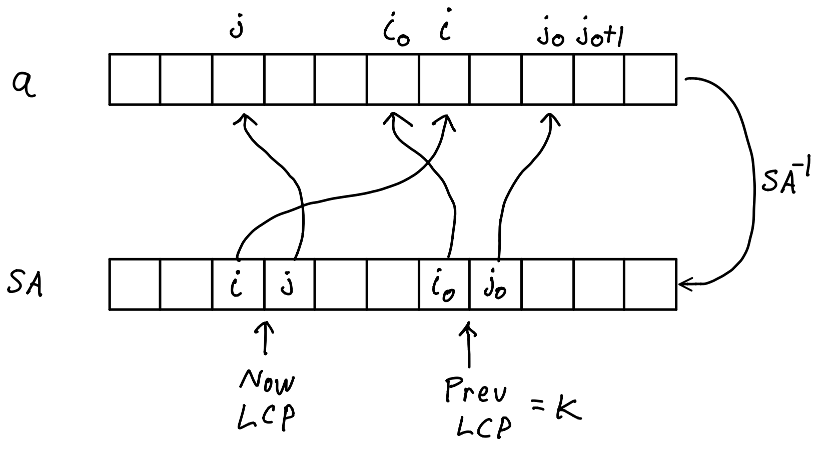
Let SA denote the suffix array computed above and $\mathtt{SA}^{-1}$ denote its inverse (also known as the final surrogate array) also computed above.

The simple brute-force algorithm to compute the LCP array is the following. Consider in turn each pair of consecutive elements of the suffix array. For each of them, look in the original array, and traverse right from those starting points until they differ. So, using the previous example, we'd start at positions SA[0]=5 and SA[1]=1 and see how many characters these prefixes have in common. In this case the answer is 2. Then continue with the suffixes beginning with 1 and 4 which have zero in common. Now suffixes beginning at 4 and 0 have 3 in common, etc. This algorithm is $O(n^2)$.

The trick to speeding this up is to compute these values in a different order. The order to use is left to right relative to the original input. So the following figure shows the order in which the LCP values should be computed for the previous example. The circled numbers below show the order in which the LCP is computed for the neighboring pairs in the suffix array.



Why is this useful? Suppose you've just computed the LCP of the suffixes beginning at positions (i_0, j_0) and that these suffixes have a common prefix of length k. The next one to compute is (i, j) where $i = i_0 + 1$, and $j = \mathtt{SA}[\mathtt{SA}^{-1}[i] + 1]$. The situation shown in the following figure.



Here's the key claim from which an efficient algorithm can be derived.

Claim: Suppose i_0, j_0 are two consecutive elements of the suffix array. Let $k = \text{LCP}(i_0, j_0)$. If $i = i_0 + 1$ and j is the successor of i in the suffix array then $\text{LCP}(i, j) \geq k - 1$.

Proof: First of all the statement is vacuously true unless $k > 1$, so let's take that as a given.

So we know that the suffix starting at i_0 (denoted suffix(i_0)) is less than the suffix starting at j_0, and that the two have a prefix of length $k > 1$ in common. Therefore the suffix starting at $i = i_0+1$ and the suffix starting at j_0+1 have $k-1 \geq 1$ characters in common, and $\text{suffix}(i) = \text{suffix}(i_0+1) < \text{suffix}(j_0+1)$. (suffix($i$) is the same as suffix($i_0$) with the first character removed. And suffix(j_0+1) is the same as suffix(j_0) with the first character removed.) If it is the case that $j = j_0+1$ our proof is done. On the other hand if $j \neq j_0 + 1$ we know the following facts about the ordering of these suffixes:

$$\text{suffix}(i) < \text{suffix}(j) < \text{suffix}(j_0 + 1)$$

The first inequality follows from the fact that i occurs before j in the suffix array. The second inequality follows from the fact that the suffix starting at j is *immediately* after the one starting at i (among all suffixes in the original string). Therefore it must be before any other suffix among those after i, including the one starting at $j_0 + 1$. Since suffix(i) and suffix($j_0 + 1$) have a common $k - 1$ character prefix, it follows that suffix(i) and suffix(j) have at least that many characters in common. ■

Given this claim, we can now give a linear time algorithm for computing the entire LCP array. For $i = 0, 1, \ldots, n-1$ we process the pair (i, j), where $j = \texttt{SA}[\texttt{SA}^{-1}[i] + 1]$. (In the case where $\texttt{SA}^{-1}[i] = n - 1$ we simply skip that i, because there is no LCP for that i since it's last in the suffix array.)

Initially $k = 0$. When the previous LCP value was k we know that the matching prefix is at least $k-1$ by the lemma. So we enter an inner loop trying to match more characters starting at $i+k-1$ and $j + k - 1$. Once a mismatch is found we record the common prefix length computed, and proceed by incrementing i.

How long can this take? Initially k is 0. It is reset to 0 once when $\texttt{SA}^{-1}[i] = n - 1$. It is decreased by 1 for each new i. Therefore its total decrease is at most $2n$. It's initially 0 and cannot exceed n. Therefore it can increase at most $3n$ times. Every time there's a mismatch, i advances (can happen at most n times). Every time there's a match k increases. (This can happen at most $3n$ times). Thus the algorithm is $O(n)$ time.

Here's the Ocaml code for this algorithm, which is called Kasai's algorithm.

```
let compute_LCP a n (suff, suff_inv) =
(* Kasai's algorithm.   Takes as input a suffix array, and its inverse
   (aka the final surrogate array), and returns the LCP array.
*)
  let lcp = Array.make (n-1) 0 in
  let rec loop i k = if i=n then lcp else
      if suff_inv.(i) = n-1 then loop (i+1) 0 else
        let j = suff.(suff_inv.(i)+1) in
        let rec scan k =
          if i+k < n && j+k < n && a.(i+k) = a.(j+k) then scan (k+1) else k
        in
        let k = scan k in
        lcp.(suff_inv.(i)) <- k;
        loop (i+1) (max (k-1) 0)
  in
  loop 0 0
```