LCA Preprocessing

The *lowest common ancestor* (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v. Any tree can be preprocessed in *linear time* so that the LCA of any two nodes can be computed in *constant time*. The details are omitted here.

- Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the *longest common prefix* of T_i and T_j. Thus the lcp of any two suffixes can be computed in *constant time* using the suffix tree with LCA preprocessing.

- The longest common prefix of two suffixes S_i and T_j from two different strings S and T is called the *longest common extension*. Using the generalized suffix tree with LCA preprocessing, the longest common extension for any pair of suffixes can be computed in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms use longest common extension data structures (see exercises).

Longest Palindrome

A palindrome is a string that is its own reverse. For example, `saippuakauppias` is a palindrome.

We can use the LCA preprocessed generalized suffix tree of a string T and its reverse T^R to find the *longest palindrome* in T in linear time.

- Let k_i be the length of the longest common extension of T_{i+1} and T^R_{n-i}, which can be computed in constant time. Then $T[i-k_i..i+k_i]$ is the longest odd length palindrome with the middle at i.
- We can find the longest odd length palindrome by computing k_i for all $i \in [0..n)$ in $\mathcal{O}(n)$ time.
- The longest even length palindrome can be found similarly in $\mathcal{O}(n)$ time. The longest palindrome overall is the longer of the two.

Suffix Array

The suffix array of a text T is a lexicographically ordered array of the set $T_{[0..n]}$ of all suffixes of T. More precisely, the suffix array is an array $SA[0..n]$ of integers containing a permutation of the set $[0..n]$ such that $T_{SA[0]} < T_{SA[1]} < \cdots < T_{SA[n]}$.

A related array is the *inverse suffix array* SA^{-1} which is the inverse permutation, i.e., $SA^{-1}[SA[i]] = i$ for all $i \in [0..n]$. The value $SA^{-1}[j]$ is the *lexicographical rank* of the suffix T_j

As with suffix trees, it is common to add the end symbol $T[n] = $ `$`. It has no effect on the suffix array assuming `$` is smaller than any other symbol.

Example 4.7: The suffix array and the inverse suffix array of the text $T =$ `banana$`.

i	$SA[i]$	$T_{SA[i]}$	j	$SA^{-1}[j]$	
0	6	`$`	0	4	`banana$`
1	5	`a$`	1	3	`anana$`
2	3	`ana$`	2	6	`nana$`
3	1	`anana$`	3	2	`ana$`
4	0	`banana$`	4	5	`na$`
5	4	`na$`	5	1	`a$`
6	2	`nana$`	6	0	`$`

Suffix array is much simpler data structure than suffix tree. In particular, the type and the size of the alphabet are usually not a concern.

- The size on the suffix array is $\mathcal{O}(n)$ on any alphabet.

- We will later see that the suffix array can be constructed in the same asymptotic time it takes to *sort the characters* of the text.

Suffix array construction algorithms are quite fast in practice too. For example, the fastest way to construct a suffix tree is to construct a suffix array first and then use it to construct the suffix tree. (We will see how in a moment.)

Suffix arrays are rarely used alone but are augmented with other arrays and data structures depending on the application. We will see some of them in the next slides.

Exact String Matching

As with suffix trees, *exact string matching* in T can be performed by a *prefix search* on the suffix array. The answer can be conveniently given as a *contiguous interval* $SA[b..e)$ that contains the suffixes with the given prefix. The interval can be found using string binary search.

- If we have the additional arrays $LLCP$ and $RLCP$, the result interval can be computed in $\mathcal{O}(|P| + \log n)$ time.
- Without the additional arrays, we have the same time complexity on average but the worst case time complexity is $\mathcal{O}(|P| \log n)$.
- We can then count the number of occurrences in $\mathcal{O}(1)$ time, list all occ occurrences in $\mathcal{O}(occ)$ time, or list a sample of k occurrences in $\mathcal{O}(k)$ time.

We will later see a quite different method for prefix searching called *backward search*.

LCP Array

Efficient string binary search uses the arrays $LLCP$ and $RLCP$. However, for many applications, the suffix array is augmented with the lcp array of Definition 1.8 (Lecture 2, slide 19). For all $i \in [1..n]$, we store

$$LCP[i] = lcp(T_{SA[i]}, T_{SA[i-1]})$$

Example 4.8: The LCP array for $T =$ `banana$`.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$
0	6		`$`
1	5	0	`a$`
2	3	1	`ana$`
3	1	3	`anana$`
4	0	0	`banana$`
5	4	0	`na$`
6	2	2	`nana$`

Using the solution of Exercise 3.1 (construction of compact trie from sorted array and LCP array), the suffix tree can be constructed from the suffix and LCP arrays in linear time.

However, many suffix tree applications can be solved using the suffix and LCP arrays directly. For example:

- The *longest repeating factor* is marked by the maximum value in the LCP array.
- The *number of distinct factors* can be compute by the formula

  $$\frac{n(n+1)}{2} + 1 - \sum_{i=1}^{n} LCP[i]$$

  since it equals the number of nodes in the uncompact suffix trie, for which we can use Theorem 1.10.
- Matching statistics of T with respect to S can be computed in linear time using the generalized suffix array of S and T (i.e., the suffix array of $S£T\$$) and its LCP array (exercise).

LCP Array Construction

The LCP array is easy to compute in linear time using the suffix array SA and its inverse SA^{-1}. The idea is to compute the lcp values by comparing the suffixes, but skip a prefix based on a known lower bound for the lcp value obtained using the following result.

Lemma 4.9: For any $i \in [0..n)$, $LCP[SA^{-1}[i+1]] \geq LCP[SA^{-1}[i]] - 1$

Proof. For each $j \in [0..n)$, let $\Phi(j) = SA[SA^{-1}[j] - 1]$. Then $T_{\Phi(j)}$ is the immediate lexicographical predecessor of T_j and $LCP[SA^{-1}[j]] = lcp(T_j, T_{\Phi(j)})$.

- Let $\ell = LCP[SA^{-1}[i]]$ and $m = LCP[SA^{-1}[i+1]]$. We want to show that $m \geq \ell - 1$. If $\ell = 0$, the claim is trivially true.
- If $\ell > 0$, then for some symbol c, $T_i = cT_{i+1}$ and $T_{\Phi(i)} = cT_{\Phi(i)+1}$. Thus $T_{\Phi(i)+1} < T_{i+1}$ and $lcp(T_{i+1}, T_{\phi(i)+1}) = lcp(T_i, T_{\Phi(i)}) - 1 = \ell - 1$.
- If $\Phi(i+1) = \Phi(i) + 1$, then $m = lcp(T_{i+1}, T_{\Phi(i+1)}) = lcp(T_{i+1}, T_{\Phi(i)+1}) = \ell - 1$.
- If $\Phi(i+1) \neq \Phi(i) + 1$, then $T_{\Phi(i)+1} < T_{\Phi(i+1)} < T_{i+1}$ and $m = lcp(T_{i+1}, T_{\Phi(i+1)}) \geq lcp(T_{i+1}, T_{\Phi(i)+1}) = \ell - 1$.

$\square$

The algorithm computes the lcp values in the order that makes it easy to use the above lower bound.

Algorithm 4.10: LCP array construction
Input: text $T[0..n]$, suffix array $SA[0..n]$, inverse suffix array $SA^{-1}[0..n]$
Output: LCP array $LCP[1..n]$

```
(1) ℓ ← 0
(2) for i ← 0 to n − 1 do
(3)     k ← SA⁻¹[i]
(4)     j ← SA[k − 1]        // j = Φ(i)
(5)     while T[i + ℓ] = T[j + ℓ] do ℓ ← ℓ + 1
(6)     LCP[k] ← ℓ
(7)     if ℓ > 0 then ℓ ← ℓ − 1
(8) return LCP
```

The time complexity is $\mathcal{O}(n)$:

- Everything except the while loop on line (5) takes clearly linear time.
- Each round in the loop increments ℓ. Since ℓ is decremented at most n times on line (7) and cannot grow larger than n, the loop is executed $\mathcal{O}(n)$ times in total.

RMQ Preprocessing

The *range minimum query* (RMQ) asks for the smallest value in a given range in an array. Any array can be preprocessed in linear time so that RMQ for any range can be answered in constant time.

In the LCP array, RMQ can be used for computing the lcp of any two suffixes.

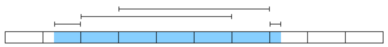
Lemma 4.11: The length of the *longest common prefix* of two suffixes $T_i < T_j$ is $lcp(T_i, T_j) = \min\{LCP[k] \mid k \in [SA^{-1}[i] + 1..SA^{-1}[j]]\}$.

The proof is left as an exercise.

- The RMQ preprocessing of the LCP array supports the same kind of applications as the LCA preprocessing of the suffix tree, but RMQ preprocessing is simpler than LCA preprocessing.
- The RMQ preprocessed LCP array can also replace the LLCP and RLCP arrays.

We will next describe the RMQ data structure for an arbitrary array $L[1..n]$ of integers.

- We precompute and store the minimum values for the following collection of ranges:
  - Divide $L[1..n]$ into blocks of size $\log n$.
  - For all $0 \leq \ell \leq \log(n/\log n))$, include all ranges that consist of 2^ℓ blocks. There are $\mathcal{O}(\log n \cdot \frac{n}{\log n}) = \mathcal{O}(n)$ such ranges.
  - Include all prefixes and suffixes of blocks. There are a total of $\mathcal{O}(n)$ of them.
- Now any range $L[i..j]$ that overlaps or touches a block boundary can be exactly covered by at most *four ranges* in the collection.



The minimum value in $L[i..j]$ is the minimum of the minimums of the covering ranges and can be computed in constant time.

Ranges $L[i..j]$ that are completely inside one block are handled differently.

- Let $NSV(i) = \min\{k > i \mid L[k] < L[i]\}$ (NSV=Next Smaller Value). Then the position of the minimum value in the range $L[i..j]$ is the last position in the sequence $i, NSV(i), NSV(NSV(i)), \ldots$ that is in the range. We call these the NSV positions for i.
- For each i, store the NSV positions for i up to the end of the block containing i as a bit vector $B(i)$. Each bit corresponds to a position within the block and is one if it is an NSV position. The size of $B(i)$ is $\log n$ bits and we can assume that it fits in a single machine word. Thus we need $\mathcal{O}(n)$ words to store $B(i)$ for all i.
- The position of the minimum in $L[i..j]$ is found as follows:
  - Turn all bits in $B(i)$ after position j into zeros. This can be done in constant time using bitwise shift -operations.
  - The right-most 1-bit indicates the position of the minimum. It can be found in constant time using a lookup table of size $\mathcal{O}(n)$.

All the data structures can be constructed in $\mathcal{O}(n)$ time (exercise).

Enhanced Suffix Array

The enhanced suffix array adds two more arrays to the suffix and LCP arrays to make the data structure fully equivalent to suffix tree.

- The idea is to represent a suffix tree node v representing a factor S_v by the suffix array interval of the suffixes that begin with S_v. That interval contains exactly the suffixes that are in the subtree rooted at v.

- The additional arrays support navigation in the suffix tree using this representation: one array along the regular edges, the other along suffix links.

With all the additional arrays the suffix array is not very space efficient data structure any more. Nowadays suffix arrays and trees are often replaced with **compressed text indexes** that provide the same functionality in much smaller space.