AC Automaton for the Set of Suffixes

As already mentioned, a suffix tree with suffix links is essentially an Aho–Corasick automaton for the set of all suffixes.

- We saw that it is possible to follow suffix link / failure transition from any locus, not just from suffix tree nodes.
- Following such an implicit suffix link may take more than a constant time, but the total time during the scanning of a string with the automaton is linear in the length of the string. This can be shown with a similar argument as in the construction algorithm.

Thus suffix tree is asymptotically as fast to operate as the AC automaton, but needs much less space.

Matching Statistics

The matching statistics of a string $S[0..n)$ with respect to a string T is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

1. $S[i..i+\ell_i)$ is the longest prefix of S_i that is a factor of T, and
2. $T[p_i..p_i+\ell_i) = S[i..i+\ell_i)$.

Matching statistics can be computed by using the suffix tree of T as an AC-automaton and scanning S with it.

- If before reading $S[i]$ we are at the locus (v, d) in the automaton, then $S[i-d..i) = T[j..j+d)$, where $j = \mathit{start}(v)$. If reading $S[i]$ causes a failure transition, then $MS[i-d] = (d, j)$.
- Following the failure transition decrements d and thus increments $i-d$ by one. Following a normal transition/edge, increments both i and d by one, and thus $i-d$ stays the same. Thus all entries are computed.

From the matching statistics, we can easily compute the longest common factor of S and T. Because we need the suffix tree only for T, this saves space compared to a generalized suffix tree.

Matching statistics are also used in some approximate string matching algorithms.

Longest Common Extension

The longest common extension (LCE) query asks for the length of the longest common prefix of two suffixes of a text T:

$$LCE(i, j) := lcp(T_i, T_j)$$

- The lowest common ancestor (LCA) of two nodes u and v in a tree 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.
- A LCE query can be implemented as a LCA query on the suffix tree of T:

  $$LCE(i, j) = LCA(w_i, w_j)$$

  where w_i and w_j are the leaves that represent the suffixes T_i and T_j. Thus, given the suffix tree augmented with a precomputed LCA data structure, LCE queries can be answered in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms are based on LCE queries.

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

j	$SA^{-1}[j]$	
0	4	`banana$`
1	3	`anana$`
2	6	`nana$`
3	2	`ana$`
4	5	`na$`
5	1	`a$`
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. Probably 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 $\mathcal{O}(|P| + \log n)$ average time complexity, and we can achieve $\mathcal{O}(|P| \log_{|P|} n)$ worst case time with the skewed string binary search (Algorithm 1.40), and even better with a more complicated algorithm (see slide 59).
- 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.

An alternative algorithm for computing the interval $SA[b..e)$ is called backward search. It is commonly used with compressed representations of suffix arrays.

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 1.11 (Lecture 2). 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 2.5 (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 1.17.
- Matching statistics of S with respect to T 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).

Range Mimimum Queries

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.

We can answer longest common extension (LCE) queries using RMQ queries on the LCP array:

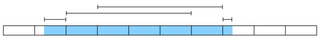
Lemma 4.9: The length of the longest common prefix of two suffixes $T_i < T_j$ is $lcp(T_i, T_j) = \min\{LCP[k] \mid k \in [SA^{-1}[i] + 1..SA^{-1}[j]]\}$.

The lemma can be seen as a generalization of Lemma 1.31(b) (Lecture 3) and holds for any sorted array of strings. The proof is left as an exercise.

- In addition to the many general applications of LCE queries, we can also replace the LLCP and RLCP arrays in binary searching.

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 $0 \le \ell \le \log(n/\log n)$), 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 crosses 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 $i, NSV(i), NSV(NSV(i)), \dots$ 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.

Burrows–Wheeler Transform

The Burrows–Wheeler transform (BWT) is an important technique for text compression, text indexing, and their combination compressed text indexing.

Let $T[0..n]$ be the text with $T[n] = \$$. For any $i \in [0..n]$, $T[i..n]T[0..i)$ is a rotation of T. Let $\mathcal{M}$ be the matrix, where the rows are all the rotations of T in lexicographical order. All columns of $\mathcal{M}$ are permutations of T. In particular:

- The first column F contains the text characters in order.
- The last column L is the BWT of T.

Example 4.10: The BWT of $T = \texttt{banana\$}$ is $L = \texttt{annb\$aa}$.

F						L
\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

Here are some of the key properties of the BWT.

- The BWT is easy to compute using the suffix array:

$$L[i] = \begin{cases} \$ & \text{if } SA[i] = 0 \\ T[SA[i] - 1] & \text{otherwise} \end{cases}$$

- The BWT is invertible, i.e., T can be reconstructed from the BWT L alone. The inverse BWT can be computed in the same time it takes to sort the characters.
- The BWT L is typically easier to compress than the text T. Many text compression algorithms are based on compressing the BWT.
- The BWT supports backward searching, a different technique for indexed exact string matching. This is used in many compressed text indexes.

Inverse BWT

Let $\mathcal{M}'$ be the matrix obtained by rotating $\mathcal{M}$ one step to the right.

Example 4.11:

$\mathcal{M}$

\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

$\xrightarrow{\text{rotate}}$

$\mathcal{M}'$

a	\$	b	a	n	a	n
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a
b	a	n	a	n	a	\$
\$	b	a	n	a	n	a
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b

- The rows of $\mathcal{M}'$ are the rotations of T in a different order.
- In $\mathcal{M}'$ without the first column, the rows are sorted lexicographically. If we sort the rows of $\mathcal{M}'$ stably by the first column, we obtain $\mathcal{M}$.

This cycle $\mathcal{M} \xrightarrow{\text{rotate}} \mathcal{M}' \xrightarrow{\text{sort}} \mathcal{M}$ is the key to inverse BWT.

Consider what happens to a column j in one round of this cycle:

- Rotation moves the column to the right and it becomes the column $j+1$ in matrix $\mathcal{M}'$.
- Sorting permutes the column and makes it the column $j+1$ in matrix $\mathcal{M}$.

Thus if we know column j, we can obtain column $j+1$ by permuting column j.

The same permutation also transforms the last column (the BWT) into the first column (the sorted sequence).

Thus we can reconstruct the matrix $\mathcal{M}$ from the BWT:

- Determine the permutation that stably sorts the BWT, i.e., that transforms the last column into the first column.
- Obtain the second column by permuting the first column, the third column by permuting the second column, etc.

To reconstruct T, we do not need to compute the whole matrix but just keep track of one row.

Example 4.12:

```
------a          a------          $-----a          a$-----          $b----a
------n          n------          a-----n          na-----          a$----n
------n  rotate  n------   sort   a-----n  rotate  na-----   sort   an----n
------b   -->    b------   -->    a-----b   -->    ba-----   -->    an----b
------$          $------          b-----$          $b-----          ba----$
------a          a------          n-----a          an-----          na----a
------a          a------          n-----a          an-----          na----a

          $ba---a          $ban--a          $bana-a          $banana
          a$b---n          a$ba--n          a$ban-n          a$banan
 rotate   ana---n  rotate  ana$--n  rotate  ana$b-n  rotate  ana$ban
 & sort   ana---b  & sort  anan--b  & sort  anana-b  & sort  anana$b
  -->     ban---$   -->    bana--$   -->    banan-$   -->    banana$
          na$---a          na$b--a          na$ba-a          na$bana
          nan---a          nana--a          nana$-a          nana$ba
```


The permutation that transforms $\mathcal{M}'$ into $\mathcal{M}$ is called the LF-mapping.

- LF-mapping is the permutation that stably sorts the BWT L, i.e., $F[LF[i]] = L[i]$. Thus it is easy to compute from L.
- Given the LF-mapping, we can easily follow a row through the permutations.

Algorithm 4.13: Inverse BWT
Input: BWT $L[0..n]$
Output: text $T[0..n]$
Compute LF-mapping:
  for $i \leftarrow 0$ **to** n **do** $R[i] = (L[i], i)$
  sort R (stably by first element)
  for $i \leftarrow 0$ **to** n **do**
    $(\cdot, j) \leftarrow R[i]$; $LF[j] \leftarrow i$
Reconstruct text:
  $j \leftarrow$ position of \$ in L
  for $i \leftarrow n$ **downto** 0 **do**
    $T[i] \leftarrow L[j]$
    $j \leftarrow LF[j]$
  return T

Everything works in linear time with the possible exception of the sorting.

On Burrows-Wheeler Compression

The basic principle of text compression is that, the more frequently a factor occurs, the shorter its encoding should be.

Let c be a symbol and w a string such that the factor cw occurs frequently in the text.

- The occurrences of cw may be distributed all over the text, so recognizing cw as a frequently occurring factor is not easy. It requires some large, global data structures.
- In the BWT, the high frequency of cw means that c is frequent in that part of the BWT that corresponds to the rows of the matrix $\mathcal{M}$ beginning with w. This is easy to recognize using local data structures.

This localizing effect makes compressing the BWT much easier than compressing the original text.

Text compression is covered in more detail on the course Data Compression Techniques.

Example 4.14: A part of the BWT of a reversed english text corresponding to rows beginning with `ht`:

```
oreeereoeeieeeeaooeeeeeaereeeeeeeeeeeeereeeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeeereeeeeeeeeeeieaeeeeieeeeaeeeeeeeeeeeieeeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeeaeeee
eeaeeeeeeereeeaeeeeeieeeeeeeeeiieee. e  eeeeiiiiii e          ,
i   o       oo e  eiiiiee,er  ,  ,     ,  . iii
```

and some of those symbols in context:

```
t raise themselves, and the hunter, thankful and r
ery night it flew round the glass mountain keeping
agon, but as soon as he threw an apple at it the b
f animals, were resting themselves.  "Halloa, comr
ple below to life.  All those who have perished on
 that the czar gave him the beautiful Princess Mil
ng of guns was heard in the distance.  The czar an
cked magician put me in this jar, sealed it with t
o acted as messenger in the golden castle flew pas
u have only to say, 'Go there, I know not where; b
```


Backward Search

Let $P[0..m)$ be a pattern and let $[b..e)$ be the suffix array range corresponding to suffixes that begin with P, i.e., $SA[b..e)$ contains the starting positions of P in the text T. Earlier we noted that $[b..e)$ can be found by binary search on the suffix array.

Backward search is a different technique for finding this range. It is based on the observation that $[b..e)$ is also the range of rows in the BWT matrix $\mathcal{M}$ beginning with P.

Let $[b_i, e_i)$ be the range for the pattern suffix $P_i = P[i..m)$. The backward search will first compute $[b_{m-1}, e_{m-1})$, then $[b_{m-2}, e_{m-2})$, etc. until it obtains $[b_0, e_0) = [b, e)$. Hence the name backward search.

Backward search uses the following data structures:

- An array $C[0..\sigma)$, where $C[c] = \left|\{i \in [0..n] \mid L[i] < c\}\right|$. In other words, $C[c]$ is the number of occurrences of symbols that are smaller than c.
- The function $rank_L : \Sigma \times [0..n+1] \to [0..n]$:
$$rank_L(c, j) = \left|\{i \mid i < j \text{ and } L[i] = c\}\right| .$$
In other words, $rank_L(c, j)$ is the number of occurrences of c in L before position j.

These data structures are closely related to the LF-mapping:

- $C[L[i]]$ is the number of symbols that are smaller than $L[i]$.
- $rank_L(L[i], i)$ is the number symbols equal to $L[i]$ that occur before $L[i]$.
- Those are exactly the symbols preceding $L[i]$ when sorted stably. Thus $LF[i] = C[L[i]] + rank_L(L[i], i)$.

In backward search, we need to compute the range $[b_i, e_i)$ from the range $[b_{i+1}, e_{i+1})$. This is done separately for each end of the range.

Given b_{i+1}, we can compute b_i as follows.

- Recall that b_i is the first row in $\mathcal{M}$ beginning with $P_i = P[i..m)$, i.e., the number of rows that are lexicographically smaller than P_i.
- $C[P[i]]$ is the number of rows beginning with a symbol smaller than $P[i]$.
- To $C[P[i]]$ we need to add the number of rows that begin with $P[i]$ and are lexicographically smaller than P_i.
- $rank_L(P[i], b_{i+1})$ is the number of rows that are lexicographically smaller than P_{i+1} and contain $P[i]$ at the last column. Rotating these rows one step to the right, we obtain the rotations of T that begin with $P[i]$ and are lexicographically smaller than $P[i]P_{i+1} = P_i$.
- Thus $b_i = C[P[i]] + rank_L(P[i], b_{i+1})$.

Computing e_i from e_{i+1} is similar: $e_i = C[P[i]] + rank_L(P[i], e_{i+1})$.

Algorithm 4.15: Backward Search
Input: array C, function $rank_L$, pattern P
Output: suffix array range $[b..e)$ containg starting positions of P

$b \leftarrow 0$; $e \leftarrow n + 1$
for $i \leftarrow m - 1$ **downto** 0 **do**
    $c \leftarrow P[i]$
    $b \leftarrow C[c] + rank_L(c, b)$
    $e \leftarrow C[c] + rank_L(c, e)$
return $[b..e)$

- The array C requires an integer alphabet that is not too large.

- The trivial implementation of the function $rank_L$ as an array requires $\Theta(\sigma n)$ space, which is often too much. There are much more space efficient (but slower) implementations. There are even implementations with a size that is close to the size of the compressed text. Such an implementation is the key component in many compressed text indexes. These are covered in the course Data Compression Techniques.