Generalized Suffix Tree
A generalized suffix tree of two strings $S$ and $T$ is the suffix tree of the string $S£T$, where $£$ and $\$$ are symbols that do not occur elsewhere in $S$ and $T$.

Each leaf is marked as an $S$-leaf or a $T$-leaf according to the starting position of the suffix it represents. Using a depth first traversal, we determine for each internal node if its subtree contains only $S$-leaves, only $T$-leaves, or both. The deepest node that contains both represents the longest common factor of $S$ and $T$. It can be computed in linear time.

The generalized suffix tree can also be defined for more than two strings.

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 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 $(v, d)$ such that

1. $S[i..i+d)$ is the longest prefix of $S$ that is a factor of $T$, and
2. $T[p..p+d) = S[i..i+d)$.

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 $S[i..i+d) = T[j..j+d]$, where $j = start(v)$. If reading $S[i]$ causes a failure transition, then $MS[i] = (d, j)$.
- Following the failure transition increments $d$ 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 Palindrome
A palindrome is a string that is its own reverse. For example, saippuskaappias 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 $T_{i+k_i}$ and $T_{n-i}$ which can be computed in constant time. Then $T[i..i+k_i]$ 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 $O(n)$ time.
- The longest even length palindrome can be found similarly in $O(n)$ time. The longest palindrome overall is the longer of the two.

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 $O(n)$ 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.

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 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 $(v, d)$ such that

1. $S[i..i+d)$ is the longest prefix of $S$ that is a factor of $T$, and
2. $T[p..p+d) = S[i..i+d)$.

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 $S[i..i+d) = T[j..j+d]$, where $j = start(v)$. If reading $S[i]$ causes a failure transition, then $MS[i] = (d, j)$.
- Following the failure transition increments $d$ 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 Palindrome
A palindrome is a string that is its own reverse. For example, saippuskaappias 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 $T_{i+k_i}$ and $T_{n-i}$ which can be computed in constant time. Then $T[i..i+k_i]$ 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 $O(n)$ time.
- The longest even length palindrome can be found similarly in $O(n)$ time. The longest palindrome overall is the longer of the two.

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 $O(n)$ 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.
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 1.10 (Lecture 2). For all $i \in [1..n]$, we store
$$\text{LCP}[i] = \text{lg}(T\text{SA}(i,j) - T\text{SA}(i-1,j))$$

Example 4.8: The LCP array for $T = \text{banana}$. 

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{SA}(i)$</th>
<th>$\text{LCP}[i]$</th>
<th>$T\text{SA}(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
<td>$\text{banana}$</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>ana</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>anana</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>banana</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>na</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>nana</td>
</tr>
</tbody>
</table>

Range Minimum 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 $\min(\text{LCP}[i], \text{LCP}[j])$. The lemma can be seen as a generalization of Lemma 1.30(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 LLC and RLCP arrays in binary searching.

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

- Let $\text{NSV}(i) = \min\{k > i | L[k] < L[i]\}$ (NSV = Next Smaller Value).
- 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 $O(n)$ 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 $O(n)$.

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

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[0] = \$. For any $i \in [0..n]$, $T[i..n]T[0..i]$ is a rotation of $T$. Let $M$ be the matrix, where the rows are all the rotations of $T$ in lexicographical order. All columns of $M$ 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 $T = \text{banana}$ is $L = \text{anambna}$.

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 computed by the formula
  $$\frac{n(n + 1)}{2} + 1 - \sum_{i=1}^{n} \text{LCP}[i]$$
  since it equals the number of nodes in the uncompact suffix trie, for which we can use Theorem 1.16.
- 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 $SETS$) and its LCP array (exercise).

Enhanced Suffix Array
The enhanced suffix array adds two more arrays to the suffix and LCP arrays to make the data structure even more efficient.

- 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 anymore. Nowadays suffix arrays and trees are often replaced with compressed text indexes that provide the same functionality in much smaller space.
Inverse BWT

Let $M'$ be the matrix obtained by rotating $M$ one step to the right.

**Example 4.11:**

<table>
<thead>
<tr>
<th>$M$</th>
<th>$M'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b a n a n a$</td>
<td>$a b a n a n$</td>
</tr>
<tr>
<td>$a b a n a b a$</td>
<td>$a b a n a b a$</td>
</tr>
<tr>
<td>$a a b a n a a$</td>
<td>$b a b a a a$</td>
</tr>
<tr>
<td>$a n a b a n a$</td>
<td>$b a a a a a$</td>
</tr>
<tr>
<td>$a n a a b a a$</td>
<td>$a a a a a a$</td>
</tr>
<tr>
<td>$a a a a a a$</td>
<td>$a a a a a a$</td>
</tr>
<tr>
<td>$a a a a a a$</td>
<td>$a a a a a a$</td>
</tr>
</tbody>
</table>

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

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

The permutation that transforms $M'$ into $M$ is called the LF-mapping.

- LF-mapping is the permutation that stably sorts the BWT, 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:** $BWT[0..n]$

**Output:** text $T[0..n]$

**Compute LF-mapping:**
1. for $i \leftarrow 0$ to $n$ do $R[i] = [L[i], i]$
2. sort $R$ (stably by first element)
3. for $i \leftarrow 0$ to $n$ do $(c, j) \leftarrow R[i]; LF[j] \leftarrow i$

**Reconstruct text:**
4. for $i \leftarrow n$ downto 0 do
5. $(j) \leftarrow L[i]$
6. $T[j] \leftarrow LF[j]$
7. $(j) \leftarrow LF[j]$
8. 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 $M$ 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.

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., $S[b..e]$ 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 $M$ beginning with $P$.

Let $[b..e]$ be the range for the pattern suffix $P = P[i..m]$. The backward search will first compute $[b_0, e_0]$, then $[b_2, e_2]$, etc. until it obtains $[b_0, e_0] = [b, e]$. Hence the name backward search.

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 $M'$.
- Sorting permutes the column and makes it the column $j + 1$ in matrix $M$.

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).

The permutation that transforms $M'$ into $M$ 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.

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

<table>
<thead>
<tr>
<th>$ht$</th>
<th>$h$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$h$</td>
<td>$h$</td>
<td>$h$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

and some of those symbols in context:

| $t$ raise themselves, and th0 hunter, thankf and r ery night in flew round th0 glass mountain keeping agon, but as soon as he threw an apple at it the b f animals, were resting themselves. "Halloa, com e to live. All th0se who have perch0n on that the czar gave him th0 beautiful Princess Mil ng of guns was heard in th0 distance. The czar an cked magician put me in this jar, sealed it with t o acted as messenger in th0 golden castle flew pas u have only to say, 'Go th0re, I know nct where; b

Backward search uses the following data structures:

- An array $C[0..n]$, where $C[i] = [i \in [0..n] | L[i] < c]$. In other words, $C[i]$ is the number of occurrences of symbols that are smaller than $c$.
- The function $rank_k : \Sigma \times [0..n] \rightarrow [0..n]$

$$rank_k(c, j) = |\{i \in [0..j] | L[i] = c\}$$

In other words, $rank_k(c, j)$ 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_k(L[i], i) is the number symbols preceding $L[i]$ that occur before $L[i]$. Those are exactly the symbols preceding $L[i]$ when sorted stably. Thus $LF[i] = C[L[i]] + rank_k(L[i], i)$.
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 \(M\) 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 \(P_i\).
- \(\text{rank}_L(P[i], b_{i+1})\) 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 \(b_i = C[P[i]] + \text{rank}_L(P[i], b_{i+1})\).

Computing \(e_i\) from \(e_{i+1}\) is similar: \(e_i = C[P[i]] + \text{rank}_L(P[i], e_{i+1})\).

**Algorithm 4.15: Backward Search**

**Input:** array \(C\), function \(\text{rank}_L\), pattern \(P\)

**Output:** suffix array range \([b..e)\) containing starting positions of \(P\)

1. \(b \leftarrow 0; e \leftarrow n + 1\)
2. for \(i \leftarrow m - 1\) downto 0 do
3. \(c \leftarrow P[i]\)
4. \(b \leftarrow C[c] + \text{rank}_L(c, b)\)
5. \(e \leftarrow C[c] + \text{rank}_L(c, e)\)
6. return \([b..e)\)

- The array \(C\) requires an integer alphabet that is not too large.
- The trivial implementation of the function \(\text{rank}_L\) 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.