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.

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

\[
LCE(i, j) := lcp(T, T_j)
\]

- 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 $O(kn)$ worst case time approximate string matching algorithms are based on LCE queries.

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

\[
T[SA[0]] < T[SA[1]] < \ldots < T[SA[n]].
\]

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 $SA^{-1}[i]$ is the
lexicographical rank of the suffix $T_i$.

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 $T = \text{banana}$. $\$

<table>
<thead>
<tr>
<th>i</th>
<th>$SA[i]$</th>
<th>$T[SA[i]]$</th>
<th>$SA^{-1}[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>$$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$a$</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$\text{ana}$</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>banana</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>$n$</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>nana</td>
<td>6</td>
</tr>
</tbody>
</table>

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 $O(|P| + \log n)$ time.
- Without the additional arrays, we have $O(|P| + \log n)$ average time
complexity, and we can achieve $O(|P|\log \log n)$ 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 $O(1)$ time, list all
  $occ$ occurrences in $O(occ)$ time, or list a sample of $k$ occurrences in $O(k)$
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.

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 $\langle i, p_i \rangle$ such that

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

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
  $S[i - d..i) = T[j + d, j + d]$, where $j = \text{start}(v)$. 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 Palindrome
A palindrome is a string that is its own reverse. For example,
$\text{sappkkaupia}$ 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
  $T_{i+k_i}$, which can be computed in constant time. Then $T[i - k_{i-1}..i + k_i]$ 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.

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 $T = \text{banana}$. $\$

<table>
<thead>
<tr>
<th>i</th>
<th>$SA[i]$</th>
<th>$LCP[i]$</th>
<th>$T[SA[i]]$</th>
</tr>
</thead>
</table>
| 0 | 0 | $\$ | $\$
| 1 | 5 | U | $a$ |
| 2 | 3 | 1 | $\text{ana}$ |
| 3 | 3 | 3 | $\text{ana}$ |
| 4 | 0 | 0 | $\text{banana}$ |
| 5 | 4 | 0 | $n$ |
| 6 | 2 | 2 | $\text{nana}$ |

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.
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 tree, 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).

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 \( 0 \leq i \leq (\log n / \log n) \), include all ranges that consist of \( 2^i \) blocks. There are \( O(\log n \log n) = O(n) \) such ranges.
  - Include all prefixes and suffixes of blocks. There are a total of \( O(n) \) 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.

### 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 \( S_i \) by the suffix array interval of the suffixes that begin with \( S_i \). 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.

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.

### 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 lexicographic 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 \( L = \text{annab}\$\).

#### Example 4.11:

Let \( M' \) be the matrix obtained by rotating \( M \) one step to the right.

- 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 \( M \xrightarrow{\text{rotate}} M' \xrightarrow{\text{sort}} M \) 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 \( M' \).
- 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).

Thus we can reconstruct the matrix \( M \) 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:**

```
<table>
<thead>
<tr>
<th>Original</th>
<th>Rotate</th>
<th>Sort</th>
<th>New BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a a a</td>
<td>a a a a</td>
<td>a a a a</td>
<td>a a a a</td>
</tr>
<tr>
<td>b b b b</td>
<td>b b b b</td>
<td>b b b b</td>
<td>b b b b</td>
</tr>
<tr>
<td>c c c c</td>
<td>c c c c</td>
<td>c c c c</td>
<td>c c c c</td>
</tr>
</tbody>
</table>
```

Thus we can reconstruct the matrix \( M \) 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.

**Algorithm 4.13:** Inverse BWT

Input: BWT \( L[0..n] \)

Output: text \( T[0..n] \)

Compute LF-mapping:

1. \( \text{for } i \leftarrow 0 \text{ to } n \text{ \text{do} } R[i] = L[i], i \)
2. \( \text{sort } R \) (stably by first element)
3. \( \text{for } i \leftarrow 0 \text{ to } n \text{ \text{do} } L[i] = R[i] \)

Reconstruct text:

1. \( j \leftarrow \text{position of } e \text{ in } L \)
2. \( \text{for } i \leftarrow n \text{ downto } 0 \text{ \text{do} } T[i] = L[i] \)
3. \( i \leftarrow L[i] \)
4. \( j \leftarrow L[i] \)
5. \( T[j] = L[i] \)
6. \( \text{return } T \)

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

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

```
the golden castle flew past the beautiful Princess Mil
```

and some of those symbols in context:

```
t raise themselves, and the hunt is thankfu and r
every night it flew round the mountains keeping
```

```
and as soon as he thought an apple at it the b
d animals, were resting themselves. "halloa, coa
```

```
the czar gave him the golden castle flew past u
```

```
Go there, I know not where; b
```

```
and of rains were heard in the distance. The cza
```

```
and the only thing they had on was the crown j
```

```
the czar an
```

**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 \( P[m] \) be a pattern and let \( [b, c] \) be the suffix array range corresponding to suffixes that begin with \( P \), i.e., \( S[1..b, c) \) contains the starting positions of \( P \) in the text \( T \). Earlier we noted that \( [b, c] \) 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 \( [b, c] \) is also the range of rows in the BWT matrix \( M \) beginning with \( P \).

Let \( [b, c] \) be the range for the pattern suffix \( P = P[m] \). The backward search will first compute \( [b_0, c_0) \), then \( [b_{n-2}, c_{n-2}) \), etc. until it obtains \( [b_n, c_n) = [b, c] \). Hence the name backward search.

In backward search, we need to compute the range \( [b_i, c_i] \) from the range \( [b_{i+1}, c_{i+1}] \). 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 \( P[0..i] \), 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 \( P_i \).
- \( rank_k(P[i], b_{i+1}) \) 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 \( P[i]P_{i+1} = P \).
- Thus \( b_i = C[P[i]] + rank_k(P[i], b_{i+1}) \).

Computing \( e_i \) from \( c_{i+1} \) is similar: \( e_i = C[P[i]] + rank_k(P[i], c_{i+1}) \).
Algorithm 4.15: Backward Search
Input: array $C$, function $rank_L$, 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) $b \leftarrow C[c] + rank_L(c,b)$
(5) $e \leftarrow C[c] + rank_L(c,e)$
(6) return $[b,e)$

- The array $C$ requires an integer alphabet that is not too large.
- The trivial implementation of the function $rank_L$ 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.