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 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 1.7 (Lecture 2, slide 21). 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 1.9.

- 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).



## 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 $i \in [0..n)$, $LCP[SA^{-1}[i]] \geq LCP[SA^{-1}[i-1]] - 1$

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

□



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 $\ell$. Since $\ell$ 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 lemma can be seen as a generalization of Lemma 1.25 and holds for any sorted array of strings. 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 $2^\ell$ 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.



## 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 4.12:** The BWT of $T =$ `banana$` is $L =$ `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 4.13:**

| $\mathcal{M}$ | | | | | | | | $\mathcal{M}'$ | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $ | b | a | n | a | n | a | rotate → | a | $ | b | a | n | a | n |
| a | $ | b | a | n | a | n | | n | a | $ | b | a | n | a |
| a | n | a | $ | b | a | n | | n | a | n | a | $ | b | a |
| a | n | a | n | a | $ | b | | b | a | n | a | n | a | $ |
| b | a | n | a | n | a | $ | | $ | b | a | n | a | n | a |
| n | a | $ | b | a | n | a | | a | n | a | $ | b | a | n |
| n | a | n | a | $ | b | a | | a | n | a | n | a | $ | b |

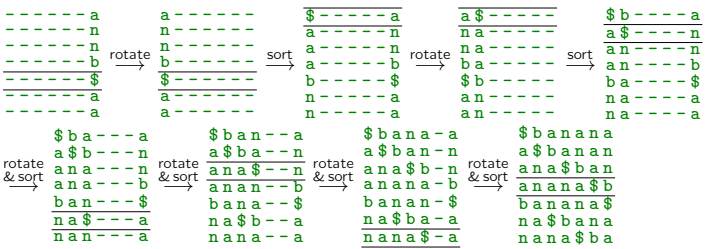
- The rows of $\mathcal{M}'$ are the rotations of $T$ in a different order.
- In $\mathcal{M}'$ without the first column, the rows are sorted lexicographically. If we sort the rows of $\mathcal{M}'$ stably by the first column, we obtain $\mathcal{M}$.

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



- In the cycle, each column moves one step to the right and is then permuted. The permutation is fully determined by the last column of $\mathcal{M}$, i.e., the BWT.
- Thus if we know column $j$, we can obtain column $j+1$ by permuting column $j$. By repeating this, we can reconstruct $\mathcal{M}$.
- To reconstruct $T$, we do not need to compute the whole matrix just one row.

**Example 4.14:**






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 4.15:** Inverse BWT
Input: BWT $L[0..n]$
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
(4) $\quad (\cdot, j) \leftarrow R[i]$; $LF[j] \leftarrow i$
Reconstruct text:
(5) $j \leftarrow$ position of $ in $L$
(6) for $i \leftarrow n$ downto 0 do
(7) $\quad T[i] \leftarrow L[j]$
(8) $\quad j \leftarrow LF[j]$
(9) return $T$

The time complexity is dominated by the stable 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.

We will not go deeper into text compression on this course.

**Example 4.16:** A part of the BWT of a reversed english text corresponding to rows beginning with `ht`:

```
oreeereoeeieeeeaooeeeeeaereeeeeeeeeeeeereeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeeeereeeeeeeeeeeieaeeeeieeeeaeeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeaeeee
eeaeeeeeereeeaeeeeeieeeeeeeeiieee. 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 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 $\mathit{rank}_L : \Sigma \times [0..n+1] \to [0..n]$:

  $$\mathit{rank}_L(c, j) = \left|\{i \mid i < j \text{ and } L[i] = c\}\right| .$$

  In other words, $\mathit{rank}_L(c, j)$ is the number of occurrences of $c$ in $L$ before position $i$.

Given $b_{i+1}$, we can now compute $b_i$ as follows. Computing $e_i$ from $e_{i+1}$ is similar.

- $C[P[i]]$ is the number of rows beginning with a symbol smaller than $P[i]$. Thus $b_i \geq C[P[i]]$.
- $\mathit{rank}_L(P[i], b_{i+1})$ 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 $P_i = P[i]P_{i+1}$.
- Thus $b_i = C[P[i]] + \mathit{rank}_L(P[i], b_{i+1})$.



**Algorithm 4.17:** Backward Search
Input: array $C$, function $\mathit{rank}_L$, pattern $P$
Output: suffix array range $[b..e)$ containg starting positions of $P$

```
(1) b ← 0; e ← n + 1
(2) for i ← m − 1 downto 0 do
(3)     c ← P[i]
(4)     b ← C[c] + rank_L(c, b)
(5)     e ← 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 $\mathit{rank}_L$ 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.



## Suffix Array Construction

Suffix array construction means simply sorting the set of all suffixes.

- Using standard sorting or string sorting the time complexity is $\Omega(\Sigma LCP(T_{[0..n]}))$.
- Another possibility is to first construct the suffix tree and then traverse it from left to right to collect the suffixes in lexicographical order. The time complexity is $\mathcal{O}(n)$ on a constant alphabet.

Specialized suffix array construction algorithms are a better option, though.



## Prefix Doubling

Our first specialized suffix array construction algorithm is a conceptually simple algorithm achieving $\mathcal{O}(n \log n)$ time.

Let $T_i^\ell$ denote the text factor $T[i..\min\{i+\ell, n+1\})$ and call it an $\ell$-factor. In other words:

- $T_i^\ell$ is the factor starting at $i$ and of length $\ell$ except when the factor is cut short by the end of the text.
- $T_i^\ell$ is the prefix of the suffix $T_i$ of length $\ell$, or $T_i$ when $|T_i| < \ell$.

The idea is to sort the sets $T^\ell_{[0..n]}$ for ever increasing values of $\ell$.

- First sort $T^1_{[0..n]}$, which is equivalent to sorting individual characters. This can be done in $\mathcal{O}(n \log n)$ time.
- Then, for $\ell = 1, 2, 4, 8, \ldots$, use the sorted set $T^\ell_{[0..n]}$ to sort the set $T^{2\ell}_{[0..n]}$ in $\mathcal{O}(n)$ time.
- After $\mathcal{O}(\log n)$ rounds, $\ell > n$ and $T^\ell_{[0..n]} = T_{[0..n]}$, so we have sorted the set of all suffixes.



We still need to specify, how to use the order for the set $T^\ell_{[0..n]}$ to sort the set $T^{2\ell}_{[0..n]}$. The key idea is assigning order preserving names (lexicographical names) for the factors in $T^\ell_{[0..n]}$. For $i \in [0..n]$, let $N_i^\ell$ be an integer in the range $[0..n]$ such that, for all $i, j \in [0..n]$:

$$N_i^\ell \leq N_j^\ell \text{ if and only if } T_i^\ell \leq T_j^\ell .$$

Then, for $\ell > n$, $N_i^\ell = SA^{-1}[i]$.

For smaller values of $\ell$, there can be many ways of satisfying the conditions and any one of them will do. A simple choice is

$$N_i^\ell = |\{j \in [0, n] \mid T_j^\ell < T_i^\ell\}| .$$

**Example 4.18:** Prefix doubling for $T =$ `banana$`.

| $N^1$ | | $N^2$ | | $N^4$ | | $N^8 = SA^{-1}$ | |
|---|---|---|---|---|---|---|---|
| 4 | b | 4 | ba | 4 | bana | 4 | banana$ |
| 1 | a | 2 | an | 3 | anan | 3 | anana$ |
| 5 | n | 5 | na | 6 | nana | 6 | nana$ |
| 1 | a | 2 | an | 2 | ana$ | 2 | ana$ |
| 5 | n | 5 | na | 5 | na$ | 5 | na$ |
| 1 | a | 1 | a$ | 1 | a$ | 1 | a$ |
| 0 | $ | 0 | $ | 0 | $ | 0 | $ |



Now, given $N^\ell$, for the purpose of sorting, we can use

- $N_i^\ell$ to represent $T_i^\ell$
- the pair $(N_i^\ell, N_{i+\ell}^\ell)$ to represent $T_i^{2\ell} = T_i^\ell T_{i+\ell}^\ell$.

Thus we can sort $T^{2\ell}_{[0..n]}$ by sorting pairs of integers, which can be done in $\mathcal{O}(n)$ time using LSD radix sort.

**Theorem 4.19:** The suffix array of a string $T[0..n]$ can be constructed in $\mathcal{O}(n \log n)$ time using prefix doubling.

- The technique of assigning order preserving names to factors whose lengths are powers of two is called the Karp–Miller–Rosenberg naming technique. It was developed for other purposes in the early seventies when suffix arrays did not exist yet.
- The best practical variant is the Larsson–Sadakane algorithm, which uses ternary quicksort instead of LSD radix sort for sorting the pairs, but still achieves $\mathcal{O}(n \log n)$ total time.

Let us return to the first phase of the prefix doubling algorithm: assigning names $N_i^1$ to individual characters. This is done by sorting the characters, which is easily within the time bound $\mathcal{O}(n \log n)$, but sometimes we can do it faster:

- On an ordered alphabet, we can use ternary quicksort for time complexity $\mathcal{O}(n \log \sigma_T)$ where $\sigma_T$ is the number of distinct symbols in $T$.
- On an integer alphabet of size $n^c$ for any constant $c$, we can use LSD radix sort with radix $n$ for time complexity $\mathcal{O}(n)$.

After this, we can replace each character $T[i]$ with $N_i^1$ to obtain a new string $T'$:

- The characters of $T'$ are integers in the range $[0..n]$.
- The character $T'[n] = 0$ is the unique, smallest symbol, i.e., `$`.
- The suffix arrays of $T$ and $T'$ are exactly the same.

Thus we can construct the suffix array using $T'$ as the text instead of $T$.

As we will see next, the suffix array of $T'$ can be constructed in linear time. Then sorting the characters of $T$ to obtain $T'$ is the asymptotically most expensive operation in the suffix array construction of $T$ for any alphabet.