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 $T = \texttt{banana\$}$ 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 4.13:

$$\begin{array}{cccccc|c|} \multicolumn{7}{c}{\mathcal{M}} \\ \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array} \quad \xrightarrow{\text{rotate}} \quad \begin{array}{|c|cccccc} \multicolumn{7}{c}{\mathcal{M}'} \\ a & \$ & b & a & n & a & n \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \\ b & a & n & a & n & a & \$ \\ \$ & b & a & n & a & n & a \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \end{array}$$

- 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:

```
------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 4.15: 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) $(\cdot, j) \leftarrow R[i]$; $LF[j] \leftarrow i$
Reconstruct text:
(5) $j \leftarrow$ position of \$ in L
(6) *for* $i \leftarrow n$ *downto* 0 *do*
(7) $T[i] \leftarrow L[j]$
(8) $j \leftarrow LF[j]$
(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`:

```
oreeereoeeieeeeaooeeeeeaereeeeeeeeeeeeereeeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeeereeeeeeeeeeeieaeeeeieeeeaeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeeaeeee
eeaeeeeeereeeaeeeeeieeeeeeeeeiieee. 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 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) $\quad c \leftarrow P[i]$
(4) $\quad b \leftarrow C[c] + \mathit{rank}_L(c, b)$
(5) $\quad e \leftarrow C[c] + \mathit{rank}_L(c, e)$
(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.

Suffix Array Construction

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

- Using standard sorting or string sorting the time complexity is $\Omega(L(T_{[0..n]}))$.
- 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.

In fact, possibly the fastest way to construct a suffix *tree* is to first construct the suffix array and the LCP array, and then the suffix tree using the algorithm we saw earlier.

Prefix Doubling

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

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

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

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

- First sort $T_{[0..n]}^1$, 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 $T_{[0..n]}^\ell$ to sort the set $T_{[0..n]}^{2\ell}$ in $\mathcal{O}(n)$ time.
- After $\mathcal{O}(\log n)$ rounds, $\ell > n$ and $T_{[0..n]}^\ell = T_{[0..n]}$, so we have sorted the set of all suffixes.

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

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

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

For smaller values of ℓ, 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^ℓ, for the purpose of sorting, we can use

- N_i^ℓ to represent T_i^ℓ
- 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 $T_{[0..n]}^{2\ell}$ 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 implementation 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 σ_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.

Recursive Suffix Array Construction

Let us now describe linear time algorithms for suffix array construction. We assume that the alphabet of the text $T[0..n)$ is $[1..n]$ and that $T[n] = 0$ (=$ in the examples).

The outline of the algorithms is:

0. Choose a subset $C \subset [0..n]$.
1. Sort the set T_C. This is done by a reduction to the suffix array construction of a string of length $|C|$, which is done *recursively*.
2. Sort the set $T_{[0..n]}$ using the order of T_C.

The set C can be chosen so that

- $|C| \leq \alpha n$ for a constant $\alpha < 1$.
- Excluding the recursive call, all steps can be done in linear time.

Then the total time complexity can be expressed as the recurrence $t(n) = \mathcal{O}(n) + t(\alpha n)$, whose solution is $t(n) = \mathcal{O}(n)$.

The set C must be chosen so that:

1. Sorting T_C can be reduced to suffix array construction on a text of length $|C|$.
2. Given sorted T_C the suffix array of T is easy to construct.

We look at two different ways of choosing C leading to two different algorithms:

- DC3 uses difference cover sampling
- SAIS uses induced sorting