Suffix Array Construction

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

- Using standard sorting or string sorting the time complexity is $\Omega(DP(T_{[0..n]}))$.
- 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 size 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 $\ell > n$, $N^\ell[i] = SA^{-1}[i]$.

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 4.12: 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 4.13: 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 assume that the text is like T' during the suffix array construction. After the construction, we can use either T or T' as the text depending on what we want to do.

Recursive Suffix Array Construction

Let us now describe a linear time algorithm 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 algorithm is:

0. Divide the suffixes into two subsets $A \subset [0..n]$ and $\bar{A} = [0..n] \setminus A$.
1. Sort the set T_A. This is done by a reduction to the suffix array construction of a string of length $|A|$, which is done *recursively*.
2. Sort the set $T_{\bar{A}}$ using the order of T_A.
3. Merge the two sorted sets T_A and $T_{\bar{A}}$.

The set A can be chosen so that

- $|A| \leq \alpha n$ 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 A must be chosen so that:

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

There are a few different options. Here we use the simplest one.

Step 0: Select A.

- For $k \in \{0, 1, 2\}$, define $C_k = \{i \in [0..n] \mid i \bmod 3 = k\}$.
- Let $A = C_1 \cup C_2$.

Example 4.14:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$

$\bar{A} = C_0 = \{0, 3, 6, 9, 12\}$, $C_1 = \{1, 4, 7, 10\}$, $C_2 = \{2, 5, 8, 11\}$ and $A = \{1, 2, 4, 5, 7, 8, 10, 11\}$.

Step 1: Sort T_A.

- For $k \in \{1, 2\}$, Construct the strings $R_k = (T_k^3, T_{k+3}^3, T_{k+6}^3, \ldots, T_{\max C_k}^3)$ whose characters are factors of length 3 in the original text, and let $R = R_1 R_2$.
- Replace each factor T_i^3 in R with a lexicographic name $N_i^3 \in [1..|R|]$. The names can be computed by sorting the factors with LSD radix sort in $\mathcal{O}(n)$ time. Let R' be the result appended with 0.
- Construct the inverse suffix array $SA_{R'}^{-1}$ of R'. This is done recursively unless all symbols in R' are unique, in which case $SA_{R'}^{-1} = R'$.
- From $SA_{R'}^{-1}$, we get lexicographic names for suffixes in T_A. For $i \in A$, let $N[i] = SA_{R'}^{-1}[j]$, where j is the position of T_i^3 in R. For $i \in \bar{A}$, let $N[i] = \perp$. Also let $N[n+1] = N[n+2] = 0$.

Example 4.15:

R	abb	ada	bba	do$	bba	dab	bad	o$	
R'	1	2	4	7	4	6	3	8	0
$SA_{R'}^{-1}$	1	2	5	7	4	6	3	8	0

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$		
$N[i]$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Step 2: Sort $T_{\bar{A}}$.

- For each $i \in \bar{A}$, we represent T_i with the pair $(T[i], N[i+1])$. Then
$$T_i \leq T_j \iff (T[i], N[i+1]) \leq (T[j], N[j+1]) \ .$$
Note that $N[i+1] \neq \perp$.

- The pairs $(T[i], N[i+1])$ are sorted by LSD radix sort in $\mathcal{O}(n)$ time.

Example 4.16:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	\$
$N[i]$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_{12} < T_6 < T_9 < T_3 < T_0$ because $(\$, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.

Step 3: Merge T_A and $T_{\bar{A}}$.

- Use comparison based merging algorithm needing $\mathcal{O}(n)$ comparisons.

- To compare $T_i \in T_A$ and $T_j \in T_{\bar{A}}$, we have two cases:
$$i \in C_1 : T_i \leq T_j \iff (T[i], N[i+1]) \leq (T[j], N[j+1])$$
$$i \in C_2 : T_i \leq T_j \iff (T[i], T[i+1], N[i+2]) \leq (T[j], T[j+1], N[j+2])$$
Note that $N[i+1] \neq \perp$ in the first case and $N[i+2] \neq \perp$ in the second case.

Example 4.17:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	\$
$N[i]$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_1 < T_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$.
$T_3 < T_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Theorem 4.18: The suffix array of a string $T[0..n)$ can be constructed in $\mathcal{O}(n)$ time plus the time needed to sort the characters of T.

- There are a few other suffix array construction algorithms and one suffix tree construction algorithm (Farach's) with the same time complexity.

- All of them have a similar recursive structure, where the problem is reduced to suffix array construction on a shorter string that represents a subset of all suffixes.

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 4.19: The BWT of $T = \mathtt{banana\$}$ is $L = \mathtt{annb\$aa}$.

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 4.20:

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

- 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 permuted. The permutation is fully determined by the last column of $\mathcal{M}$, i.e., the BWT.

- By repeating the cycle, we can reconstruct $\mathcal{M}$ from the BWT.

- To reconstruct T, we do not need to compute the whole matrix just one row.

Example 4.21:

```
------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 4.22: 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.