Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Advanced Data Structures | **Suffix Arrays** | Dr. Amin Allam

[For more details, refer to “Jewels of Stringology” by *Maxime Crochemore* and *Wojciech Rytter*]

1 Suffix arrays

The *suffix array* of a given *string* of length n (including a *sentinel* \$) is an integer array containing the *suffix IDs* of the lexicographically sorted suffixes of the *original string* (the *sentinel* \$ simplifies algorithms and is considered the smallest character). A *suffix ID* is the start index of this suffix inside the *original string*.

The purpose of the *suffix array* is the same as the *suffix tree*, but *suffix array* is less powerful (enables less operations than *suffix tree*) but more compact (needs less space than *suffix tree*).

Consider the suffixes of the string `ACGACTACGATAAC$` of length $n = 15$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	C	G	A	C	T	A	C	G	A	T	A	A	C	$

The right table shows the suffixes of the string `ACGACTACGATAAC$` sorted lexicographically:

Suffix ID	Suffix string
0	ACGACTACGATAAC$
1	CGACTACGATAAC$
2	GACTACGATAAC$
3	ACTACGATAAC$
4	CTACGATAAC$
5	TACGATAAC$
6	ACGATAAC$
7	CGATAAC$
8	GATAAC$
9	ATAAC$
10	TAAC$
11	AAC$
12	AC$
13	C$
14	$

Suffix ID	Suffix string
14	$
11	AAC$
12	AC$
0	ACGACTACGATAAC$
6	ACGATAAC$
3	ACTACGATAAC$
9	ATAAC$
13	C$
1	CGACTACGATAAC$
7	CGATAAC$
4	CTACGATAAC$
2	GACTACGATAAC$
8	GATAAC$
10	TAAC$
5	TACGATAAC$

Therefore, the *suffix array* of the string `ACGACTACGATAAC$` is:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Suffix array	14	11	12	0	6	3	9	13	1	7	4	2	8	10	5

Using the *suffix array* and the *original string*, we can search for any *substring query* of length m inside the *original string* in $O(m \times (\log n + occ))$ time using *binary search*, where occ is the number of occurrences of the *substring query* inside the *original string*.

The above complexity is achieved because each of the $O(\log n)$ *binary search* iterations includes an $O(m)$ string comparison of the *substring query* to an *original string suffix*. After the *binary search* is done, we perform string comparison of the *substring query* to all *suffixes* starting from the result location of the *binary search* until we encounter a *suffix* which is not prefixed by the *substring query*. The number of such *suffixes* is occ.

The third column of *suffix* strings in the following figure is not part of the *suffix array* and is shown for illustration only since it can be deduced easily from the *suffix array* and the *original string*.

Index	Suffix array	Corresponding suffix
0	14	`$`
1	11	`AAC$`
2	12	`AC$`
3	0	`ACGACTACGATAAC$`
4	6	`ACGATAAC$`
5	3	`ACTACGATAAC$`
6	9	`ATAAC$`
7	13	`C$`
8	1	`CGACTACGATAAC$`
9	7	`CGATAAC$`
10	4	`CTACGATAAC$`
11	2	`GACTACGATAAC$`
12	8	`GATAAC$`
13	10	`TAAC$`
14	5	`TACGATAAC$`

Here we trace the *binary search* for the substring `CGA` using the above *suffix array* only. We start with an unexplored interval $[0, 15]$ representing $[first\_index, last\_index + 1]$.

Middle index is $\lfloor (0 + 15)/2 \rfloor = 7$. `CGA` > `C$`. Interval shrinks to $[8, 15]$.
Middle index is $\lfloor (8 + 15)/2 \rfloor = 11$. `CGA` < `GACTACGATAAC$`. Interval shrinks to $[8, 11]$.
Middle index is $\lfloor (8 + 11)/2 \rfloor = 9$. `CGA` < `CGATAAC$`. Interval shrinks to $[8, 9]$.
Middle index is $\lfloor (8 + 9)/2 \rfloor = 8$. `CGA` < `CGACTACGATAAC$`. Interval shrinks to $[8, 8]$.

Then, we test if `CGA` is prefix of suffixes at indexes ≥ 8 in *suffix array*:
`CGA` is prefix of `CGACTACGATAAC$` at index 8. Report occurrence at index 1 in *original string*.
`CGA` is prefix of `CGATAAC$` at index 9. Report occurrence at index 7 in *original string*.
`CGA` is not prefix of `CTACGATAAC$` at index 10. Stop.

The complexity can be improved to $O(m \log n + occ)$ by performing another *binary search* instead of comparing all *suffixes* starting from result index of the first *binary search*. The second *binary search* is similar to the first one, except that if *query substring* is *prefix* of compared *suffix*, it is considered greater than this *suffix*. In the above example, the second *binary search* will result the interval $[10, 10]$ which is $1+$ index of the last occurrence of *query substring* in *suffix array*.

2 Suffix array construction

A *suffix array* can be constructed naively in $O(n^2 \log n)$ using an $O(n \log n)$ sorting algorithm such as *merge sort*. The additional n factor in complexity arises because the complexity of each *suffix* comparison performed by the algorithm is $O(n)$, not $O(1)$.

We can utilize the strong relation between *suffixes* of the same *original string* to improve the *suffix array* construction time to $O(n \log n)$ using the following *prefix doubling* algorithm.

Consider constructing *suffix array* of string `ACGACTACGATAAC$` using *prefix doubling*. The initial step is to sort all *suffixes* by their first character only, simply by assigning to each *suffix* the order of its first character in the alphabet. Remember that `$` is the smallest character.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0

From the above table, we recognize that the smallest *suffix* `$` gets the smallest integer 0. The immediately larger *suffixes* are those starting with `A`. They all got the next smallest integer 1, because they are equal if we look at their first character only which is `A`. The immediately larger *suffixes* are those starting with `B`. They all got the next smallest integer 2, because they are equal if we look at their first character only which is `B`.

The general rule is that, in iteration i, all *suffixes* are sorted according to their first 2^i characters only. That is, we assume that the length of each suffix is only 2^i. All *suffixes* starting with the same prefix of size 2^i are considered equal and assigned the same integer. Thus, the second iteration $i = 1$ assigns the same integer to all *suffixes* starting with the same $2^1 = 2$ characters:

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

From the above table, we recognize that the smallest *suffix* `$` gets the smallest integer 0. The immediately larger *suffixes* are those starting with `AA`. It is exactly one *suffix* and it got the next smallest integer 1. The immediately larger *suffixes* are those starting with `AC`. They all got the next smallest integer 2, because they are equal if we look at their first two characters only which are `AC`.

Here we explain how to reduce time complexity. The next iteration $i = 2$ of the algorithm is going to sort *suffixes* according to their first $2^i = 2^2 = 4$ characters. Instead of comparing two *suffixes* by performing a string comparison of their first 4 characters, we will perform a more efficient *suffix* comparison using the results of the previous iteration $i = 1$.

To compare two *suffixes* at iteration i, look at their assigned integers at iteration $i-1$. If the integers are not equal, their relative order remains the same. If the integers are equal, look at relative order of the two *suffixes* shifted by 2^i positions from the locations of the needed *suffixes*.

For example, to compare the first 4 characters of the two *suffixes* at indexes 4 (`CTAC`) and 7 (`CGAT`), look at their relative order according to their first 2 characters, appearing in last row in the above table to be 6 and 5, indicating that *suffix* 4 is larger than *suffix* 7. The relation remains the same.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

Another example for the other case, to compare the first 4 characters of the two *suffixes* at indexes 2 (`GACT`) and 8 (`GATA`). Their orders according to their first 2 characters, appearing in last row in the above table are 7 and 7, indicating that *suffix* 2 is equal to *suffix* 8 with respect to the first 2 characters (`GA`).

Since they are equal, we consider the two *suffixes* shifted by 2 from the original *suffixes* indexes, which are *suffixes* at indexes $2 + 2 = 4$ (`CT`) and $8 + 2 = 10$ (`TA`). Their orders according to their first 2 characters, appearing in last row in the above table are 6 and 8, indicating that *suffix* 4 is smaller than *suffix* 10 with respect to their first 2 characters (`GA`), which implies the same relation between the original two *suffixes* 2 (`GACT`) and 8 (`GATA`) with respect to their first 4 characters.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7	10	4	9	13	3	8	11	5	12	1	2	6	0

Here we explain how to obtain all *suffix* orders from of iteration 2 from iteration 1. There is exactly one *suffix* with order 0 which is *suffix* 14, its order remains the same. Also, only *suffix* 11 has order 1 and remains the same.

There are 4 *suffixes* with order 2 which are 0, 3, 6, 12. We look at shifted-by-2 *suffixes* 2, 5, 8, 14 their orders are 7, 8, 7, 0 to conclude that the smallest *suffix* is 12 so we assign to it order of 2 (because last assigned order was 1). Then, next smallest *suffixes* are 0 and 6 with the same order of 3, meaning that they are still equal with respect to their first 4 characters, then *suffix* 3 takes order of 4 (because last assigned order was 3).

Only *suffix* 9 has order 3 in iteration 1. It is assigned order 5 in iteration 2 (because last assigned order was 4). Only *suffix* 13 has order 4. It is assigned order 6. There are 2 *suffixes* with order 5 which are 1, 7. We look at shifted-by-2 *suffixes* 3, 9 their orders are 2, 3 to conclude that the smaller *suffix* is 1 so we assign to it order of 7, then *suffix* 7 takes order of 8. Only *suffix* 4 has order 6. It is assigned order 9.

There are 2 *suffixes* with order 7 which are 2, 8. We look at shifted-by-2 *suffixes* 4, 10 their orders are 6, 8 to conclude that the smaller *suffix* is 2 so we assign to it order of 10, then *suffix* 8 takes order of 11. There are 2 *suffixes* with order 8 which are 5, 10. We look at shifted-by-2 *suffixes* 7, 12 their orders are 5, 1 to conclude that the smaller *suffix* is 10 so we assign to it order of 12, then *suffix* 5 takes order of 13.

Note that actually there should not be any two equal *suffixes*, so the algorithm terminates only if there are no two *suffixes* with the same order.

To move to the next iteration 3, the only two suffixes with same order are 0, 6. We look at shifted-by-4 *suffixes* 4, 10 their orders in iteration 2 are 9, 12 to conclude that the smaller *suffix* is 0 so we assign to it a smaller order than suffix 6 as follows:

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7	10	4	9	13	3	8	11	5	12	1	2	6	0
3	$2^3 = 8$	3	8	11	5	10	14	4	9	12	6	13	1	2	7	0

The algorithm terminates because all suffixes have different orders as they should. Since we terminated at iteration 3 we conclude that no two *suffixes* share the same prefix of $2^3 = 8$ characters.

The resulting array is not the *suffix array*, but it is the *inverse* of the *suffix array*. The resulting array tells the order given a *suffix ID* (example: suffix 12 has the order 2). The *suffix array* tells the *suffix ID* given an order (example: the suffix of order 2 is 12). The *suffix array* can be easily obtained from its *inverse* by $O(n)$ sequential scan:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Suffix array	14	11	12	0	6	3	9	13	1	7	4	2	8	10	5

In the worst case, no two *suffixes* share the same prefix of n characters because all *suffixes* are different. Therefore, the number of iterations is $O(\log n)$ can be concluded from the second column because maximum sorted prefix length is n and it is multiplied by 2 at each iteration.

At each iteration suffixes are sorted using $O(n \log n)$ sorting algorithm, then the required array in the figures above is obtained in $O(n)$ sequential scan over the sorted array. Each suffix comparison needs only $O(1)$ operations since we compare 2 orders, and when equal we compare 2 shifted orders. It can be viewed as comparing pairs of integers. Therefore, the total time complexity is $O(n \log^2 n)$. Since the sorting procedure compares two pairs of integers whose range is n, we can use two-pass $O(n)$ radix sorting at each iteration to reduce the total complexity to $O(n \log n)$. There is also an $O(n)$ *suffix array* construction algorithm called *induced sorting*.