7

Suffix Array Construction

In this lecture we will be interested in solving the following problem, also known as *full-text searching*.

> **The substring-search problem.** *Given a text string $T[0, n-1]$, drawn from an alphabet of size σ, retrieve (or just count) all text positions where a query pattern $P[0, p-1]$ occurs as a substring of T.*

It is evident that this problem can be solved by brute-forcedly comparing P against every substring of T, thus taking $O(np)$ time in the worst case. But it is equivalently evident that this *scan*-based approach is unacceptably slow when applied to massive text collections, which is the scenario investigated in these lectures, such as genomic databases or search engines. This suggests the usage of a so called *indexing* data structure which is built over T before that searches start. A setup cost is required for this construction, but this cost is amortized over the subsequent pattern searches, thus resulting convenient in a quasi-static environment in which T is changed very rarely.

The *suffix array* (shortly *SA*) is a fundamental data structure proposed in 1989 by Manber and Myers [6] which solves the substring search problem fast and space succinctly. Typically this data structure is looked as the space-efficient substitute of the *suffix tree* data structure. As it will appear clear from our discussion, the applications of the suffix array go far beyond the context of full-text search and extend to many other text-mining applications.

7.1 Notation and terminology

We assume that text T ends with a special character $T[n-1] = \$$, which is smaller than any other alphabet character. This ensures that text suffixes are prefix-free and thus no one is a prefix of another. We use $suff_i$ to denote the i-th suffix of T, namely the substring $T[i, n]$. The following observation is crucial:

> If $P = T[i, i+p-1]$, then the pattern occurs at text position i and thus we can equivalently state that P is a prefix of the i-th text suffix, namely it is a prefix of the string $suff_i$.

As an example, if $P =$"siss" and $T =$"mississippi\$", then P occurs at text position 3 and indeed it prefixes the suffix $suff_3 = T[3, 12] =$"sissippi\$".

As a consequence, if all text suffixes form the dictionary $SUF(T)$, then *searching for P as a substring of T boils down to searching for P as a prefix of some string in $SUF(T)$*. In addition, since there is a bijective correspondence Among the text suffixes prefixed by P and the pattern occurrences in T, then

1. the suffixes prefixed by P occur contiguously into the sorted $SUF(T)$, and
2. P has a lexicographic position in $SUF(T)$ that immediately precedes that block of matching suffixes.

An attentive reader may have noticed that these are the properties deployed in Chapter **??** to efficiently support prefix searches. And indeed the solutions known in the literature for efficiently solving the substring-search problem hinge either on array-based data structures (i.e. the Suffix Array) or on trie-based data structures (i.e. the Suffix Tree). So the use of these data structures in pattern searching is pretty immediate, and thus we detail it below only for completeness and for stating some cute improvements. What is challenging is the efficient construction of these data structures which is definitely not obvious and indeed took almost 20 years to find an elegant and optimal solution.

Text suffixes	**Indexes**
mississippi$	1
ississippi$	2
ssissippi$	3
sissippi$	4
issippi$	5
ssippi$	6
sippi$	7
ippi$	8
ppi$	9
pi$	10
i$	11
$	12

Sorted Suffixes	*SA*	***Lcp***
$	12	0
i$	11	1
ippi$	8	1
issippi$	5	4
ississippi$	2	0
mississippi$	1	0
pi$	10	1
ppi$	9	0
sippi$	7	2
sissippi$	4	1
ssippi$	6	3
ssissippi$	3	-

FIGURE 7.1: *SA* and lcp array for the string T =“mississippi$”.

7.2 The suffix array

The suffix array is essentially the lexicographically-sorted permutation of all text suffixes. We use the notation $SA(T)$ to denote the suffix array built over the text T (or just *SA* if it is clear from the context). Because of the lexicographic ordering, $SA[i]$ is the i-th smallest suffix of the text T. So we have that

$$suff_{SA[0]} < suff_{SA[1]} < \cdots < suff_{SA[n-1]}$$

where $<$ is the lexicographical order. For space reasons, each suffix is represented by its starting position in T (i.e. an integer). So *SA* consists of n integers in the range $[0, n-1]$ and hence it occupies $O(n \log n)$ bits. With respect to the suffix tree data structure (which stores explicitly the

whole tree structure), the suffix array stores less information which, in practice, can be quantified in a saving of a factor $10 - 30$.

Another useful concept is the *longest common prefix* between two consecutive suffixes $suff_{SA[i]}$ and $suff_{SA[i+1]}$, denoted as $lcp[i]$. lcp is an array of integers too, with $n-1$ entries and values smaller than n. There is an optimal linear time algorithm to build the lcp-array [5], which will be detailed in Section 7.4.1. The interest in Lcp rests in its usefulness to design efficient/optimal algorithms to solve various pattern matching problems.

7.3 The substring-search problem

We observed that this problem can be reduced to a prefix search over the string dictionary $SUF(T)$. In Chapter 6 we commented widely about algorithmic solutions to the prefix-search problem. Here we observe that the suffix array $SA(T)$ can be used to implement the approach based on binary search, so that if no suffix is found to be prefixed by P, the pattern does not occur in T. Figure 7.1 shows the pseudo-code.

Algorithm 7.1 SEARCH($P, SA(T)$)

```
1:  L = 0, R = n − 1;
2:  while (L ≠ R) do
3:      M = (L + R)/2;
4:      if (strcmp(P, suff_M) > 0) then
5:          L = M;
6:      else
7:          R = M;
8:      end if
9:  end while
10: return (strcmp(P, suff_L) == 0);
```

A binary search in $SA(T)$ requires $O(\log n)$ string comparisons, each taking $O(p)$ time in the worst case. Thus, the time complexity is $O(p \log n)$.

Figure 7.2 shows a running example, which highlights an interesting property: the comparison between P and $suff_M$ does not need to start from their initial character. In fact one could exploit the lexicographic sorting of the suffixes and skip the characters comparisons that have already been carried out in previous iterations. This can be done with the help of three arrays:

- the $lcp[0, n-1]$ array.
- the arrays $Llcp[0, n-1]$ and $Rlcp[0, n-1]$ which are defined for every triple (L, M, R) that may arise in the inner loop of a binary search, as follows:

$$Llcp[M] = lcp(suff_{SA[L_M]}, suff_{SA[M]}) \qquad Rlcp[M] = lcp(suff_{SA[M]}, suff_{SA[R_M]})$$

We notice that each triple (L, M, R) is uniquely identified by its midpoint M because the execution of a binary search over an array of size n (such as SA) can be described as traversing a downward path in a binary tree of n leaves. Thus each triple corresponds to a node of this tree, and hence we have $O(n)$ triples. So these three arrays cost $O(n)$ space.

As far as their building time is concerned, we observe that the lcp array can be built in $O(n)$ optimal time as described in section 7.5. The arrays $Llcp$ and $Rlcp$ can be also built in linear time

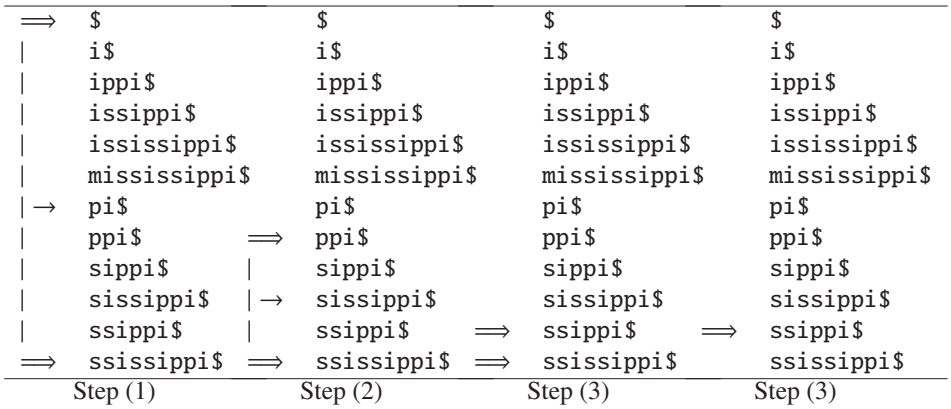


FIGURE 7.2: Binary search steps for the pattern P ="ssi" in "mississippi$".

by exploiting the following observation: the $lcp(i, j)$ between two suffixes in $suff_{SA[i]}$ and $suff_{SA[j]}$ can be computed as the minimum of a range of *lcp*-values, namely $lcp(i, j) = \min_{k=i,\ldots,j-1} lcp[k]$. By associativity of the min we can split the computation as $lcp(i, j) = \min\{lcp(i, k), lcp(k, j)\}$ where k is any index in the range $[i, j]$. This implies that the arrays *Llcp* and *Rlcp* can be computed by a bottom-up traversal in binary-tree order of the *lcp* array, taking in each step the minimum *lcp* values on the left and right children. This can be performed in $O(n)$ time. Another solution could be to compute $Llcp[M]$ and $Rlcp[M]$ on-the-fly via a Range-Minimum Data structure built over the array *lcp* (see Chapter **??**). This would occupy $O(n)$ space and take constant time to report the minimum value in a given range. Whichever is the approach chosen to compute these three arrays, the overall construction time is the optimal $O(n)$.

We are left with showing how the binary search can be speeded up by using these arrays. Consider a binary search iteration on the subarray $SA[L, R]$, and let M be the midpoint of the binary search. A lexicographic comparison between P and $suff_{SA[M]}$ has to be made in order to choose the next search-range between $SA[L, M-1]$ and $SA[M+1, R]$. Assume that we know inductively the values $l = |lcp(P, suff_{SA[L]})|$ and $r = |lcp(P, suff_{SA[R]})|$. At the beginning they can be computed in $O(p)$ time.

Since P lies between $suff_{SA[L]}$ and $suff_{SA[R]}$, it shares with these suffixes $k = |lcp(L, R)|$ characters. Since we inductively know the lcp-values between P and the suffixes at the extremes of the range, namely l and r, we can conclude that $l \geq k$ and $r \geq k$, so for sure $m = |lcp(P, suff_{SA[M]})| \geq k$. Starting from the k-th character of P is not enough to avoid its rescanning, because r and l could be larger and thus have involved subsequent characters in previous comparisons. Nevertheless, we can be more precise. In fact, if $l = r$ then all suffixes in the range $[L, R]$ share l characters (hence $suff_{SA[M]}$ too) and these are equal to the first l characters of P. So the comparison between P and $suff_{SA[M]}$ can start from their $(l+1)$-th character, which means that we are advancing in the scanning of P. Otherwise (i.e. $l \neq r$), a more complicated test has to be done. Consider the case where $l > r$ (the other being symmetric):

- If $l < Llcp[M]$, then P is greater that $suff_{SA[M]}$ and it is $m = l$. In fact $P > suff_{SA[L]}$ and the mismatch lies at position $l + 1$. But this text suffix shares more than l characters with $suff_{SA[M]}$. So the mismatch between P and $suff_{SA[M]}$ is the same as it was with $suff_{SA[L]}$, hence their comparison gives the same answer— i.e. $P > suff_{SA[M]}$— and the search continues in the subrange $SA[M, R]$. we remark that this case did not induce any character comparison.

- If $l > Llcp[M]$, this case is similar as the one commented above. We can conclude that P is smaller than $suff_{SA[M]}$ and it is $m = Llcp[M]$. So the search continues in the subrange $SA[L, M]$, without additional character comparisons.
- If $l = Llcp[M]$, then P shares l characters also with $suff_{SA[M]}$. So the comparison between P and $suff_{SA[M]}$ can start from their $(l + 1)$-th character. Eventually we have determined m and found their lexicographic order.

It is clear that every binary-search step either advances the comparison of P's characters, or it does not compare any character but it halves the range $[L, R]$. The first case can occur at most p times, the second $O(\log n)$ time.

LEMMA 7.1 Given the three arrays *lcp*, *Llcp* and *Rlcp* built over a text $T[0, n-1]$, counting the occurrences of a pattern $P[0, p-1]$ in T takes $O(p + \log n)$ time. Retrieving the positions of these *occ* occurrences takes additional $O(occ)$ time.

Proof We remind that searching for all strings having a pattern P as a prefix requires two lexicographic searches: one for P and the other for $P\#$, where # is a special character larger than any other alphabet character. So $O(p + \log n)$ character comparisons are enough to delimit the range $SA[i, j]$ of suffixes having P as a prefix. It is then trivial to count the occurrences in constant time, as $occ = j - i + 1$, or print all of them in $O(occ)$ time.

7.4 Suffix Array Construction Algorithms

Given that the suffix array is a sorted sequence of items, the most intuitive way to construct *SA* is to use an efficient comparison-based sorting algorithm. `Algorithm ??` implements this idea in C-style using the procedure QSORT as sorter and the subroutine Suffix_cmp for comparing suffixes.

Algorithm 7.2 COMPARISON_BASED_CONSTRUCTION(char *T, int n, char **SA)

```
1: for (i = 0; i < n; i ++) do
2:       SA[i] = T + i;
3: end for
4: qsort(SA, n, sizeof(char *), Suffix_cmp);
```

For completeness, the latter procedure could be implemented as

```
Suffix_cmp(char **p, char **q){ return strcmp(*p, *q) };
```

A major drawback of this simple approach is that it is not I/O-efficient for two main reasons: the optimal $O(n \log n)$ comparisons involve variable-length strings of length up to $O(n)$; locality in *SA* does not translate into locality in suffix comparisons because of the fact that string pointers are permuted rather than their pointed strings. Both these issues elicit I/Os, and turn this simple algorithm into a slow one.

THEOREM 7.1 *In the worst case the use of a comparison-based sorter to construct the suffix array of a given a string $T[1, n]$ requires $O((\frac{n}{B})n \log n)$ I/Os, and $O(n \log n)$ bits of working space.*

In Section 7.4.1 we describe a Divide-and-Conquer algorithm— the *Skew* algorithm by Kärkkäinen and Sanders [4]— which is elegant, easy to code, and flexible enough to achieve the optimal I/O-bounds in various models of computations. In Section **??** we describe also the algorithm proposed by Gonnet-BaezaYates and Sniders in [3], which is also simple and offers the positive feature of processing the input data in passes thus being suitable for slow disks.

7.4.1 The Skew Algorithm

In 2003, Kärkkäinen and Sanders [4] provided an optimal algorithm that constructs a suffix array for a given string without passing through the construction of its suffix tree. This algorithm is named *Skew* in the literature, and works with a time complexity of $O(n)$, an I/O-complexity equal to the one needed to sort n items, and with a space occupancy of $O(n \log n)$ bits. Since suffix-array construction is reduced to sorting atomic items, this algorithm can be used over every model of computation in which an efficient sorting primitive is available: disk, distributed, parallel.

Let $T[0, n) = t_0 t_1 \dots t_{n-1}$ be the input string[1] of n characters over the alphabet $\Sigma = \{1, 2, \dots, \sigma\}$ and let $t_j = \$$ for $j \geq n$ be a special character smaller than any other alphabet character. The algorithm hinges on a divide&conquer approach that executes a $\frac{2}{3} : \frac{1}{3}$ split, crucial to make the final merge-step easy and implementable via arrays only. Previous approaches used the more natural $\frac{1}{2} : \frac{1}{2}$ split (such as [1]) but were forced to use a more sophisticated merge-step which needed the use of the suffix-tree structure. The Skew algorithm consists of tree basic steps:

Step 1. Construct the suffix array of the suffixes starting at positions $P_{1,2} = \{i : i \bmod 3 \neq 0\}$:

- This is done by building the string $T^{1,2}$ of length $(2/3)n$ which compactly encodes all suffixes of T starting at positions $P_{1,2}$.
- And then running the suffix-array construction algorithm recursively over it. The result is the suffix array $SA^{1,2}$, which actually corresponds to the lexicographically sorted sequence of text suffixes starting at positions $P_{1,2}$.

Step 2 Construct the suffix array of the remaining text suffixes starting at positions $P_0 = \{i : i \bmod 3 = 0\}$:

- This is done by representing every text suffix $T[i, n]$ with a pair $\langle T[i], \mathtt{pos}(i+1)\rangle$, where we have that $i + 1 \in P_{1,2}$ and $\mathtt{pos}(i + 1)$ is the position of the $(i + 1)$-th text suffix in $SA^{1,2}$.
- And then running radix-sort over this set of $O(n)$ pairs.

Step 3. Merge the two suffix arrays into one:

- This is done by deploying the $\frac{2}{3} : \frac{1}{3}$ which ensures a constant-time lexicographic comparison between any pair of suffixes.

The execution of the algorithm is illustrated using the string $T[0, 10] =$"mississippi", where the final suffix array will be $SA = (10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2)$[2]. In this example we have: $P_{1,2} = \{1, 2, 4, 5, 7, 8, 10, 11\}$ and $P_0 = \{0, 3, 6, 9\}$.

Step 1. The first step is the most time consuming and consists of lexicographically sorting the suffixes which start at the text positions $P_{1,2}$. The resulting array is denoted by $SA^{1,2}$ and represents

[1] For the sake of presentation we assume that the input string starts from the index 0.

[2] This suffix array is analogous to suffix array shown in figure 7.1. The difference is just in the indexes that start from 0.

a *sampled* version of the final suffix array SA because it corresponds to the suffixes starting at the positions $P_{1,2}$.

To efficiently obtain $SA^{1,2}$, we reduce the problem to the construction of the suffix array for a string $T^{1,2}$ of length $\frac{2}{3}$. This way the construction can occur recursively without impairing in the final time complexity.

The key difficulty is how to define $T^{1,2}$ so that its suffix array corresponds to the sorted sequence of text suffixes starting at the positions in $P_{1,2}$. The elegant solution consists of constructing the strings $R_k = [t_k, t_{k+1}, t_{k+2}][t_{k+3}, t_{k+4}, t_{k+5}]\ldots]$, for $k = 1, 2$. These strings are composed by triples of symbols $[t_i, t_{i+1}, t_{i+2}]$ which therefore take $O(\log n)$ bits.

$$R_1 = \left\{\underset{1}{\texttt{[i s s]}}\,\underset{4}{\texttt{[i s s]}}\,\underset{7}{\texttt{[i p p]}}\,\underset{10}{\texttt{[i \$ \$]}}\right\} \qquad R_2 = \left\{\underset{\mathbf{2}}{\texttt{[s s i]}}\,\underset{\mathbf{5}}{\texttt{[s s i]}}\,\underset{\mathbf{8}}{\texttt{[p p i]}}\right\}$$

We denote with $R = R_1 \bullet R_2$ the string formed by concatenating the triples of R_1 with the triples of R_2.

$$R = \left\{\underset{1}{\texttt{[i s s]}}\,\underset{4}{\texttt{[i s s]}}\,\underset{7}{\texttt{[i p p]}}\,\underset{10}{\texttt{[i \$ \$]}}\underset{2}{\texttt{[s s i]}}\,\underset{5}{\texttt{[s s i]}}\,\underset{8}{\texttt{[p p i]}}\right\}$$

The key property is the following:

Property 1 *Every text suffix $T[i, n)$ starting at a position in $P_{1,2}$, can be put in correspondence with a suffix of R consisting of a sequence of triples. Specifically, if $i \bmod 3 = 2$ then the text suffix coincides exactly with a suffix of R; if $i \bmod 3 = 1$, then the text suffix prefixes a suffix of R which nevertheless terminates with special symbol \$.*

This property is crucial to state that the lexicographic comparison between text suffixes starting in $P_{1,2}$ can be derived by comparing lexicographically the suffixes of R, provided that they are aligned to its triples.

In order to manage those triples efficiently, we encode them via integers in a way that their lexicographic comparison can be obtained by comparing those integers. In the literature this is called *lexicographic naming* and can be easily obtained by *radix sorting* the triples in R and associating to each distinct triple its *rank* in the lexicographic order. Since we have $O(n)$ triples, each consisting of symbols in a range $[0, n]$, their radix sort takes $O(n)$ time.

In our example, the sorted triples are labeled with the following ranks:

[i $ $]	[i p p]	[i s s]	[i s s]	[p p i]	[s s i]	[s s i]	sorted triples
1	2	3	3	4	5	5	sorted ranks

$R =$ [i s s]	[i s s]	[i p p]	[i $ $]	[s s i]	[s s i]	[p p i]	triples
3	3	2	1	5	5	4	$T^{1,2}$ (string of ranks)

In a general recursive step, T is a sequence of integers in a range $[0, n)$. So we can assume that this is true also at the beginning, it is enough to re-code the text symbols via a sorting step.

It is evident from the discussion above that, since the ranks are assigned in the same order as the lexicographic order of their triples, the lexicographic comparison between suffixes of R (aligned to the triples) equals the lexicographic comparison between suffixes of $T^{1,2}$. Moreover $T^{1,2}$ consists of $\frac{2}{3}$ symbols (integers smaller than n). So $SA^{1,2}$ can be obtained by building recursively the suffix array $T^{1,2}$.

There are two notes to be added at this point. The first one is that, if all symbols in $T^{1,2}$ are different, then recursion is useless because suffixes can be sorted by looking just at their first characters.

The second observation is for the programmers that should be careful in turning the suffix-positions in $T^{1,2}$ into the suffix positions in T, according to the layout of the triples of R.

Since $T^{1,2} = (3, 3, 2, 1, 5, 5, 4)$ and not all ranks are distinct, the algorithm is applied recursively returning the array $(3, 2, 1, 0, 6, 5, 4)$, whose positions refer to the positions in $T^{1,2}$. Mapping them to positions in T we get $SA^{1,2} = (10, 7, 4, 1, 8, 5, 2)$.

Step 2. Once the suffix array $SA^{1,2}$ has been built, it is possible to sort lexicographically the remaining suffixes of T, starting at the text position $i \bmod 3 = 0$, in a simple way. We decompose a suffix as composed by its first character $T[i]$ and its remaining suffix; and then encode this as a pair of integers $\langle T[i], \mathtt{pos}(i+1)\rangle$, where $\mathtt{pos}(j)$ denotes the rank in $SA^{1,2}$ of the suffix j. Clearly, if $i \in P_1$ then $i + 1 \in P_{1,2}$ so that $\mathtt{pos}(i + 1)$ is well defined.

Given this observation, two text suffixes starting at positions in P_1 can then be compared in constant time by comparing their corresponding pairs. Therefore, SA^1 can be efficiently computed by radix-sorting the $O(n)$ pairs encoding its suffixes in $O(n)$ time.

In our example, this boils down to the radix sort of the pairs:

Pairs:	$\langle \mathtt{m}, 4\rangle$	$\langle \mathtt{s}, 3\rangle$	$\langle \mathtt{s}, 2\rangle$	$\langle \mathtt{p}, 1\rangle$	suffixes mod 0
	0	3	6	9	starting positions in T
Sorted pairs:	$\langle \mathtt{m}, 4\rangle$	$< \langle \mathtt{p}, 1\rangle$	$< \langle \mathtt{s}, 2\rangle$	$< \langle \mathtt{s}, 3\rangle$	sorted suffixes
	0	9	6	3	SA^0

Step 3. The final step merges the two sorted arrays SA^0 and $SA^{1,2}$ in linear time by resorting an interesting observation which motivates the split $\frac{2}{3} : \frac{1}{3}$. Let us given two suffixes $T[i, n) \in SA^0$ and $T[j, n) \in SA^{1,2}$, which we wish to lexicographically compare. They belong to two different suffix arrays so we have no *lexicographic relation* known for them, and we cannot compare them character-by-character because this would incur in a much higher cost. Nevertheless, we can deploy a decomposition idea similar to the one exploited in Step 2, which consists of looking at a suffix as composed by *one or two characters* plus the lexicographic rank of its remaining suffix. This decomposition becomes effective if the remaining suffixes of the compared ones lie in the same suffix array, so that their rank is enough to get their order in constant time. Elegantly enough this is possible with the split $\frac{2}{3} : \frac{1}{3}$, but it could not be possible with the split $\frac{1}{2} : \frac{1}{2}$. This observation is implemented as follows:

1. if $j \bmod 3 = 1$ then we compare $T[j, n) = T[j]T[j+1, n)$ against $T[i, n) = T[i]T[i+1, n)$. Both the suffixes $T[j+1, n)$ and $T[i+1, n)$ occur in $SA^{1,2}$ so that we can derive the above lexicographic comparison by comparing the pairs $\langle T[i], \mathtt{pos}(i+1)\rangle$ and $\langle T[j], \mathtt{pos}(j+1)\rangle$. This comparison takes $O(1)$ time, provided that the array $\mathtt{pos}$ is available.[3]
2. if $j \bmod 3 = 2$ then we compare $T[j, n) = T[j]T[j + 1]T[j + 2, n)$ against $T[i, n) = T[i]T[i + 1]T[i + 2, n)$. Both the suffixes $T[j + 2, n)$ and $T[i + 2, n)$ occur in $SA^{1,2}$ so that we can derive the above lexicographic comparison by comparing the triples $\langle T[i], T[i + 1], \mathtt{pos}(i+2)\rangle$ and $\langle T[j], T[j+1], \mathtt{pos}(j+2)\rangle$. This comparison takes $O(1)$ time, provided that the array $\mathtt{pos}$ is available.

In our running example we have that $T[7, 12) < T[9, 12)$, and in fact $\langle i, 5\rangle < \langle p, 1\rangle$. Also we have that $T[6, 12) < T[5, 12)$ and in fact $\langle s, i, 5\rangle < \langle s, s, 2\rangle$. In the following figure we depict all possible pairs of triples which may be involved in a comparison, where $(\star\star)$ and $(\star\star\star)$ denote the pairs for rule 1 and 2 above. Conversely $(\star)$ denotes the starting position in T of the suffix.

[3] Of course, the array $\mathtt{pos}$ can be derived from $SA^{1,2}$ in linear time, since it is its inverse.

SA^0				$SA^{1,2}$							
0	9	6	3	10	7	4	1	8	5	2	($\star$)
$\langle$m, 4$\rangle$	$\langle$p, 1$\rangle$	$\langle$s, 2$\rangle$	$\langle$s, 3$\rangle$	$\langle$i, 0$\rangle$	$\langle$i, 5$\rangle$	$\langle$i, 6$\rangle$	$\langle$i, 7$\rangle$				($\star\star$)
$\langle$m, i, 7$\rangle$	$\langle$p, i, 0$\rangle$	$\langle$s, i, 5$\rangle$	$\langle$s, i, 6$\rangle$					$\langle$p, p, 1$\rangle$	$\langle$s, s, 2$\rangle$	$\langle$s, s, 3$\rangle$	($\star\star\star$)

At the end of the merge step we obtain the final suffix array: $SA = (10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2)$. The execution time of the algorithm can be modeled by the recurrence $T(n) = T(\frac{2n}{3}) + O(n)$, because Steps 2 and 3 cost $O(n)$ and the recursive call is executed over a string $T^{1,2}$ whose length is $(2/3)n$. It can be easily verified that the recurrence has solution $T(n) = O(n)$, which is clearly optimal.

From the discussion above it is clear that every step can be implemented via a *sorting* or a scanning of a set of n atomic items, which are possibly triples of integers. Therefore the algorithm can be seen as a *algorithmic reduction* of the suffix-array construction problem to the n-items sorting problem. This problem has been solved optimally in several models of computation. For what concerns the disk model, the skew algorithm can be implemented in $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$ I/Os. we have therefore proved the following.

THEOREM 7.2 *The skew algorithm builds the suffix array of a string $T[1, n]$ in $O(Sort(n))$ I/Os and $O(n/B)$ disk pages. If the alphabet Σ has size polynomial in n, the CPU time is $O(n)$.*

7.4.2 The GBS Algorithm

Before the Skew algorithm, the best know disk-based algorithm was the one proposed by Gonnet and Baeza-Yates and Sniders in 1992 [3] (shortly GBS). It is also a divide&conquer algorithm in which the divide step is unbalanced, thus inducing a *cubic* time complexity because of a quadratic number of suffix comparisons. Nevertheless the algorithm is very fast in practice because it processes the data into passes thus deploying the high throughput of modern disks.

Let $\ell < 1$ be a positive constant, properly fixed to build the suffix array of a text piece of $m = \ell M$ characters in internal memory. For the sake of presentation, we assume that the text T is logically divided into pieces of m characters each, numbered rightward: namely $T = T_1 T_2 \cdots T_{n/m}$ where $T_h = T[hm + 1, (h + 1)m]$ for $h = 0, 1, \ldots$. The BGS algorithm computes *incrementally* the suffix array of the text string $T[1, n]$ in $\Theta(n/M)$ stages, rather than the logarithmic number of stages of the Skew algorithm. At the beginning of stage h, we assume to have on disk *the array SA^h that contains the sorted sequence of the first hm suffixes of T*. Initially $h = 0$ and thus SA^0 is the empty array. In the generic h-th stage, The BGS algorithm loads the next text piece T^{h+1} in internal memory, builds SA' as the sorted sequence of suffixes starting in T^{h+1}, and then computes the new SA^{h+1} by merging the two sorted sequences SA^h and SA'.

There are two main issues when detailing this algorithmic idea is a running code: how to efficiently construct SA', since its suffixes start in T^{h+1} but may extend outside that string up to the end of T; and how to efficiently merge the two sorted sequences SA^h and SA', since they involve suffixes whose length may be up to $\Theta(n)$ characters. For the first issue BGS does not implement any special trick, it just compares pairs of suffixes character-by-character in $O(n)$ time and $O(n/B)$ I/Os. This means that over the total execution of the $O(n/M)$ stages, BGS takes $O(\frac{n}{B}\frac{n}{m}m\log m) = O(\frac{n^2}{B}\log m)$ I/Os to construct SA'.

The final merge between SA' with SA^h is executed by resorting the use of an auxiliary array $C[1, m+1]$ which counts in $C[j]$ the number of suffixes of SA^h that are lexicographically greater than the $SA'[j-1]$-th text suffix and smaller than the $SA'[j]$-th text suffix. Since SA^h is longer and longer, we need to process it by scan only, and thus devise a method that scans rightward the text T (from its beginning) and then searches each of its suffixes by binary-search in SA'. If the the lexicographic position is j, then the entry $C[j]$ is incremented. The binary search may involve a part of a suffix of

SA' which lies outside the internal memory, thus taking again $O(n/B)$ I/Os per binary-search step. Over all the n/M stages, this binary search takes $O(\sum_{h=0}^{n/m-1} \frac{n}{B}(hm)\log m) = O(\frac{n^3}{MB}\log M)$ I/Os.

Array C is then exploited in the next substep to quickly merge the two arrays SA' and SA^h: $C[j]$ indicates how many consecutive suffixes of SA^h lexicographically lie after $SA'[j-1]$ and before $SA'[j]$. Hence a disk scan of SA^h suffices to perform the merging process in $O(n/B)$ I/Os.

THEOREM 7.3 *The BGS algorithm builds the suffix array of a string $T[1,n]$ in $O(\frac{n^3}{MB}\log M)$ I/Os and $O(n/B)$ disk pages.*

Since the worst-case number of total I/Os is cubic, a purely theoretical analysis would classify this algorithm as not much interesting. However, in practical situations it is very reasonable to assume that each suffix comparison finds in internal memory all the characters used to compare the two involved suffixes. The practical behavior could be described more precisely by the formula $O(\frac{n^2}{MB})$ I/Os. Additionally, all I/Os in this analysis are sequential and the actual number of random seeks is only $O(n/M)$ (i.e., at most a constant number per stage). Consequently, the algorithm takes fully advantage of the large bandwidth of current disks and of the high speed of current CPUs.

Before detailing a significant improvement for the previous approach, let us concentrate on a running example, useful to fix the BGS' ideas:

	1	2	3	4	5	6	7	8	9	10	11	12
$T[1,12] =$	m	i	s	s	i	s	s	i	p	p	i	$

Suppose that $m = 3$ and that, at the beginning of stage $h = 1$, the algorithm has stored on the disk the array $SA^1 = (2,1,3)$ which corresponds to the lexicographic order of the text suffixes: "mississippi$", "ississippi$" and "ssissippi$". During the stage $h = 1$, the algorithm loads in internal memory $T^1 = T[4,6] = sis$ and lexicographically sorts the text suffixes which start in that substring, but may end up at the end of T, see figure 7.3.

Text suffixes	`sissippi$`	`issippi$`	`ssippi$`
		⇓	
		Lexicographic ordering	
		⇓	
Sorted suffixes	`issippi$`	`sissippi$`	`ssippi$`
SA′	5	4	6

FIGURE 7.3: Stage 1 of the BGS algorithm.

The figure shows that the comparison between the text suffixes: $T[4,12]$ = "sissippi$" and $T[6,12]$ = "ssippi$" involves characters that lie outside the text piece $T[4,6]$ loaded in internal memory, so that their comparison induces some I/Os.

The final step merges $SA^1 = (2,1,3)$ with $SA' = (5,4,6)$, in order to compute SA^2. This step uses the information of the counter array C. For example $C[1] = 2$ because two suffixes $T[1,12]$ = "mississippi$" and $T[2,12]$ = "ississippi$" are between the $SA'[0]$-th suffix "issippi$" and the $SA'[1]$-th suffix "sissippi$".

The second stage and the third stage are summarized in figures 7.5 and 7.6.

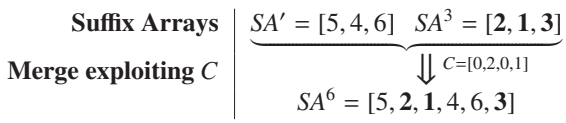
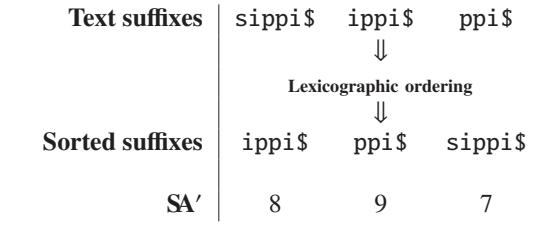


FIGURE 7.4: Example BGS. Stage 1 - Step 3

Stage 2*:*

(1) Load into internal memory the text substring $T^2 = T[7, 9] = sip$.

(2) Build SA' by sorting lexicographically the text suffixes which start in T^2:



(3) Merge SA' with SA^2 exploiting C:

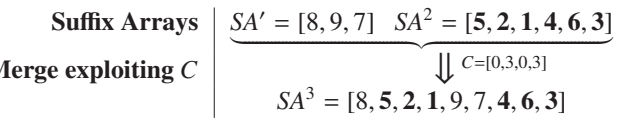


FIGURE 7.5: Stage 2 in the BGS algorithm for the string $T[1, 12]$ = "mississippi$".

The performance of BGS can be improved via a simple observation [2]. Assume that, at the beginning of stage h, in addition to the SA^h we have on disk a bit array, called gt_h, such that $\mathsf{gt}_h[i] = 1$ if and only if the suffix $T[(hm + 1) + i, n]$ is greater than the suffix $T[(hm + 1), n]$. During the h-th stage the algorithm loads into internal memory the substring $t = T^h T^{h+1}$ and the binary array $\mathsf{gt}_{h+1}[1, m - 1]$. Then it builds SA' by deploying the two arrays above and without performing any other I/Os. The key observation here is that two suffixes starting at positions i and j of T^h, with $i < j$, can be compared lexicographically by looking at their characters in t, namely at the strings $t[i, m]$ and $t[j, j + m - i]$. These two strings have the same length and are completely in $t[1, 2m]$, hence in internal memory. If these strings differ we are done; otherwise, the order between these two suffixes is determined by the order of the suffixes starting at the characters $t''[m + 1]$ and $t''[j + m - i + 1]$. This order is given by the bit stored in $\mathsf{gt}_{h+1}[j - i]$, also available in internal memory.

This argument shows that t'' and gt_{h+1} contain all the information we need to build SA^{h+1} working in internal memory.

THEOREM 7.4 *The new BGS algorithm builds the suffix array of a string $T[1, n]$ in $O(\frac{n^2}{MB})$ I/Os and $O(n/B)$ disk pages.*

Stage 3*:*

(1) Load into internal memory the text substring $T^3 = T[10, 12] = pi\$$.

(2) Build SA' by sorting lexicographically the text suffixes which start in T^3:

Text suffixes	pi$	i$	$
	⇓		
	Lexicographic ordering		
	⇓		
Sorted suffixes	$	i$	pi$
SA′	12	11	10

(3) Merge SA' with SA^3 exploiting C:

Suffix Arrays	$\underbrace{SA' = [12, 11, 10] \quad SA^3 = [\mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]}$
Merge exploiting C	⇓ C=[0,0,4,5]
	$SA^4 = [12, 11, \mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, 10, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

FIGURE 7.6: Stage 2 in the BGS algorithm for the string $T[1, 12]$ = "mississippi$".

We detail the running of the BGS algorithm on the same example used for the Skew algorithm, namely the string T = "mississippi$". At the stage $h = 1$ we read SA^0, which is empty, build SA' and then merge the two arrays to obtain SA^2. The construction of SA' deploys the availability in internal memory of $\mathsf{gt}_2 = (1, 0)$ and $t = T^h T^{h+1} = sissip$. In fact, given $i < j$ with $1 \le i \le 3$ we compare lexicographically the substrings $t''[i, m]$ and $t''[j, j + m - i]$ by looking just at those two arrays. As an example, take $i = 1$ and $j = 3$ so we are comparing $t[i, m]$ = "sis" with $t[j, j + m - i + 1]$ = "ssi". The strings are different so we obtain the following lexicographic order: "iss" ¡ "sis". Now take $i = 3$ and $j = 4$ so we have to compare the two equal suffixes "s" and "**s**". The strings are not different so we use $\mathsf{gt}_2[j - i] = \mathsf{gt}_2[1] = 1$, hence the "**s**" is lexicographically greater than "s" and this can be computed without any I/Os. It remains to show that the computation of gt can occur efficiently, and this is a technicality shown in [2].

7.5 Lcp construction

Surprisingly enough the longest common prefix array, shortly *lcp*, can be derived from the input string and its suffix array in linear time. This time bound can be obtained only by avoid the re-scanning of the characters of the input string. In fact, if we compute *lcp* via the pairwise comparison character-by-character of the $n - 1$ contiguous suffix pairs of SA, we would get a time complexity of $O(n^2)$ in the worst case. The algorithm we describe below has been proposed in 2001 by Kasai *et al* [5], it is elegant and optimal in time and space.

For the sake of presentation we will refer to Figure 7.7 which illustrates clearly the main algorithmic idea. Let us concentrate on two consecutive text suffixes, say $suff_{i-1}$ and $suff_i$, which occur at positions p and q in the suffix array. And assume that the value of $lcp[p]$ storing the lcp between $SA[p] = suff_{i-1}$ and its previous suffix $SA[p - 1] = suff_{j-1}$ is known. Our goal is to show that $lcp[q]$ storing the lcp between $SA[q] = suff_i$ and its previous suffix $SA[q - 1] = suff_k$ can be com-

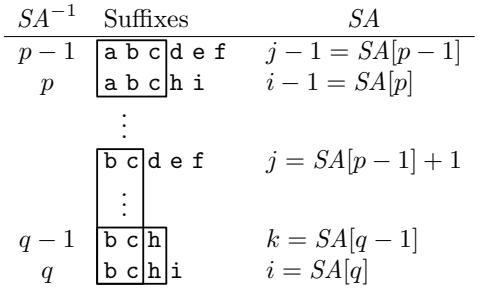


FIGURE 7.7: Relation between suffixes and Lcp values in the Kasai's method.

puted without re-scanning these suffixes from scratch but can start where the comparison between $SA[p-1]$ and $SA[p]$ ended. This will ensure that re-scanning is avoided and will lead to a linear time complexity.

We need the following property that we already mentioned when dealing with prefix search, and that we restate here in the context of the suffix array.

FACT 7.1 $lcp(suff_{SA[y-1]}, suff_{SA[y]}) \geq lcp(suff_{SA[x]}, suff_{SA[y]}), \quad \forall x < y$

Proof This property derives from the observation that suffixes in SA are ordered lexicographically, so that as we go farther from $SA[y]$ we reduce the length of the shared prefix. ∎

Let us now refer to Figure 7.7, concentrate on the pair of suffixes $suff_{j-1}$ and $suff_{i-1}$, and take their next suffixes $suff_j$ and $suff_i$. There are two possible cases: Either they share some characters in their prefix, i.e. $lcp[p] > 0$, or they do not. In the former case we can conclude that, since $suff_{j-1} < suff_{i-1}$, the next suffixes preserve that lexicographic order, so that $suff_j < suff_i$, and moreover $lcp(suff_j, suff_i) = lcp[p] - 1$. In fact, the first shared character is dropped but the next $lcp[p] - 1$ shared characters (possibly none) remain, as well as their mismatch character that drives the lexicographic order, which is therefore preserved. In the Figure above, we have $lcp[p] = 3$, so when we consider the next suffixes their lcp is 2, their order is preserved (as indeed $suff_j$ occurs before $suff_i$) although now they lie not adjacent in SA.

FACT 7.2 *If* $lcp(suff_{SA[y-1]}, suff_{SA[y]}) > 0$ *then:*

$$lcp(suff_{SA[y-1]+1}, suff_{SA[y]+1}) = lcp(suff_{SA[y-1]}, suff_{SA[y]}) - 1$$

By Fact 7.1 and Fact 7.2, we can conclude the key property deployed by Kasai's algorithm:

FACT 7.3 *According to the notation above, it is*

$$lcp(suff_{SA[q-1]}, suff_{SA[q]}) \geq lcp(suff_{SA[p-1]}, suff_{SA[p]}) - 1$$

This fact can be rephrased shortly as $lcp[q] = \max\{lcp[p] - 1, 0\}$. So this algorithmically shows that the computation of $lcp[q]$ can take full advantage of what we compared for the computation of $lcp[p]$. By adding to this the fact that we are processing the text suffixes rightward, we can conclude that the characters involved in the suffix comparisons move themselves rightward and, since re-scanning is avoided, their total number is $O(n)$. A sketch of the Kasai's algorithm is shown in figure

7.3, where we make use of the inverse suffix array, denoted by SA^{-1}, which returns for every suffix its position in SA.

Algorithm 7.3 LCP-BUILD(char $*T$, int n, char $**SA$)

```
1:  for (i = 0; i < n, i++) do
2:      i = SA[SA^{-1}[i]];
3:  end for
4:  h = 0;
5:  for (i = 0; i < n, i++) do
6:      p = SA^{-1}[p];
7:      if (p > 0) then
8:          k = SA[p − 1];
9:          if (h > 0) then
10:             h–;
11:         end if
12:         while (T[k + h] == T[i + h]) do
13:             h++;
14:         end whilelcp[p] = h;
15:     end if
16: end for
```

Step 5 checks whether $suff_p$ occupies the first position of the suffix array, in this case the lcp with the previous suffix is undefined. The **for**-loop then scans the text suffixes $suff_i$ from left to right, and for each of them first retrieves its position in SA, namely $i = SA[p]$, and its preceding suffix in SA, namely $k = SA[p-1]$. Then extends possibly the longest common prefix starting from the offset determined for $suff_{i-1}$. This is the algorithmic application of Fact 7.3.

As far as the time complexity is concerned, we notice that h is decreased at most n times (once per iteration of the for-loop), and it cannot move outside T (within each iteration of the for-loop), so $h \leq n$. This implies that h can be increased at most $2n$ times and this is the upper bound to the number of character comparisons executed by the above algorithm. The total time complexity is therefore $O(n)$. Moreover, since the algorithm uses the arrays SA^{-1} its additional working space is of n integers.

We conclude this section by noticing that an I/O-efficient algorithm to compute the lcp-array is still missing.

References

[1] Martin Farach-Colton, Paolo Ferragina, S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6): 987-1011, 2000.

[2] Paolo Ferragina and Travis Gagie and Giovanni Manzini. Lightweight data indexing and compression in external memory. In *Procs of the Symposium on Theoretical Informatics (LATIN)*, Lecture Notes in Computer Science vol. 6034, Springer, 697–710, 2010.

[3] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82, Prentice-Hall, 1992.

[4] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction.

In *Procs of the International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science vol. 2791, Springer, 943–955, 2003.

[5] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Procs of the Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science vol. 2089, Springer, 181–192, 2001.

[6] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.