# 7

# Searching Strings by Substring


7.1 Notation and terminology ............................... 7-1
7.2 The Suffix Array ........................................... 7-2
The substring-search problem • The LCP-array and its construction • Suffix-array construction
7.3 The Suffix Tree ............................................ 7-15
The substring-search problem • Construction from Suffix Arrays and vice versa • McCreight's algorithm$^\infty$
7.4 Some interesting problems .............................. 7-22
Approximate pattern matching • Text Compression • Text Mining


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

> **The substring-search problem.** *Given a text string $T[1,n]$, drawn from an alphabet of size $\sigma$, retrieve (or just count) all text positions where a query pattern $P[1,p]$ 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 subject to a massive number of queries, which is the scenario involving 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.

In this lecture we will describe to main approaches to substring searching, one based on arrays and another one based on trees, that mimic what we have done for the prefix-search problem. The two approaches hinge on the use of two fundamental data structures: the *suffix array* (shortly *SA*) and the *suffix tree* (shortly *ST*). We will describe in much detail those data structures because their use goes far beyond the context of full-text search.

## 7.1 Notation and terminology

We assume that text $T$ ends with a special character $T[n] = \$$, 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 suffix. We use $suff_i$ to denote the $i$-th suffix of text $T$, 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 state that $P$ is a prefix of the $i$-th text suffix, namely $P$ is a prefix of the string $suff_i$.

© Paolo Ferragina, 2009-2012

As an example, if $P$ ="siss" and $T$ ="mississippi$", then $P$ occurs at text position 4 and indeed it prefixes the suffix $suff_4 = T[4, 12]$ ="sissippi$". For simplicity of exposition, and for historical reasons, we will use this text as running example; nevertheless we point out that a text may be an arbitrary sequence of characters, hence not necessarily a single word.

Given the above observation, we can form with all text suffixes the dictionary $SUF(T)$ and state that *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 lexicographically sorted $SUF(T)$,
2. the lexicographic position of $P$ in $SUF(T)$ immediately precedes the block of suffixes prefixed by $P$.

An attentive reader may have noticed that these are the properties we deployed 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. What is challenging is the efficient construction of these data structures and their mapping onto disk to achieve efficient I/O-performance. These will be the main issues dealt with in this lecture.

| 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 for a text $T$ is the array of pointers to all text suffixes ordered lexicographically. We use the notation $SA(T)$ to denote the suffix array built over $T$, or just $SA$ if the indexed text is clear from the context. Because of the lexicographic ordering, $SA[i]$ is the $i$-th smallest text suffix, so we have that $suff_{SA[1]} < suff_{SA[2]} < \cdots < suff_{SA[n]}$, where $<$ is the lexicographical order between strings. For space reasons, each suffix is represented by its starting position in $T$ (i.e. an integer). $SA$ consists of $n$ integers in the range $[1, n]$ and hence it occupies $O(n \log n)$ bits.

Another useful concept is the *longest common prefix* between two consecutive suffixes $suff_{SA[i]}$ and $suff_{SA[i+1]}$. We use `lcp` to denote the array of integers representing the lengths of those lcps. Array `lcp` consists of $n-1$ entries containing values smaller than $n$. There is an optimal and non obvious linear-time algorithm to build the *lcp*-array which will be detailed in Section 7.2.3. The interest in `lcp` rests in its usefulness to design efficient/optimal algorithms to solve various search and mining problems over strings.

### 7.2.1 The substring-search problem

We observed that this problem can be reduced to a prefix search over the string dictionary $SUF(T)$, so it can be solved by means of a binary search for $P$ over the array of text suffixes ordered lexicographically, hence $SA(T)$. Figure 7.1 shows the pseudo-code which coincides with the classic binary-search algorithm specialized to compare strings rather than numbers.

**Algorithm 7.1** SubstringSearch($P, SA(T)$)

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

A binary search in $SA$ requires $O(\log n)$ string comparisons, each taking $O(p)$ time in the worst case. Thus, the time complexity of this algorithm 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`$[1, n-1]$ array;
- two other arrays $Llcp[1, n-1]$ and $Rlcp[1, n-1]$ which are defined for every triple $(L, M, R)$ that may arise in the inner loop of a binary search. We define $Llcp[M] =$ `lcp`$(suff_{SA[L_M]}, suff_{SA[M]})$ and $Rlcp[M] =$ `lcp`$(suff_{SA[M]}, suff_{SA[R_M]})$, namely $Llcp[M]$ accounts for the prefix shared by the leftmost and the middle suffix of the range currently explored by the binary search; $Rlcp[M]$ accounts for the prefix shared by the rightmost and the middle suffix of that range.

We notice that each triple $(L, M, R)$ is uniquely identified by its midpoint $M$ because the execution of a binary search defines actually a hierarchical partition of the array $SA$ into smaller and smaller sub-arrays delimited by $(L, R)$ and thus centered in $M$. Hence we have $O(n)$ triples overall, and these three arrays occupy $O(n)$ space in total.

We can build arrays $Llcp$ and $Rlcp$ in linear time by exploiting one of three different approaches. We can deploy the observation that the `lcp`$[i, j]$ between the two suffixes $suff_{SA[i]}$ and $suff_{SA[j]}$ can be computed as the minimum of a range of `lcp`-values, namely `lcp`$[i, j] = \min_{k=i,\dots,j-1}$ `lcp`$[k]$. By

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\Longrightarrow$ | `$` | | `$` | | `$` | | `$` |
| $\vert$ | `i$` | | `i$` | | `i$` | | `i$` |
| $\vert$ | `ippi$` | | `ippi$` | | `ippi$` | | `ippi$` |
| $\vert$ | `issippi$` | | `issippi$` | | `issippi$` | | `issippi$` |
| $\vert$ | `ississippi$` | | `ississippi$` | | `ississippi$` | | `ississippi$` |
| $\vert\rightarrow$ | `mississippi$` | | `mississippi$` | | `mississippi$` | | `mississippi$` |
| $\vert$ | `pi$` | $\Longrightarrow$ | `pi$` | | `pi$` | | `pi$` |
| $\vert$ | `ppi$` | $\vert$ | `ppi$` | | `ppi$` | | `ppi$` |
| $\vert$ | `sippi$` | $\vert$ | `sippi$` | | `sippi$` | | `sippi$` |
| $\vert$ | `sissippi$` | $\vert\rightarrow$ | `sissippi$` | | `sissippi$` | | `sissippi$` |
| $\vert$ | `ssippi$` | $\vert$ | `ssippi$` | $\Longrightarrow$ | `ssippi$` | $\Longrightarrow$ | `ssippi$` |
| $\Longrightarrow$ | `ssissippi$` | $\Longrightarrow$ | `ssissippi$` | $\Longrightarrow$ | `ssissippi$` | | `ssissippi$` |
| | Step (1) | | Step (2) | | Step (3) | | Step (3) |

FIGURE 7.2: Binary search steps for the lexicographic position of the pattern $P$ =“ssi” in “mississippi$”.

associativity of the min we can split the computation as $\texttt{lcp}[i, j] = \min\{\texttt{lcp}[i, k], \texttt{lcp}[k, j]\}$ where $k$ is any index in the range $[i, j]$, so in particular we can set $\texttt{lcp}[L, R] = \min\{\texttt{lcp}[L, M], \texttt{lcp}[M, R]\}$. This implies that the arrays *Llcp* and *Rlcp* can be computed via a bottom-up traversal of the triplets $(L, M, R)$ in $O(n)$ time. Another way to deploy the previous observation is to compute $\texttt{lcp}[i, j]$ on-the-fly via a Range-Minimum Data structure built over the array `lcp` (see Section **??**); or with the elegant approach proposed in 2001 by Kasai *et al* and described below. All of these approaches take $O(n)$ time and space, and thus they are optimal.

We are left with showing how the binary search can be speeded up by using these arrays. Consider a binary search iteration on the sub-array $SA[L, R]$, and let $M$ be the midpoint of this range. A lexicographic comparison between $P$ and $suff_{SA[M]}$ has to be made in order to choose the next search-range between $SA[L, M]$ and $SA[M, R]$. Assume that we know inductively the values $l = \texttt{lcp}(P, suff_{SA[L]})$ and $r = \texttt{lcp}(P, suff_{SA[R]})$ which denote the number of characters the pattern $P$ shares with the strings at the extreme of the range currently explored by the binary search. At the first step, in which $L = 0$ and $R = n - 1$, these two values can be computed in $O(p)$ time by comparing character-by-character the involved strings, and thus they can be assumed to be known. At a generic step, $P$ lies between $suff_{SA[L]}$ and $suff_{SA[R]}$, so it surely shares $k = \texttt{lcp}[L, R]$ characters with these suffixes given that all suffixes in this range share this number of characters. Therefore larger than $k$ are the lcp-values $l, r$, as well as larger than $k$ is the number of characters $m$ that the pattern $P$ shares with $suff_{SA[M]}$. We could then take advantage of the inequality $m \geq k$ and thus compute $m = \texttt{lcp}(P, suff_{SA[M]})$ starting from their $(k + 1)$-th character. But actually we can do better because we know $r$ and $l$, and these values can be significantly larger than $k$, thus more characters of $P$ have been already involved in previous comparisons and so they are known.

We distinguish two main cases. If $l = r$ then all suffixes in the range $[L, R]$ share (at least) $l$ characters (hence $suff_{SA[M]}$ too) which are equal to $P[1, l]$. 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 we can set $m = l$. In fact, by induction, $P > suff_{SA[L]}$ and their mismatch lies at position $l + 1$. By definition of $Llcp[M]$ and the hypothesis, we have that $suff_{SA[L]}$ shares more than $l$ characters with $suff_{SA[M]}$. So the mismatch between $P$ and $suff_{SA[M]}$ is the same as it is with $suff_{SA[L]}$, hence their compari-

son gives the same answer— i.e. $P > suff_{SA[M]}$— and the search can thus continue in the subrange $SA[M, R]$. We remark that this case does 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 with $suff_{SA[L]}$ and $suff_{SA[M]}$. So the comparison between $P$ and $suff_{SA[M]}$ can start from their $(l+1)$-th character. Eventually we determine $m$ and their lexicographic order. Here some character comparisons are executed, but the *knowledge* about $P$'s characters advanced too.

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

**LEMMA 7.1** Given the three arrays `lcp`, *Llcp* and *Rlcp* built over a text $T[1, n]$, we can count the occurrences of a pattern $P[1, p]$ in the text taking $O(p + \log n)$ time in the worst case. Retrieving the positions of these *occ* occurrences takes additional $O(occ)$ time. The total required space is $O(n)$.

**Proof** We remind that searching for all strings having the 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 easy to count the pattern occurrences in constant time, as $occ = j - i + 1$, or print all of them in $O(occ)$ time. ■

### 7.2.2 The LCP-array and its construction

Surprisingly enough the longest common prefix array $\texttt{lcp}[1, n-1]$ can be derived from the input string $T$ and its suffix array $SA[1, n]$ in optimal linear time.[1] This time bound cannot be obtained by the simple approach that compares character-by-character the $n-1$ contiguous pairs of text suffixes in $SA$; as this takes $\Theta(n^2)$ time in the worst case. The optimal $O(n)$ time needs to avoid the re-scanning of the text characters, so some property of the input text has to be proved and deployed in the design of an algorithm that achieves this complexity. This is exactly what Kasai *et al* did in 2001 [6], their algorithm is elegant, deceptively simple, and optimal in time and space.

For the sake of presentation we will refer to Figure 7.3 which illustrates clearly the main algorithmic idea. Let us concentrate on two consecutive suffixes in the text $T$, say $suff_{i-1}$ and $suff_i$, which occur at positions $p$ and $q$ in the suffix array $SA$. And assume that we know inductively the value of $\texttt{lcp}[p-1]$, storing the longest common prefix between $SA[p-1] = suff_{j-1}$ and the next suffix $SA[p] = suff_{i-1}$ in the lexicographic order. Our goal is to show that $\texttt{lcp}[q-1]$ storing the longest common prefix between suffix $SA[q-1] = suff_k$ and the next ordered suffix $SA[q] = suff_i$, which interests us, can be computed without re-scanning these suffixes from their first character but can start where the comparison between $SA[p-1]$ and $SA[p]$ ended. This will ensure that re-scanning of text characters is avoided, precisely it is avoided the re-scanning of $suff_{i-1}$, and as a result we will get a linear time complexity.

[1] Recall that $\texttt{lcp}[i] = \texttt{lcp}(suff_{SA[i]}, suff_{SA[i+1]})$ for $i < n$.

| **Sorted Suffixes** | *SA* | *SA* positions |
|---|---|---|
| **abc**def | $j-1$ | $p-1$ |
| **abc**hi | $i-1$ | $p$ |
| . | . | . |
| . | . | . |
| . | . | . |
| **bc**def | $j$ | |
| . | . | . |
| . | . | . |
| . | . | . |
| **bc**h | $k$ | $q-1$ |
| **bc**hi | $i$ | $q$ |

FIGURE 7.3: Relation between suffixes and lcp values in the Kasai's algorithm.

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

***FACT 7.1*** *For any position $x < y$ it holds $\texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) \geq \texttt{lcp}(suff_{SA[x]}, suff_{SA[y]})$.*

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

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

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

By Fact 7.1 and Fact 7.2, we can conclude the key property deployed by Kasai's algorithm: $\texttt{lcp}[q-1] = \max\{\texttt{lcp}[p-1]-1, 0\}$. This algorithmically shows that the computation of $\texttt{lcp}[q-1]$ can take full advantage of what we compared for the computation of $\texttt{lcp}[p-1]$. 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 7.2, where we make use of the inverse suffix array, denoted by $SA^{-1}$, which returns for every suffix its position in *SA*. Referring to Figure 7.3, we have that $SA^{-1}[i] = p$.

Step 4 checks whether $suff_q$ occupies the first position of the suffix array, in which case the lcp with the previous suffix is undefined. The **for**-loop then scans the text suffixes $suff_i$ from left to right,

**Algorithm 7.2** LCP-Build(char *$T$, int $n$, char **$SA$)

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

and for each of them it first retrieves the position of $suff_i$ in $SA$, namely $i = SA[q]$, and its preceding suffix in $SA$, namely $k = SA[q-1]$. Then it extends their longest common prefix starting from the offset $h$ determined for $suff_{i-1}$ via character-by-character comparison. This is the algorithmic application of the above observations.

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 Kasai's algorithm. The total time complexity is therefore $O(n)$.

We conclude this section by noticing that an I/O-efficient algorithm to compute the $lcp$-array is still missing in the literature, some heuristics are known to reduce the number of I/Os incurred by the above computation but an optimal $O(n/B)$ I/O-bound is yet to come, if possible.

### 7.2.3 Suffix-array construction

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 and specialize the comparison-function in such a way that it computes the lexicographic order between strings. `Algorithm 7.3` implements this idea in C-style using the built-in procedure Qsort as sorter and a properly-defined subroutine Suffix_cmp for comparing suffixes:

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

Notice that the suffix array is initialized with the pointers to the real starting positions in memory of the suffixes to be sorted, and not the integer offsets from 1 to $n$ as stated in the formal description of $SA$ of the previous pages. The reason is that in this way Suffix_cmp does not need to know $T$'s position in memory (which would have needed a global parameter) because its actual parameters passed during an invocation provide the starting positions in memory of the suffixes to be compared. Moreover, the suffix array $SA$ has indexes starting from 0 as it is typical of C-language.

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

**Algorithm 7.3** 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);
```

***THEOREM 7.1*** *In the worst case the use of a comparison-based sorter to construct the suffix array of a given 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 7.2.3 we describe a Divide-and-Conquer algorithm— the *Skew* algorithm proposed by Kärkkäinen and Sanders [5]— which is elegant, easy to code, and flexible enough to achieve the optimal I/O-bound in various models of computations. In Section **??** we describe another algorithm— the *Scan-based* algorithm proposed by BaezaYates, Gonnet and Sniders [4]— which is also simple, but incurs in a larger number of I/Os; we nonetheless introduce this algorithm because it offers the positive feature of processing the input data in passes (streaming-like) thus forcing pre-fetching, allows compression and hence it turns to be suitable for slow disks.

### The Skew Algorithm

In 2003 Kärkkäinen and Sanders [5] showed that the problem of constructing suffix-arrays can be *reduced* to the problem of sorting a set of triplets whose components are integers in the range $[1, O(n)]$. Surprisingly this reduction takes *linear time and space* thus turning the complexity of suffix-array construction into the complexity of sorting atomic items, a problem about which we discussed deeply in the previous chapters and for which we know optimal algorithms for hierarchical memories. In addition we can observe that, since the items to be sorted are integers bounded in value by $O(n)$, the sorting of the triplets takes $O(n)$ time in the RAM model, so this is the optimal time complexity of suffix-array construction in RAM. Really impressive!

This algorithm is named *Skew* in the literature, and it works in every model of computation for which an efficient sorting primitive is available: disk, distributed, parallel. 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.

For the sake of presentation we use $T[1,n] = t_1 t_2 \dots t_n$ to denote the input string drawn from the alphabet $\Sigma = \{1, 2, \dots, \sigma\}$. We can assume that $\sigma = O(n)$ otherwise we can sort the characters of $T$ and rename them with integers in $O(n)$, taking overall $O(n\log\sigma)$ time in the worst-case. Furthermore we assume that $t_n = \$$ and logically pad $T$ with an infinite number of occurrences of that special character which is assumed smaller than any other alphabet character. Given this notation, we can sketch the three main steps of the Skew algorithm:

**Step 1.** Construct the suffix array limited to the suffixes starting at positions $P_{2,0} = \{i : i \bmod 3 = 2, \text{ or } i \bmod 3 = 0\}$:

- This is done by building the string $T^{2,0}$ of length $(2/3)n$ which compactly encodes all suffixes of $T$ starting at positions $P_{2,0}$.
- And then running the suffix-array construction algorithm recursively over it. The result is the suffix array $SA^{2,0}$, which actually corresponds to the lexicographically sorted sequence of text suffixes starting at positions $P_{2,0}$.

**Step 2** Construct the suffix array of the remaining text suffixes starting at positions $P_1 = \{i : i \bmod 3 = 1\}$:

- This is done by representing every text suffix $T[i, n]$ with a pair $\langle T[i], \texttt{pos}(i+1)\rangle$, where we have that $i + 1 \in P_{2,0}$ and $\texttt{pos}(i + 1)$ is the position of the $(i + 1)$-th text suffix in $SA^{2,0}$.
- 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 decomposition $\frac{2}{3} : \frac{1}{3}$ which ensures a constant-time lexicographic comparison between any pair of suffixes (see details below).

The execution of the algorithm is illustrated using the string $T[1, 12]$ ="mississippi$", where the final suffix array will be $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$. In this example we have: $P_{2,0} = \{2, 3, 5, 6, 8, 9, 11, 12\}$ and $P_1 = \{1, 4, 7, 10\}$.

**Step 1.** The first step is the most involved one and constitutes the backbone of the entire recursive process. It lexicographically sorts the suffixes starting at the text positions $P_{2,0}$. The resulting array is denoted by $SA^{2,0}$ and represents a *sampled* version of the final suffix array $SA$ because it corresponds to the suffixes starting at positions $P_{2,0}$.

To efficiently obtain $SA^{2,0}$, we reduce the problem to the construction of the suffix array for a string $T^{2,0}$ of length about $\frac{2n}{3}$. This way the construction can occur recursively without impairing in the final time complexity. The key difficulty is how to define $T^{2,0}$ so that its suffix array corresponds to the sorted sequence of text suffixes starting at the positions in $P_{2,0}$. The elegant solution consists of constructing the two strings $R_k = [t_k, t_{k+1}, t_{k+2}][t_{k+3}, t_{k+4}, t_{k+5}] \ldots]$, for $k = 2, 3$. These two strings are composed by triplets of symbols $[t_i, t_{i+1}, t_{i+2}]$ which therefore take $O(\log n)$ bits each, and hence occupy $O(1)$ memory cells each. With reference to the previous example, we have:

$$R_2 = \Big\{\underset{2}{[\texttt{i s s}]}\ \underset{5}{[\texttt{i s s}]}\ \underset{8}{[\texttt{i p p}]}\ \underset{11}{[\texttt{i \$ \$}]}\Big\} \qquad R_3 = \Big\{\underset{\mathbf{3}}{[\texttt{s s i}]}\ \underset{\mathbf{6}}{[\texttt{s s i}]}\ \underset{\mathbf{9}}{[\texttt{p p i}]}\Big\}$$

Notice that the infinite padding of $T$ with the special symbol \$ allows to force $R_2$ and $R_3$ to have multiple-of-three length. Notice that the triplet starting from the last text character $T[12]$ = \$, and thus consisting of only these special characters, is not considered.

We then construct $R = R_2 \bullet R_3$ as the string formed by concatenating the triplets of $R_2$ with the triplets of $R_0$.

$$R = \Big\{\underset{2}{[\texttt{i s s}]}\ \underset{5}{[\texttt{i s s}]}\ \underset{8}{[\texttt{i p p}]}\ \underset{11}{[\texttt{i \$ \$}]}\underset{3}{[\texttt{s s i}]}\ \underset{6}{[\texttt{s s i}]}\ \underset{9}{[\texttt{p p i}]}\Big\}$$

The key property on which the first step of the Skew algorithm hinges on, is the following:

**Property 7.2** *Every text suffix $T[i, n]$ starting at a position in $P_{2,0}$, can be put in correspondence with a suffix of R consisting of a sequence of triplets. Specifically, if i* $\texttt{mod}$ 3 = 0 *then the text suffix coincides exactly with a suffix of R; if i* $\texttt{mod}$ 3 = 2*, then the text suffix prefixes a suffix of R which nevertheless terminates with special symbol \$.*

The correctness of this property can be inferred by the previous running example, in fact the suffix $T[6, 11]$ = `ssippi` occurs at the second triplet of $R_0$ and thus constitutes the sixth suffix of $R$, when it is interpreted as a string of triplets. Vice versa, the suffix $T[8, 11]$ = `ippi` occurs at the third triplet of $R_2$ so that it prefixes the third suffix of $R$. Even if it is not a full suffix of $R$, we have that $T[8, 11]$ ends with two \$s, which will be useful when comparing suffixes of $R$ independently whether they start in $R_2$ or $R_0$.

In order to manage those triplets 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 triplets in $R$ and associating to each distinct triplet its *rank* in the lexicographic order. Since we have $O(n)$ triplets, each consisting of symbols in a range $[0, n]$, their radix sort takes $O(n)$ time. Ranks are numbered starting from 1, the value 0 is reserved to the triplets formed by all $s, such as the one starting at the last position $T[12]$, of our running example.

In our example, the sorted triplets 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 triplets |
| 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 ] | triplets |
| | 3 | 3 | 2 | 1 | 5 | 5 | 4 | $T^{2,0}$ (string of ranks) |

As a result of the naming of the triplets in $R$, we get the new text $T^{2,0} = 3321554$ whose length is about $\frac{2n}{3}$. Moreover, as at the beginning, each text *character* is an integer bounded by $O(n)$ so the radix-step for the triplets-naming takes $O(n)$ time at every recursive call.

It is evident from the discussion above that, since the ranks are assigned in the same order as the lexicographic order of their triplets, the lexicographic comparison between suffixes of $R$ (aligned to the triplets) equals the lexicographic comparison between suffixes of $T^{2,0}$. Moreover $T^{2,0}$ consists of $\frac{2n}{3}$ characters which are indeed integers smaller than $n$. So $SA^{2,0}$ can be obtained by building recursively the suffix array $T^{2,0}$.

There are two notes to be added at this point. The first one is that, if all symbols in $T^{2,0}$ are different, then we do not need to recurse because suffixes can be sorted by looking just at their first characters. The second observation is for programmers that should be careful in turning the suffix-positions in $T^{2,0}$ into the suffix positions in $T$ to get the final $SA^{2,0}$, because they must take into account the layout of the triplets of $R$.

In our running example $T^{2,0} = (3, 3, 2, 1, 5, 5, 4)$, and since not all ranks are distinct the algorithm is applied recursively computing the suffix-array $(4, 3, 2, 1, 7, 6, 5)$ of $T^{2,0}$. Clearly the entries of this array represent suffixes of triplets $T^{2,0}$, we can turn them into suffix positions in $T$ by computing their correspondence with the positions in $P_{2,0}$. This can be done via simple arithmetic operations, given the layout of the triplets in$T^{2,0}$, and obtains in our running example the suffix array $SA^{2,0} = (11, 8, 5, 2, 9, 6, 3)$.

**Step 2.** Once the suffix array $SA^{2,0}$ has been built (recursively), it is possible to sort lexicographically the remaining suffixes of $T$, namely the ones starting at the text positions $i \bmod 3 = 1$, in a simple way. We decompose a suffix $T[i, n]$ as composed by its first character $T[i]$ and its remaining suffix $T[i + 1, n]$. Since $i \in P_1$, the next position $i + 1 \in P_{2,0}$, and thus the suffix $T[i + 1, n]$ occurs in $SA^{2,0}$. We can then encode the suffix $T[i, n]$ with a pair of integers $\langle T[i], \texttt{pos}(i + 1)\rangle$, where $\texttt{pos}(i+1)$ denotes the lexicographic rank in $SA^{2,0}$ of the suffix $T[i+1, n]$. If $i+1 = n+1$ then we set $\texttt{pos}(n+1) = 0$ given that the character $ is assumed to be smaller than any other alphabet character.

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 computed in $O(n)$ time by radix-sorting the $O(n)$ pairs encoding its suffixes.

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

| | | | | | |
|---|---|---|---|---|---|
| Pairs/suffixes: | $\langle \texttt{m}, 4\rangle$ | $\langle \texttt{s}, 3\rangle$ | $\langle \texttt{s}, 2\rangle$ | $\langle \texttt{p}, 1\rangle$ | |
| | 1 | 4 | 7 | 10 | starting positions in $P_1$ |
| Sorted pairs/suffixes: | $\langle \texttt{m}, 4\rangle <$ | $\langle \texttt{p}, 1\rangle <$ | $\langle \texttt{s}, 2\rangle <$ | $\langle \texttt{s}, 3\rangle$ | |
| | 1 | 10 | 7 | 4 | $SA^1$ |

**Step 3.** The final step merges the two sorted arrays $SA^1$ and $SA^{2,0}$ in linear time by resorting an interesting observation which motivates the split $\frac{2}{3} : \frac{1}{3}$. Let us given two suffixes $T[i,n] \in SA^1$ and $T[j,n] \in SA^{2,0}$, which we wish to lexicographically compare for implementing the merge-step. 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. We deploy a decomposition idea similar to the one exploited in Step 2 above, 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 $j \,\texttt{mod}\, 3 = 2$ then we compare $T[j,n] = T[j]T[j{+}1,n]$ against $T[i,n] = T[i]T[i{+}1,n]$. Both suffixes $T[j+1,n]$ and $T[i+1,n]$ occur in $SA^{2,0}$ (given that their starting positions are congruent $0, 2$ modulo 3, respectively), so we can derive the above lexicographic comparison by comparing the pairs $\langle T[i], \texttt{pos}(i+1)\rangle$ and $\langle T[j], \texttt{pos}(j+1)\rangle$. This comparison takes $O(1)$ time, provided that the array `pos` is available.[2]
2. if $j \,\texttt{mod}\, 3 = 0$ 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 $SA^{2,0}$ (given that their starting positions are congruent $2, 0$ modulo 3, respectively), so we can derive the above lexicographic comparison by comparing the triples $\langle T[i], T[i+1], \texttt{pos}(i+2)\rangle$ and $\langle T[j], T[j{+}1], \texttt{pos}(j{+}2)\rangle$. This comparison takes $O(1)$ time, provided that the array `pos` is available.

In our running example we have that $T[8,11] < T[10,11]$, and in fact $\langle i, 5\rangle < \langle p, 1\rangle$. Also we have that $T[7,11] < T[6,11]$ 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, respectively. Conversely $(\star)$ denotes the starting position in $T$ of the suffix. Notice that, since we do not know with which suffix of $SA^{2,0}$ will be compared with a suffix of $SA^1$ during the merging process, for each of the latter suffixes we need to compute both representations $(\star\star)$ and $(\star\star\star)$, hence as a pair and as a triplet.[3]

| $SA^1$ | | | | $SA^{2,0}$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 10 | 7 | 4 | 11 | 8 | 5 | 2 | 9 | 6 | 3 | $(\star)$ |
| ⟨m,4⟩ | ⟨p,1⟩ | ⟨s,2⟩ | ⟨s,3⟩ | ⟨i,0⟩ | ⟨i,5⟩ | ⟨i,6⟩ | ⟨i,7⟩ | | | | $(\star\star)$ |
| ⟨m,i,7⟩ | ⟨p,i,0⟩ | ⟨s,i,5⟩ | ⟨s,i,6⟩ | | | | | ⟨p,p,1⟩ | ⟨s,s,2⟩ | ⟨s,s,3⟩ | $(\star\star\star)$ |

At the end of the merge step we obtain the final suffix array: $SA = (11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$. 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 triplets 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 RAM model, the time complexity of the Skew 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 $T^{2,0}$ whose length is $(2/3)n$. This recurrence has solution $T(n) = O(n)$, which is clearly optimal. For what concerns the disk model, the Skew algorithm can be implemented in

[2] Of course, the array `pos` can be derived from $SA^{2,0}$ in linear time, since it is its inverse.

[3] Recall that $\texttt{pos}(n+1) = 0$.

$O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os, that is the I/O-complexity of sorting $n$ atomic items. We have therefore proved the following.

**THEOREM 7.3** *The Skew algorithm builds the suffix array of a text string $T[1, n]$ in $O(Sort(n))$ I/Os and $O(n/B)$ disk pages. If the alphabet $\Sigma$ has size polynomial in n, the CPU time is $O(n)$.*

### The Scan-based Algorithm

Before the introduction of the Skew algorithm, the best known disk-based algorithm was the one proposed by Baeza-Yates, Gonnet and Sniders in 1992 [4]. It is also a divide&conquer algorithm whose divide step is strongly unbalanced, thus it executes a quadratic number of suffix comparisons which induce a *cubic* time complexity. Nevertheless the algorithm is 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. Then assume that the text $T[1, n]$ 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 algorithm computes *incrementally* the suffix array of $T$ 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 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 in a running code: how to efficiently construct $SA'$, since its suffixes start in $T^{h+1}$ but may extend outside that block of characters 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 the algorithm 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, the algorithm takes $O(\frac{n}{B}\frac{n}{m} m \log m) = O(\frac{n^2}{B} \log m)$ I/Os to construct $SA'$.

For the second issue, we note that the merge between $SA'$ with $SA^h$ is executed in a smart way 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. Two special cases occur if $j = 1, m + 1$: in the former case we assume that $SA'[0]$ is the empty suffix, in the latter case we assume that $SA'[m + 1]$ is a special suffix larger than any string. Since $SA^h$ is longer and longer, we process it streaming-like by devising 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 lexicographic position of the searched suffix is $j$, then the entry $C[j]$ is incremented. The binary search may involve a part of a suffix which lies outside the block $T^{h+1}$ currently in internal memory, thus taking $O(n/B)$ 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 $SA'$ (in internal memory) and $SA^h$ (on disk): $C[j]$ 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 7.4** *The Scan-based algorithm builds the suffix array of a text 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. And indeed the practical behavior of this algorithm is better described 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. As a final notice we remark that the suffix arrays $SA^h$ and the text $T$ are scanned sequentially, so some form of compression can be adopted to reduce the I/O-volume and thus further speed-up the underlying algorithm.

Before detailing a significant improvement for the previous approach, let us concentrate on the same running example used in the previous section to sketch the Skew algorithm.

| | 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 already processed the text block $T^0 = T[1,3] =$ `mis` and thus stored on disk the array $SA^1 = (2,1,3)$ which corresponds to the lexicographic order of the text suffixes which start in that block: namely, `mississippi$`, `ississippi$` and `ssissippi$`. During the stage $h = 1$, the algorithm loads in internal memory the next block $T^1 = T[4,6] = sis$ and lexicographically sorts the text suffixes which start in positions [4, 6], but may end up at the end of $T$, see figure 7.4.

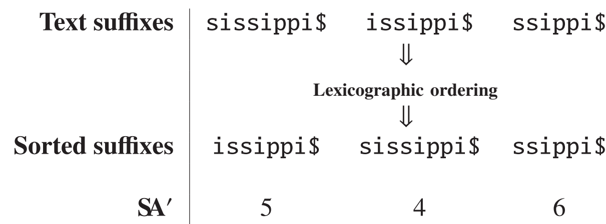



FIGURE 7.4: Stage 1 of the Scan-based algorithm.

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 next figure details the content of $C$ for the running example.

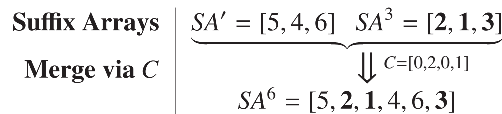



FIGURE 7.5: Stage 1, step 3, of the Scan-based algorithm

The second stage is summarized in Figure 7.6 where the text substring $T^2 = T[7,9] =$ `sip` is loaded in memory and the suffix array $SA'$ for the suffixes starting at positions $[7,9]$ is built. Then, the suffix array $SA'$ is merged with the suffix array on disk $SA^2$ containing the suffixes which start in $T[1,6]$.

*Stage* 2*:*

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

(2) Build $SA'$ for the suffixes starting in $[7,9]$:

| | | | |
|---|---|---|---|
| **Text suffixes** | `sippi$` | `ippi$` | `ppi$` |
| | | ⇓ **Lexicographic ordering** ⇓ | |
| **Sorted suffixes** | `ippi$` | `ppi$` | `sippi$` |
| **SA′** | 8 | 9 | 7 |

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

**Suffix Arrays** $SA' = [8,9,7]$ $SA^2 = [\mathbf{5},\mathbf{2},\mathbf{1},\mathbf{4},\mathbf{6},\mathbf{3}]$

**Merge via** $C$ ⇓ $C$=[0,3,0,3]

$SA^3 = [8,\mathbf{5},\mathbf{2},\mathbf{1},9,7,\mathbf{4},\mathbf{6},\mathbf{3}]$

FIGURE 7.6: Stage 2 of the Scan-based algorithm.

The second stage is summarized in Figure 7.7 where the last substring $T^2 = T[10,12] =$ `pi$` is loaded in memory and the suffix array $SA'$ for the suffixes starting at positions $[10,12]$ is built. Then, the suffix array $SA'$ is merged with the suffix array on disk $SA^3$ containing the suffixes which start in $T[1,9]$.

The performance of this algorithm can be improved via a simple observation [3]. Assume that, at the beginning of stage $h$, in addition to the $SA^h$ we have on disk a bit array, called $\mathsf{gt}_h$, such that $\mathsf{gt_h}[i] = 1$ if and only if the suffix $T[(hm+1)+i,n]$ is Greater Than the suffix $T[(hm+1),n]$. The computation of gt can occur efficiently, but this technicality is left to the original paper [3] and not detailed here.

During the $h$-th stage the algorithm loads into internal memory the substring $t[1,2m] = T^hT^{h+1}$ (so this is double in size with respect to the previous proposal) and the binary array $\mathsf{gt}_{h+1}[1,m-1]$ (so it refers to the second block of text loaded in internal memory). The key observation is that we can build $SA'$ by deploying the two arrays above without performing any I/Os. This seems surprising, but the key property the algorithm exploits here is that any two text suffixes starting at positions $i$ and $j$ within $T^h$, with $i < j$, can be compared lexicographically by looking at their characters in the substring $t$, 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, their order is determined and we are done; otherwise, the order between these two suffixes is determined by the order of the remaining suffixes starting at the characters $t[m+1]$ and $t[j+m-i+1]$. This order is

*Stage* 3*:*

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

(2) Build $SA'$ for the suffixes starting in [10, 12]:

| | |
|---|---|
| **Text suffixes** | `pi$` `i$` `$` |
| | ⇓ **Lexicographic ordering** ⇓ |
| **Sorted suffixes** | `$` `i$` `pi$` |
| **SA′** | 12 11 10 |

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

| | |
|---|---|
| **Suffix Arrays** | $SA' = [12, 11, 10]$ $SA^3 = [\mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$ |
| **Merge via** $C$ | ⇓ $C$=[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 7.7: Stage 3 of the Scan-based algorithm.

given by the bit stored in $\mathsf{gt}_{h+1}[j - i]$, also available in internal memory.

This argument shows that the two arrays $t$ and $\mathsf{gt}_{h+1}$ contain all the information we need to build $SA^{h+1}$ working in internal memory, and thus without performing any I/Os.

***THEOREM 7.5*** *The new variant of the Scan-based 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.*

As an example consider stage $h = 1$ and thus load in memory the text substring $t = T^h T^{h+1} = T[4, 9]$ = `sis sip` and the array $\mathsf{gt}_2 = (1, 0)$. Now consider the positions $i = 1$ and $j = 3$ in $t$, we can compare the text suffixes starting at these positions by first taking the substrings $t[1, 3] = T[4, 6]$ = `sis` with $t[3, 5] = T[6, 9]$ `ssi`. The strings are different so we obtain their order without accessing the disk. Now consider the positions $i = 3$ and $j = 4$ in $t$, they would not be taken into account by the algorithm since the block has size 3, but let us consider them for the sake of explanation. We can compare the text suffixes starting at these positions by first taking the substrings $t[3, 3]$ = `s` with $t[4, 4]$ = `s`. The strings are not different so we use $\mathsf{gt}_2[j - i] = \mathsf{gt}_2[1] = 1$, hence the remaining suffix $T[4, n]$ is lexicographically greater than $T[5, n]$ and this can be determined again without any I/Os.

## 7.3 The Suffix Tree

The *suffix tree* is a fundamental data structure used in many algorithms processing strings [**?**]. In its essence it is a compacted trie that stores all suffixes of an input string, each suffix is represented by a (unique) path from the root of the trie to one of its leaves. We already discussed compacted tries in the previous chapter, now we specialize the description in the context of suffix trees and point out some issues, and their efficient solutions, that arise when the dictionary of indexed strings is composed by suffixes of one single string.

Let us denote the suffix tree built over an input string $T[1, n]$ as $ST_T$ (or just $ST$ when the input is clear from the context) and assume, as done for suffix arrays, that the last character of $T$ is the special symbol \$ which is smaller than any other alphabet character. The suffix tree has the following properties:

1. Each suffix of $T$ is represented by a *unique* path descending from root of $ST$ to one of its leaves. So there are $n$ leaves, one per text suffix, and each leaf is labeled with the starting position in $T$ of its corresponding suffix.
2. Each internal node of $ST$ has at least two outgoing edges. So there are less than $n$ internal nodes and less than $2n - 1$ edges. Every internal node $u$ spells out a text substring, denoted by $s[u]$, which prefixes everyone of the suffixes descending from $u$ in the suffix tree. Typically the value $|s[u]|$ is stored as satellite information of node $u$.
3. The edge labels are non empty substrings of $T$. The labels of the edges spurring from any internal node start with different characters, called *branching characters*. Edges are assumed to be ordered alphabetically according to their branching characters. So every node has at most $\sigma$ outgoing edges.[4]

In Figure 7.8 we show the suffix tree built over our exemplar text $T[1, 12]$ = `mississippi$`. The presence of the special symbol $T[12]$ = \$ ensures that no suffix is a prefix of another suffix of $T$ and thus every pair of suffixes differs in some character. So the paths from the root to the leaves of two different suffixes coincide up to their common longest prefix, which ends up in an internal node of $ST$.

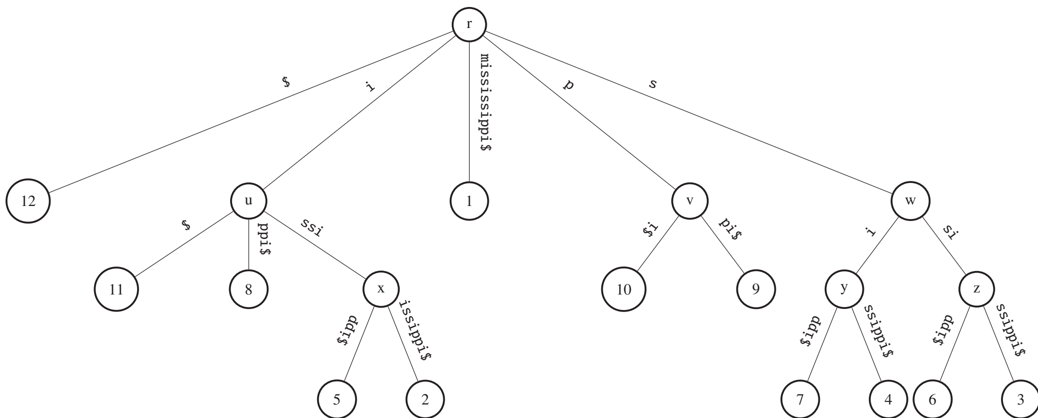



FIGURE 7.8: The suffix tree of the string `mississippi$`

It is evident that we cannot store explicitly the substrings labeling the edges because this would end up in a total space complexity of $\Theta(n^2)$. You can convince yourself by building the suffix tree for the string $T[1, n] = a^{n-1}\$$, and observe that there is an edge for every substring $a^j$; with $j = 1, 2, \ldots, n$. We can circumvent this space explosion by encoding the edge labels with pairs

[4]The special character \$ is included in the alphabet $\Sigma$.

of integers which represent the starting position of the substring and its length. With reference to Figure 7.8 we have that the label of the edge leading to leaf 5, namely the substring $T[9, 12]$ = `ppi$`, can be encoded with the integer pair $\langle 9, 4\rangle$. Other obvious encodings could be possible — say the pair $\langle 9, 12\rangle$—, but we will not detail them here. The key consequence is that every edge label can be encoded in $O(1)$ space, and thus the storage of all edge labels takes $O(n)$ space.

***FACT 7.3*** *The suffix tree of a string $T[1, n]$ consists of n leaves, at most $n-1$ internal nodes and at most $2n-2$ edges. Its space occupancy is $O(n)$, provided that a proper edge-label encoding is adopted.*

As a final notation, we call *locus* of a text substring $t$ the node $v$ whose spelled string is exactly $t$, hence $s[v] = t$. We call *extended locus* of $t'$ the locus of its shortest extension that has defined locus in $ST$. In other words, the path spelling the string $t'$ in $ST$ ends within an edge label, say the label of the edge $(u, v)$. This way $s[u]$ prefixes $t'$ which in turn prefixes $s[v]$. Therefore $v$ is the extended locus of $t'$. Of course if $t'$ has a locus in $ST$ then this coincides with its extended locus. As an example the node $z$ of the suffix tree in Figure 7.8 is the locus of the substring `ssi` and the extended locus of the substring `ss`, which is indeed a prefix of `ssi`.

### 7.3.1 The substring-search problem

The search for a pattern $P[1, p]$ as a substring of the text $T[1, n]$, with the help of the suffix tree $ST$, consists of a tree traversal which starts from its root and proceeds downward as pattern characters are matched against characters labeling the tree edges (see Figure 7.9). Note that, since the first character of the edges outgoing from each traversed node is distinct, the matching of $P$ can follow only one downward path. If the traversal determines a mismatch character, the pattern $P$ does not occur in $T$; otherwise the pattern is fully matched, the extended locus of $P$ is found, and all leaves of $ST$ descending from this node identify all text suffixes which are prefixed by $P$. The text positions associated to these descending leaves are the positions of the *occ* occurrences of the pattern $P$ in $T$. These positions can be retrieved in $O(occ)$ time by visiting the subtree that descends from the extended locus of $P$. In fact this subtree has size $O(occ)$ because it consists of *occ* leaves and each one of its internal nodes has (at least) binary fan-out.

In the running example of Figure 7.9, the pattern $P$ = `na` occurs twice in $T$ and in fact the traversal of $ST$ fully matches $P$ and stops at the node $z$, from which descend two leaves labeled 3 and 5. And indeed the pattern $P$ occurs at positions 3 and 5 of $T$. The cost of pattern searching is $O(pt_\sigma + occ)$ time in the worst case, where $t_\sigma$ is the time to branch out of a node during the tree traversal. This cost depends on the alphabet size $\sigma$ and the kind of data structure used to store the branching characters of the edges spurring from that node. We discussed this issue in the previous Chapter, when solving the prefix-search problem via compacted tries. There we observed that $t_\sigma = O(1)$ if we use a perfect-hash table indexed by the branching characters; it is $t_\sigma = O(\log \sigma)$ if we use a plain array and the branching is implemented by a binary search.

***FACT 7.4*** *The occ occurrences of a pattern $P[1, p]$ in a text $T[1, n]$ can be found in $O(p + occ)$ time and $O(n)$ space by using a suffix tree built on the input text $T$ and in which the branching characters at each node are indexed via a perfect hash table.*

### 7.3.2 Construction from Suffix Arrays and vice versa

It is not difficult to observe that the suffix array $SA_T$ of the text $T$ can be obtained from the suffix tree $ST_T$ by performing an in-order visit: each time a leaf is encountered, the suffix-index stored in this

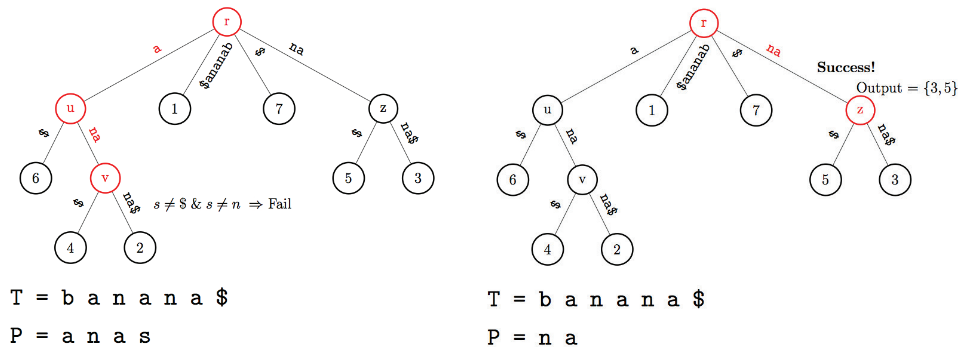



FIGURE 7.9: Two examples of substring searches over the suffix tree built for the text `banana$`. The search for the pattern $P$ = `anas` fails, the search for the pattern $P$ = `na` is successful and stops at node $z$.

leaf is written into the suffix array $SA$; each time an internal node $u$ is encountered, its associated value $|s[u]|$ is written into the array `lcp`.

***FACT 7.5*** *Given the suffix tree of a string $T[1,n]$, we can derive in $O(n)$ time and space the corresponding suffix array SA and the longest-common-prefix array* `lcp`.

Vice versa, we can derive the suffix tree $ST$ from the two arrays $SA$ and `lcp` in $O(n)$ time as follows. The algorithm constructs incrementally $ST$ starting from a tree, say $ST_1$, that contains a root node denoting the empty string and one leaf labeled $SA[1]$. At step $i > 1$, we have inductively constructed the partial suffix tree $ST_{i-1}$ which contains all suffixes in $SA[1, i-1]$, hence the $(i-1)$-smallest suffixes of $T$. During step $i$, the algorithm inserts in the current (partial) suffix tree the $i$-th smallest suffix $SA[i]$. This requires the addition of one leaf labeled $SA[i]$ and, as we will prove next, at most one single other internal node which becomes the father of the inserted leaf. The final tree $ST_n$ will be the suffix tree of the string $T[1,n]$.

The key issue here is to show how to insert the leaf for $SA[i]$ into $ST_{i-1}$ and its father in constant amortized time. This will be enough to ensure a total time complexity of $O(n)$ for the overall construction process. The key difficulty consists in the detection of the node $u$ father of the leaf $SA[i]$. This node $u$ may already exist in $ST_{i-1}$, in this case $SA[i]$ is attached to $u$; otherwise, $u$ must be created by splitting an edge of $ST_{i-1}$. Whether $u$ exists or not is discovered by percolating $ST_{i-1}$ upward, starting from the leaf $SA[i-1]$ and stopping when a node $x$ is reached such that $\mathtt{lcp}[i] \leq |s[x]|$. Recall that $\mathtt{lcp}[i]$ is the number of characters that the text suffix $suff_{SA[i-1]}$ shares with next suffix $suff_{SA[i]}$ in the lexicographic order. The leaves corresponding to these two suffixes are of course consecutive in the in-order visit of $ST$. At this point if $\mathtt{lcp}[i] = |s[x]|$, the node $x$ is the parent of the leaf labeled $SA[i]$, we connect them and the new $ST_i$ is obtained. If instead $\mathtt{lcp}[i] < |s[x]|$, the edge leading to $x$ has to be split by inserting a node that has two children: the left child is $x$ and the right child is the leaf $SA[i]$. This node is associated with the value $\mathtt{lcp}[i]$. The reader can run this algorithm over the string $T[1,12]$ = `mississippi$` and convince herself that the final suffix tree $ST_{12}$ is exactly the one showed in Figure 7.8.

The time complexity of the algorithm derives from an accounting argument which involves the edges traversed by the upward percolation of $ST$. Since the suffix $suff_{SA[i]}$ is lexicographically greater than the suffix $suff_{SA[i-1]}$, the leaf labeled $SA[i]$ lies to the right of the leaf $SA[i-1]$. So every time

we traverse an edge, we either discard it from the next traversals and proceed upward, or we split it and a new leaf is inserted. In particular all edges from $SA[i-1]$ to $x$ are never traversed again because they lie to the right of the newly inserted edge $(u, SA[i])$. The total number of these edges is bounded by the total number of edges in $ST$, which is $O(n)$ from Fact 7.3. The total number of edge-splits equals the number of inserted leaves, which is again $O(n)$.

***FACT 7.6*** *Given the suffix array SA and the longest-common-prefix array* `lcp` *of a string* $T[1,n]$*, we can derive the corresponding suffix tree ST in* $O(n)$ *time and space.*

### 7.3.3 McCreight's algorithm$^{\infty}$

A naïve algorithm for constructing the suffix tree of an input string $T[1,n]$ could start with an empty trie and then iteratively insert text suffixes, one after the other. The algorithm maintains the property by which each intermediate trie is indeed a compacted trie of the suffixes inserted so far. In the worst case, the algorithm costs up to $O(n^2)$ time, take e.g. the highly repetitive string $T[1,n] = a^{n-1}\$$. The reason for this poor behavior is due to the *re-scanning* of parts of the text $T$ that have been already examined during the insertion of previous suffixes. Interestingly enough do exist algorithms that construct the suffix tree directly, and thus without passing through the suffix- and lcp-arrays, and still take $O(n)$ time. Nowadays the space succinctness of suffix arrays and the existence of the Skew algorithm, drive the programmers to build suffix trees passing through suffix arrays (as explained in the previous section). However, if the *average lcp* among the text suffixes is small then the direct construction of the suffix tree may be advantageous both in internal memory and on disk. We refer the interested reader to [2] for a deeper analysis of these issues.

In what follows we present the classic McCreight's algorithm [8], introduced in 1976. It is based on a nice technique that adds some special pointers to the suffix tree that allow to avoid the *rescanning* drawback mentioned before. These special pointers are called *suffix links* and are defined as follows. The suffix link $SL(z)$ connects the node $z$ to the node $z'$ such that $s[z] = as[z']$. So $z'$ spells out a string that is obtained by dropping the first character from $s[z]$. The existence of $z'$ in $ST$ is not at all clear: Of course $s[z']$ is a substring of $T$, given that $s[z]$ is, and thus we can trace a path in $ST$ and find the extended locus of $s[z']$; but nothing seems to ensure that $s[z']$ has indeed a locus in $ST$ and this is $z'$. This property is derived by observing that the existence of $z$ implies the existence of at least 2 suffixes, say $suff_i$ and $suff_j$ that have the node $z$ as their lowest common ancestor in $ST$, and thus $s[z]$ is their longest common prefix. Looking at Figure 7.8, we can take for node $z$ the suffixes $suff_3$ and $suff_6$ (which are actually children of $z$). Now take the two suffixes following those ones, namely $suff_{i+1}$ and $suff_{j+1}$ (i.e. $suff_4$ and $suff_7$ in the figure). They will share $s[z']$ as their longest common prefix, given that we dropped just their first character, and thus will have $z'$ as their lowest common ancestor. In Figure 7.8, $s[z] =$ `ssi`, $s[z'] =$ `si` and the node $z'$ does exist and is indicated with $y$. In conclusion every node $z$ has one suffix link correctly defined; more subtle is to observe that all suffix links form a tree rooted in the root of $ST$.

McCreight's algorithm works in $n$ steps, it starts with the suffix tree $ST_1$ which consists of a root node, denoting the empty string, and one leaf labeled $suff_1 = T[1,n]$ (namely the entire text). In a generic step $i > 1$, the current suffix tree $ST_{i-1}$ is the compacted trie built over all text suffixes $suff_j$ such that $j = 1, 2, \ldots, i-1$. Hence suffixes are inserted in $ST$ from the longest to the shortest one, so at any step $ST_{i-1}$ indexes the $(i-1)$ longest suffixes of $T$.

To ease the description of the algorithm we need to introduce the notation $head_i$ which denotes the longest prefix of suffix $suff_i$ which occurs in $ST_{i-1}$. Given that $ST_{i-1}$ is a partial suffix tree, $head_i$ is the longest common prefix between $suff_i$ and any of its previous suffixes in $T$, namely $suff_j$ with $j = 1, 2, \ldots, i-1$. Given $head_i$ we denote by $h_i$ the (extended) locus of that string in the current suffix tree: actually $h_i$ is the extended locus in $ST_{i-1}$ because $suff_i$ has not yet been inserted, but

$head_i = s[h_i]$ in $ST_i$ and indeed $h_i$ is the parent of the leaf associated to the suffix $suff_i$ after its insertion. As an example, consider the suffix $suff_5$ = `byabz$` in the partial suffix trees of Figure 7.10. We have that this suffix shares only the character `b` with the previous four suffixes of $T$, so $head_5$ = `b` in $ST_4$. Moreover, $head_5$ has extended locus in $ST_4$ (i.e. the leaf 2), but it has locus $h_5 = v$ in $ST_5$.

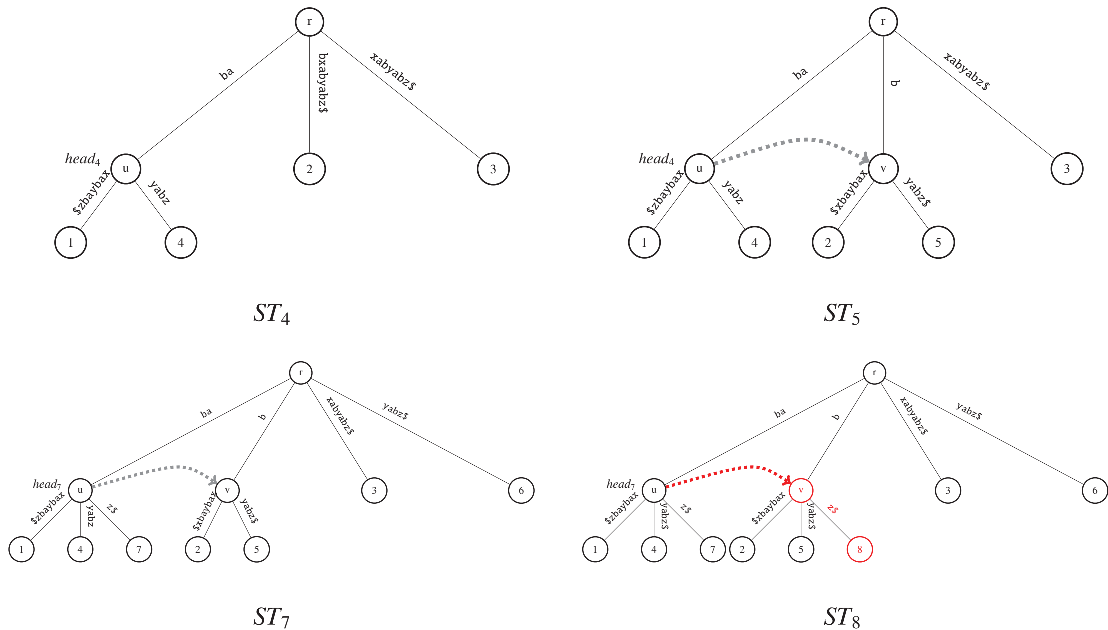



FIGURE 7.10: Several steps of the McCreight's algorithm for the string $T$ =`abxabyabz$`.

Now we are ready to describe the McCreight's algorithm in detail. To produce $ST_i$, we must locate in $ST_{i-1}$ the (extended) locus $h_i$ of $head_i$. If it is an extended locus, then the edge incident in this node is split by inserting an internal node, which corresponds to $h_i$, to which the leaf for $suff_i$ is attached. In the naïve algorithm, $head_i$ and $h_i$ were found tracing a downward path in $ST_{i-1}$ matching $suff_i$ character-by-character. However this induced a quadratic time complexity in the worst case. Instead McCreight's algorithm determines $head_i$ and $h_i$ by using the information inductively available for string $head_{i-1}$, node $h_{i-1}$ (which is the locus of $head_{i-1}$, because of the observation above), and the *suffix links* which are already available in $ST_{i-1}$.

***FACT 7.7*** *In $ST_{i-1}$ the suffix link $SL(u)$ is defined for all nodes $u \neq h_{i-1}$. It may be the case that $SL(h_{i-1})$ is defined too, because that node was already present in $ST_{i-1}$ before the insertion of $suff_{i-1}$.*

**Proof** Since $head_{i-1}$ prefixes $suff_{i-1}$, the second suffix of $head_{i-1}$ starts at position $i$ and thus prefixes the suffix $suff_i$. We denote this second suffix with $head^-_{i-1}$. By definition $head_i$ is the longest prefix shared between $suff_i$ and anyone of the previous text suffixes, so that $|head_i| \geq |head_{i-1}| - 1$ and the string $head^-_{i-1}$ prefixes $head_i$. ∎

McCreight's algorithm starts with $ST_1$ that consists of two nodes: the root and the leaf for $suff_1$. At step 1 we have that $head_1$ is the empty string, $h_1$ is the root, and $SL(root)$ points to the root itself. At a generic step $i > 1$, we know $head_{i-1}$ and $h_{i-1}$, and we wish to determine $head_i$ and $h_i$, in order to insert the leaf for $suff_i$ as a child of $h_i$. These data are found via the following three sub-steps:

1. if $SL(head_{i-1})$ is defined, we set $w = SL(head_{i-1})$ and we go to step 3;
2. Otherwise we need to perform a **rescanning** whose goal is to find/create the locus $w$ of $head^{-}_{i-1}$ and consequently set the suffix link $SL(h_{i-1}) = w$. This is implemented by taking the parent $f$ of $head_{i-1}$, jumping via its suffix link $f' = SL(f)$ (which is defined according to Fact 7.7), and then tracing a downward path from $f'$ starting from the $(|s[f']| + 1)$-th character of $suff_i$. Since we know that $head^{-}_{i-1}$ occurs in $T$ and it prefixes $suff_i$, this downward tracing to find $w$ can be implemented by comparing only the branching characters of the traversed edges with $head^{-}_{i-1}$. If the landing node of this traversal is the locus of $head^{-}_{i-1}$, then this landing node is the searched $w$; otherwise the landing node is the extended locus of $head^{-}_{i-1}$, so we split the last traversed edge and insert the node $w$ such that $s[w] = head^{-}_{i-1}$. In all cases we set $SL(h_{i-1}) = w$;
3. Finally, we locate $head_i$ starting from $w$ and **scanning** the rest of $suff_i$. If the locus of $head_i$ does exist, then we set it to $h_i$; otherwise the scanning of $head_i$ stopped within some edge, and so we split it by inserting $h_i$ as the locus of $head_i$. We conclude the process by installing the leaf for $suff_i$ as a child of $h_i$.

Figure 7.10 shows an example of the advantage induced by suffix links. As step 8 we have the partial suffix tree $ST_7$, $head_7 = \texttt{ab}$, $h_7 = u$, and we need to insert the suffix $suff_8 = \texttt{bz\$}$. Using McCreight's algorithm, we find that $SL(h_7)$ is defined and equal to $v$, so we reach that node following the suffix link (without rescanning $head^{-}_{i-1}$). Subsequently, we scan the rest of $suff_i$, namely `z$`, searching for the locus of $head_i$, but we find that actually $head_i = head^{-}_{i-1}$, so $h_i = v$ and we can attach there the leaf 8.

From the point of view of time complexity, we observe that the rescanning and the scanning steps perform two different types of traversals: the former traverses edges by comparing only the branching characters, since it is rescanning the string $head^{-}_{i-1}$ which is already known from the previous step $i - 1$; the latter traverses edges by comparing their labels in their entirety because it has to determine $head_i$. This last type of traversal always advances in $T$ so the cost of the scanning phase is $O(n)$. The difficulty is to evaluate that the cost of rescanning is $O(n)$ too. The proof comes from an observation on the structure of suffix links and suffix trees: if $SL(u) = v$ then all ancestors of $u$ point to a distinct ancestor of $v$. This comes from Fact 7.7 (all these suffix links do exist), and from the definition of suffix links (which ensures ancestorship). Hence the tree-depth of $v$, say $d[v]$, is larger than $d[u] - 1$ (where $-1$ is due to the dropping of the first character). Therefore, the execution of rescanning can decrease the current depth at most by 2 (i.e., one for reaching the father of $h_{i-1}$, one for crossing $SL(h_{i-1})$. Since the depth of $ST$ is most $n$, the number of edges traversed by rescanning is at most $3n$, and each edge traversal takes $O(1)$ time because only the branching character is matched.

The last issue to be considered regards the cost of branching out of a node during the re-scanning and the scanning steps. Previously we stated that this costs $O(1)$ by using perfect hash-tables built over the branching characters of each internal node of $ST$. In the context of suffix-tree construction the tree is dynamic and thus we should adopt dynamic perfect hash-tables, which is a pretty involved solution. A simpler approach consists of keeping the branching characters and their associated edges within a binary-search tree thus supporting the branching in $O(\log \sigma)$ time. Practically, programmers relax the requirement of worst-case complexity and use either hash tables with chaining, or dictionary data structures for integer values (such as the Van Emde-Boas tree, whose search

complexity is $O(\log\log\sigma)$ time) because characters can be looked at as sequences of bits and hence integers.

***THEOREM 7.6*** *McCreight's algorithm builds the suffix tree of a string $T[1,n]$ in $O(n\log\sigma)$ time and $O(n)$ space.*

This algorithm is inefficient in an external-memory setting because it may elicit one I/O per each tree-edge traversal. We refer the reader to [2] for details on this issue.

## 7.4 Some interesting problems

### 7.4.1 Approximate pattern matching

The problem of approximate pattern matching can be formulated as: *finding all substrings of a text $T[1,n]$ that match a pattern $P[1,p]$ with at most $k$ errors*. In this section we restrict our discussion to the simplest type of errors, the ones called *mismatches* or *substitutions* (see Figure 7.11). This way the text substrings which "$k$-mismatch" the searched pattern $P$ have length $p$ and coincide with the pattern in all but at most $k$ characters. The following figure provides an example by considering two DNA strings formed over the alphabet of four nucleotide bases $\{A,T,G,C\}$. The reason for this kind of strings is that Bio-informatics is the context which spurred interest around the approximate pattern-matching problem.

```
C  C  G  T  A  C  G  A  T  C  A  G  T  A
C  C  G  A  A  C  T
         ⇑        ⇑
```

FIGURE 7.11: An example of matching between $T$ (top) and $P$ (bottom) with $k = 2$ mismatches.

The naïve solution to this problem consists of trying to match $P$ with every possible substring of $T$, having length $p$, counting the mismatches and returning the positions were their number is at most $k$. This would take $O(pn)$ time, independently of $k$. The inefficiency comes from the fact that each pattern-substring comparison starts from the beginning of $P$, thus taking $O(p)$ time. In what follows we describe a sophisticated solution which hinges on an elegant data structure that solves an apparently un-related problem formulated over an array of integers, and called *Range Minimum Query* (shortly, RMQ). This data structure is the backbone of many other algorithmic solutions in problems arising in Data Mining, Information Retrieval, and so on.

The following Algorithm 7.4 solves the $k$-mismatches problem in $O(nk)$ time by making the following basic observation. If $P$ occurs in $T$ with $j \leq k$ mismatches, then we can align the pattern $P$ with a substring of $T$ so that $j$ or $j-1$ substrings coincide and $j$ characters mismatch. Actually equal substrings and mismatches interleave each other. As an example consider again Figure 7.11, the pattern occurs at position 1 in the text $T$ with 2 mismatches, and in fact two substrings of $P$ match their corresponding substrings of $T$. This means that if we could compare pattern and text substrings for equality in constant time, then we could execute the naïve-approach taking $O(nk)$ time, instead of $O(np)$ time. To be operational, this observation can be rephrased as follows: if $T[i,i+\ell] = P[j,j+\ell]$ is one of these matching substrings, then $\ell$ is the longest common prefix between the pattern and the text suffixes starting at the matching positions $i$ and $j$. Algorithm 7.4

deploys this rephrasing to code a solution which takes $O(nk)$ time provided that `lcp`-computations take $O(1)$ time.

**Algorithm 7.4** Approximate-pattern matching based on LCP-computations

```
matches = {}
for i = 1 to n do
    m = 0, j = 1;
    while m ≤ k and j ≤ p do
        ℓ = lcp(T[i, n], P[j, p];
        j = j + ℓ;
        if j ≤ p then
            m = m + 1; j = j + 1;
        end if
    end while
    j = 1;
    if m ≤ k then
        matches = matches ∪ {T[i, i + p − 1)]};
    end if
end for
return matches;
```

If we run the Algorithm 7.4 over the strings showed in Figure **??**, we perform two `lcp`-computations and find that $P$ occurs at text position 1 with 2-mismatches:

- `lcp`$(T[1, 14], P[1, 7])$ = `lcp`(CCGTACGATCAGTA, CCGTACG) = CCG.
- `lcp`$(T[5, 14], P[5, 7])$ = `lcp`(ACGATCAGTA, ACG) = AC.

How do we compute `lcp`$(T[i, n], P[j, p])$ in constant time? We know that suffix trees and suffix arrays have built-in some `lcp`-information, but we similarly recall that these data structures were built on one single string, namely the text $T$. Here we are talking of suffixes of $P$ and $T$ together. Nonetheless we can easily circumvent this difficulty by constructing the suffix array, or the suffix tree, over the string $X = T\#P$, where # is a new character not occurring elsewhere. This way each computation of the form `lcp`$(T[i, n], P[j, p])$ can now be turned into an `lcp`-computation between suffixes of $X$, precisely `lcp`$(T[i, n], P[n + 1 + j, n + 1 + p])$. We are therefore left with showing how these `lcp`-computations can be performed in constant time, whichever is the pair of compared suffixes. This is the topic of the next subsection.

### Lowest Common Ancestor, Range Minimum Query and Cartesian Tree

Let us start from an example, by considering the suffix tree $ST_X$ and the suffix array $SA_X$ built on the string $X = CCGTACGATCAGTA$. This string is not in the form $X = T\#P$ because we wish to stress the fact that the considerations and the algorithmic solutions proposed in this section apply to any string $X$, not necessarily the ones arising from the Approximate Pattern-Matching problem.

The key observation, whose correctness spurs immediately from Figure 7.12, is that there is a strong relation between the `lcp`-problem over $X$'s suffixes and the computation of *lowest common ancestors* (`lca`) in the suffix tree $ST_X$. Consider the problem of finding the longest common prefix between suffixes $X[i, x]$ and $X[j, x]$) where $x = |X|$. It is not difficult to convince yourself that the node $u$ = `lca`$(X[i, x], X[j, x])$ in the suffix tree $ST_X$ spells out their `lcp`, and thus the value $|s[u]|$

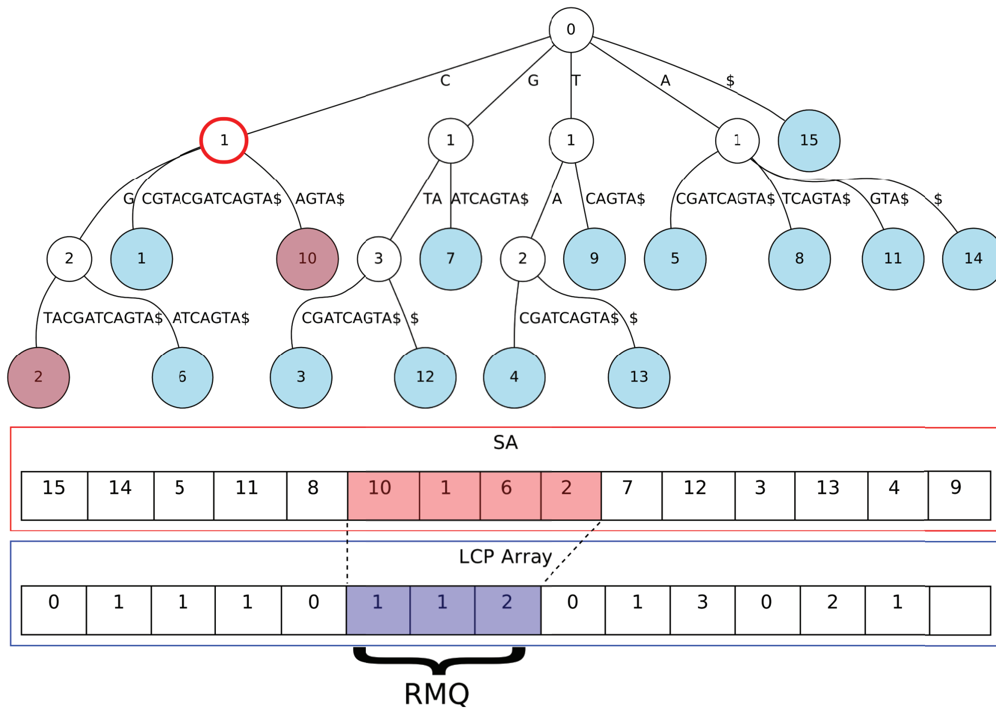



FIGURE 7.12: An example of suffix tree (unordered), suffix array, `lcp`-array for the string $X$ = `CCGTACGATCAGTA`. The figure highlights that the computation of $\texttt{lcp}(X[2,16], X[10,16])$ boils down to finding the depth of the `lca`-node in $ST_X$ between the leaf 2 and the leaf 10, as well as to solve a range minimum query on the sub-array $\texttt{lcp}[6,8]$ since $SA_X[6] = 10$ and $SA_X[9] = 2$.

stored in node $u$ is exactly the `lcp`-value we are searching for. Notice that this property holds independently of the lexicographic sortedness of the edge labels, and thus of the suffix tree leaves.

Equivalently, the same value can be derived by looking at the suffix array $SA_X$. In particular take the lexicographic positions $i_p$ and $j_p$ where those two suffixes occur in $SA_X$, say $SA_X[i_p] = i$ and $SA_X[j_p] = j$ (we are assuming for simplicity that $i_p < j_p$). It is not difficult to convince yourself that the *minimum value* in the sub-array $\texttt{lcp}[i_p, j-1]$[5] is exactly equal to $|s[u]|$ since the values contained in that sub-array are the values stored in the suffix-tree nodes of the subtree that descends from $u$. Actually the order of these values is the one given by the in-order visit of $u$'s descendants. Anyway, this order is not important for our computation which actually takes the smallest value, because it is interested in the shallowest node (namely the root $u$) of that subtree.

Figure 7.12 provides a running example which clearly shows these two strong properties, which actually do not depend on the order of the children of suffix-tree nodes. As a result, we have two approaches to compute `lcp`in constant time, either through `lca`-computations over $ST_X$ or through `RMQ`-computations over $\texttt{lcp}_X$. For the sake of presentation we introduce an elegant solution for the latter, which actually induces in turn an elegant solution for the former, given that their are strongly related.

[5]Recall that $\texttt{lcp}[q]$ stores the length of the longest common prefix between suffix $SA[i]$ and its next suffix $SA[i+1]$.

In general terms the RMQ problem can be stated as follows:

**The range-minimum-query problem.** *Given an array $A[1, n]$ of elements drawn from an ordered universe, build a data structure $\mathtt{RMQ}_A$ that is able to compute efficiently the position of a smallest element in $A[i, j]$, for any given queried range $(i, j)$. We say "a position" because the array may contain many minimum elements.*

We underline that this problem asks for the *position* of a minimum element in the queried subarray, rather than its value. This is more general because the value of the minimum can be obviously retrieved from its position in $A$ by accessing this array, which is available.

In this lecture we aim for constant-time queries. The simplest solution achieves this goal via a table that stores the index of a minimum entry for each possible range $(i, j)$. Such table requires $O(n^2)$ space and $O(n^2)$ time to be built. A better solution hinges on the following observation: any range $(i, j)$ can be decomposed into two (possibly overlapping) ranges whose size is a power of two, namely $(i, i + 2^L)$ and $(j - 2^L, j)$ where $L = \lfloor \log(j - i + 1) \rfloor$. This allows us to *sparsify* the previous quadratic-sized table by storing only ranges whose size is a power of two. This way, for each position $i$ we store the answers to the queries $\mathtt{RMQ}_A(i, i + 2^L)$, thus occupying a total space of $O(n \log n)$ without impairing the time complexity of the query which is still constant and corresponds to return $\mathtt{RMQ}_A(i, j) = \mathtt{argmin}_{i,j}\{\mathtt{RMQ}_A(i, i + 2^L), \mathtt{RMQ}_A(j - 2^L, j)\}$.

In order to get the optimal $O(n)$ space occupancy, we need to dig into the structure of the RMQ-problem and make a twofold reduction which goes back-and-forth from RMQ-computations to lca-computations: namely, we reduce (1) the RMQ-computation over the lcp-array to an lca-computation over Cartesian Trees (that we define next); we then reduce (2) the lca-computation over Cartesian Trees to an RMQ-computation over a binary array. This last problem will then be solved in $O(n)$ space and constant query time. Clearly reduction (2) can be applied to any tree, and thus can be applied to Suffix Trees in order to solve lca-queries over them.

**First reduction step: from RMQ to lca.** We transform the $\mathtt{RMQ}_A$-problem "back" into an lca-problem over a special tree which is known as *Cartesian Tree* and is built over the entries of the array $A[1, n]$. The *Cartesian Tree* $C_A$ is a binary tree of $n$ nodes, each labeled with one of $A$'s entries (i.e. value and position in $A$). The labeling is defined recursively as follows: the root of $C_A$ is labeled by the minimum entry in $A[1, n]$, say this is $\langle A[m], m \rangle$. Then the left subtree of the root is recursively defined as the Cartesian Tree of the subarray $A[1, m - 1]$, and the right subtree is recursively defined as the Cartesian Tree of the subarray $A[m + 1, n]$. Tree $C_A$ can be constructed in $O(n)$ time as follows: Suppose that we have already built the tree $C_i$ for the array $A[1, i]$, then we insert the element $A[i + 1]$ in two steps (see Figure 7.13):

1. we climb the rightmost path of $C_i$ and determine the first node $u$ whose associated entry $A[j]$ is smaller than $A[i + 1]$;
2. we make the node corresponding to $A[i + 1]$ the right son of $u$, and turn the previous right subtree of $u$ into the left subtree of $A[i + 1]$.

The following Figure 7.14 shows the Cartesian tree built on the lcp-array depicted in Figure **??**. Given the construction process, we can state that ranges in the lcp-array correspond to subtrees of the Cartesian tree. Therefore computing $\mathtt{RMQ}_A(i, j)$ boils down to compute an lca-query between the nodes of $C_A$ associated to the entries $i$ and $j$. Differently of what occurred for lca-queries on $ST_X$, where the arguments were leaves of that suffix tree, the queried nodes in the Cartesian Tree may be internal nodes, and actually it might occur that one node is ancestor of the other node. For example, executing $\mathtt{RMQ}_{\mathtt{lcp}}(6, 8)$ equals to executing $\mathtt{lca}(6, 8)$ over the Cartesian Tree $C_{\mathtt{lcp}}$ of Figure 7.14. The result of this query is the node $\langle \mathtt{lcp}[7], 7 \rangle = \langle 1, 7 \rangle$. Notice that we have another minimum value in $\mathtt{lcp}[6, 8]$ at $\mathtt{lcp}[6] = 1$; the answer provided by the lca is one of the existing minima in the queried-range.

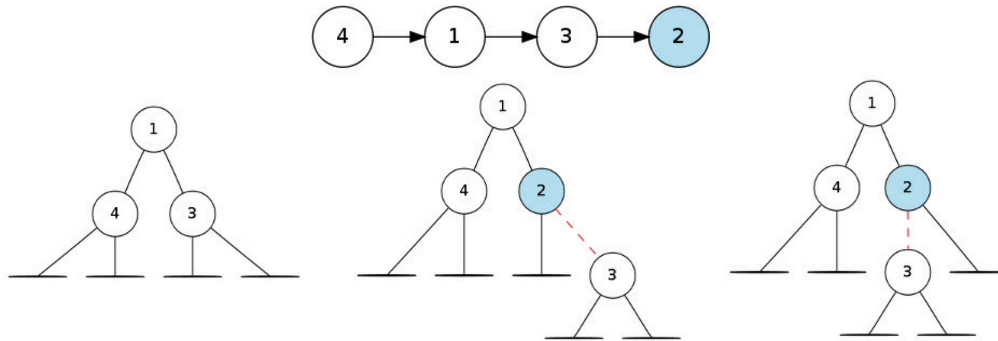



FIGURE 7.13: Example of Cartesian Tree built over the array $A[1, 3] = \{1, 4, 3\}$, and the insertion of the value 2. Observe that nodes of $C_A$ store only $A$'s values, their positions in $A$ are dropped to ease the presentation.

**Second reduction step: from `lca` to `RMQ`.** We transform the `lca`-problem over the Cartesian Tree $C_{\texttt{lcp}}$ "back" into an `RMQ`-problem over a special binary array $\Delta[1, 2e]$, where $e$ is the number of edges in the Cartesian Tree (of course $e = O(n)$). It seems strange this "circular" sequence of reductions that now has turned us back into an `RMQ`-problem. But the current `RMQ`-problem, unlike the original one, is formulated on a binary array and thus admits an optimal solution in $O(n)$ space.

To build the binary array $\Delta[1, 2e]$ we need first to build the array $D[1, 2e]$ which is obtained as follows. Take the *Euler Tour* of Cartesian Tree $C_A$, visiting the tree in pre-order and writing down each node everytime the visit passes through it. A node can be visited multiple times, precisely it is visited/written as many times as its number of incident edges.

Given the Euler Tour of the Cartesian Tree $C_A$, we build the array $D[1, 2e]$ which stores the depths of the visited nodes in the order stated by the Euler Tour (see Figure 7.14). Given $D$ and the way the Euler Tour is built, we can conclude that query $\texttt{lca}(i, j)$ in $C_A$ boils down to compute the node of minimum depth in the sub-array $D[i', j']$ where $i'$ (resp. $j'$) is the position of the first (resp. last) occurrence of the node $i$ (resp. $j$) in the Euler Tour. In fact, the range $D[i', j']$ corresponds to the part of the Euler Tour that starts at node $i$ and ends at node $j$. The node of minimum depth encountered in this Euler sub-Tour is properly the $\texttt{lca}(i, j)$.

So we reduced an `lca`-query over the Cartesian Tree into an `RMQ`-query over node depths. In our running example on Figure 7.14 this reduction transforms the query $\texttt{lca}(6, 8)$ into a query $\texttt{RMQ}_D(11, 13)$, which is highlighted by a red rectangle. Turning nodes into ranges can be done in constant time by simply storing two integers per node of the Cartesian Tree, denoting their first/last occurrence in the Euler Tour, thus taking $O(n)$ space.

We are again "back" to an `RMQ`-query over an integer array. But the current array $D$ is special because its consecutive entries differ by 1 given that they refer to the depths of consecutive nodes in an Euler Tour. And in fact, two consecutive nodes in the Euler Tour are connected by an edge and thus one node is the parent of the other, and hence their depths differ by one unit. The net result of this property is that we can solve the `RMQ`-problem over $D[1, 2e]$ in $O(n)$ space and $O(1)$ time as follows. (Recall that $e = O(n)$.) Solution is based on two data structures which are detailed next.

First, we split the array $D$ into $\frac{2e}{d}$ subarrays $D_k$ of size $d = \frac{1}{2} \log e$ each. Next, we find the minimum element in each subarray $D_k$, and store its position at the entry $M[k]$ of a new array whose size is therefore $\frac{2e}{d}$. We finally build on the array $M$ the sparse-table solution indicated above which takes superlinear space (in the size of $M$) and solves `RMQ`-queries in constant time. The key point here is that $M$'s size is sublinear in $e$, and thus in $n$, so that the overall space taken by array $M$ and its sparse-table is $O((\frac{e}{\log e}) * \log \frac{e}{\log e}) = O(e) = O(n)$.

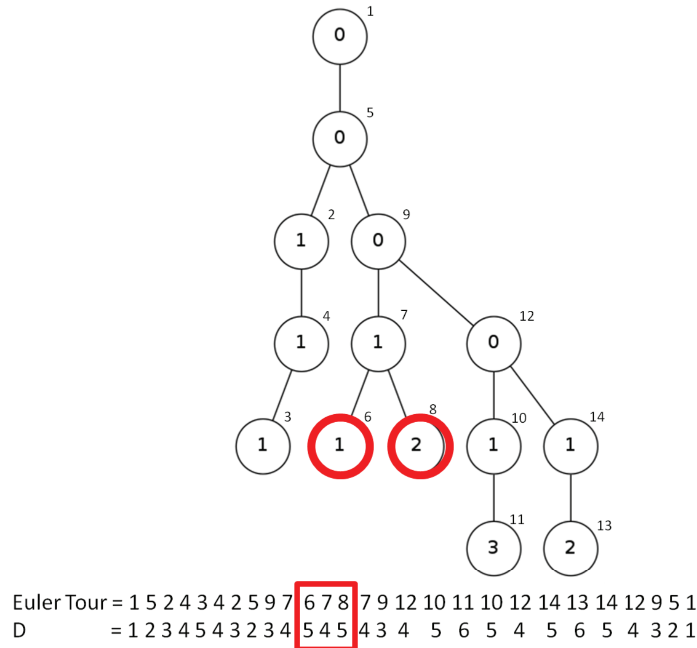



FIGURE 7.14: Cartesian tree built on the `lcp`-array of Figure 7.12. On the bottom part are reported the Euler Tour of the Cartesian Tree and the array $D$ of the depths of the nodes according to the Euler-Tour order.

The second data structure is built to efficiently answer RMQ-queries in which $i$ and $j$ are in the same block $D_k$. It is clear that we cannot tabulate all answers to all such possible pairs of indexes because this would end up in $\Theta(n^2)$ space occupancy. So the solution we describe here spurs from two simple, deep observations whose proof is immediate and left to the reader:

**Binary entries:** Every block $D_k$ can be transformed into a pair that consists of its first element $D_k[1]$ and a binary array $\Delta_k[i] = D_k[i] - D_k[i-1]$ for $i = 2, \ldots, d$. Entries of $\Delta_k$ are either $-1$ or $+1$ because of the unit difference between adjacent entries of $D$.

**Minimum location:** The position of the minimum value in $D_k$ depends only on the content of the binary sequence $\Delta_k$ and does not depend on the starting value $D_k[1]$.

Nicely, the possible configurations that every block $D_k$ can assume are infinite, given that infinite is the number of ways we can instantiate the input array $A$ on which we want to issue the RMQ-queries; but the possible configurations of the image $\Delta_k$ is finite and equal to $2^d$. This suggests to apply the so called *Four Russians trick* to the binary arrays by tabulating all possible binary sequences $\Delta_k$ and, for each of them, storing the position of the minimum value. Since the blocks $\Delta_k$ have length $d = \frac{\log e}{2}$, the total number of possible binary sequences is $2^d = O(2^{\frac{\log e}{2}}) = O(\sqrt{e}) = O(\sqrt{n})$. Moreover since both query-indexes $i$ and $j$ can take at most $d = \frac{\log e}{2}$ possible values, being internal in a block $D_k$, we can have at most $(\frac{\log e}{2})^2$ queries of this third type. Consequently, we build a lookup table $T[i_o, j_o, \Delta_k]$ that is indexed by the possible query-offsets $i_o$ and $j_o$ within the block $D_k$ and its binary configuration $\Delta_k$. Table $T$ stores at that entry the position of the minimum value in $D_k$. We also assume that, for each $k$, we have stored $\Delta_k$ so that the binary representation $\Delta_k$ of $D_k$ can be retrieved in constant time. Each of these indexing parameters takes $O(\log e) = O(\log n)$ bits

of space, hence one memory word, and thus can be managed in $O(1)$ time and space. In summary, the whole table $T$ consists of $O(\sqrt{n}(\log n)^2) = o(n)$ entries. The time needed to build $T$ is $O(n)$. The power of transforming $D_k$ into $\Delta_k$ is evident now, every entry of $T[i_o, j_o, \Delta_k]$ is actually encoding the answer for an infinite number of blocks $D_k$, namely the ones that can be turned to the same binary configuration $\Delta_k$.

At this point we are ready to design an algorithm that, using the three data structures illustrated above, answers a query $\mathtt{RMQ}_D(i, j)$ in constant time. If $i, j$ are inside the same block $D_k$ then the answer is retrieved in two steps: first we compute the offsets $i_o$ and $j_o$ with respect to the beginning of $D_k$ and determine the binary configuration $\Delta_k$ from $k$; then we use this triple to access the proper entry of $T$. Otherwise the range $(i, j)$ spans at least two blocks and can thus be decomposed in three parts: a suffix of some block $D_{i'}$, a consecutive sequence of blocks $D_{i'+1} \cdots D_{j'-1}$, and finally the prefix of block $D_{j'}$. The minimum for the suffix of $D_{i'}$ and the prefix of $D_{j'}$ can be retrieved from $T$, given that these ranges are inside two blocks. The minimum of the range spanned by $D_{i'+1} \cdots D_{j'-1}$ is stored in $M$. All this information can be accessed in constant time and the final minimum-position can be retrieved by comparing these three minimum values, in constant time too.

***THEOREM 7.7*** *Range-minimum queries over an array $A[1, n]$ of elements drawn from an ordered universe can be answered in constant time using a data structure that occupies $O(n)$ space.*

Given the stream of reductions we illustrated above, we can conclude that Theorem 7.7 applies also to computing `lca` in generic trees: it is enough to take the input tree in place of the Cartesian Tree.

***THEOREM 7.8*** *Lowest-common-ancestor queries over a generic tree of size n can be answered in constant time using a data structure that occupies $O(n)$ space.*

### 7.4.2 Text Compression

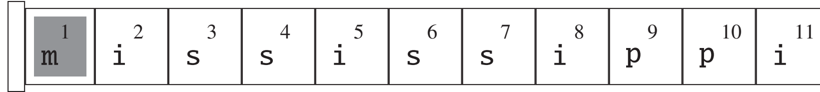
Data compression will be the topic of one of the following chapters; nonetheless in this section we address the problem of compressing a text via the simple algorithm which is at the core of the well known `gzip` compressor, named *LZ77* from the initials of its inventors (Ronny Lempel and Jacob Ziv) and from the year of its publication (1977). We will show that there exists an optimal implementation of the LZ77-algorithm taking $O(n)$ time and using suffix trees.

Given a text string $T[1, n]$, the algorithm LZ77 produces a parsing of $T$ into substrings that are defined as follows. Assume that it has already parsed the prefix $T[1, i-1]$ (at the beginning this prefix is empty), the remaining text suffix $T[i, n]$ is then decomposed in three parts: the longest substring $T[i, i+\ell-1]$ which starts at $i$ and repeats before in the text $T$, the next character $T[i+\ell]$, and the remaining suffix $T[i+\ell+1, n]$. Given this decomposition, the next substring to add to the parsing of $T$ is $T[i, i+\ell]$, and thus corresponds to the shortest string that is *new* in $T[1, i-1]$. Parsing then continues onto the remaining suffix $T[i+\ell+1, n]$, if any.

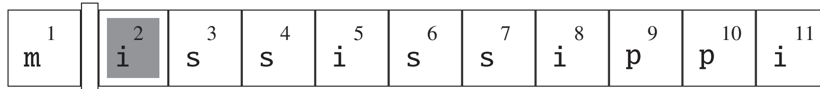
Compression is obtained by succinctly encoding the triple of integers $\langle d, \ell, T[i+\ell]\rangle$, where $d$ is the distance (in characters) from $i$ to the previous copy of $T[i, i+\ell-1]$; $\ell$ is the length of the copied string; $T[i+\ell]$ is the appended character. By saying "previous copy" of $T[i, i+\ell-1]$, we mean that its copy starts before position $i$ but it might extend after this position, hence it could be $d < \ell$; furthermore, the previous copy can be any previous occurrence of $T[i, i+\ell-1]$, although space-efficiency issues suggest us to take the closest copy (and thus the smallest $d$). Finally we observe that the reason for adding the character $T[i+\ell]$ to the emitted triple is that this character behaves like an *escape*-mechanism; in fact it is useful when no-copy is possible and thus $\ell = 0$ (this occurs

when a new character is met in $T$).[6]

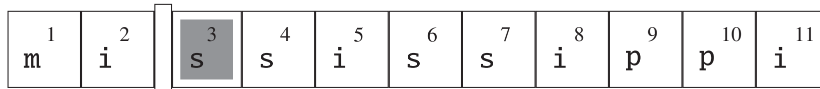
Before digging into an efficient implementation of the LZ77-algorithm let us consider our example string $T$ = `mississippi`. Its LZ77-parsing is computed as follows:



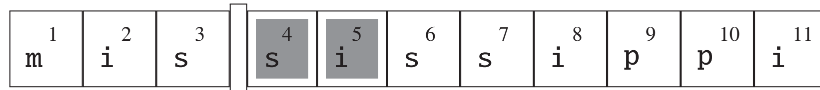

**Output:** $< 0, 0, m >$



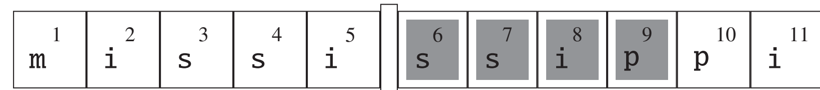

**Output:** $< 0, 0, i >$



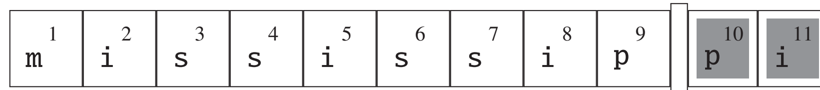

**Output:** $< 0, 0, s >$




**Output:** $< 1, 1, i >$




**Output:** $< 3, 3, p >$




**Output:** $< 1, 1, i >$

We can compute the LZ77-parsing in $O(n)$ time via an elegant algorithm that deploys the suffix tree $ST_T$. The difficulty is to find repeated substrings in a fast way. Let $\pi_i$ be the longest substring that occurs at position $i$ and repeats before in the text $T$ at distance $d$. Given our notation above we have that $\ell = |\pi_i|$. Of course $\pi_i$ is a prefix of $\mathit{suff}_i$ and a prefix of $\mathit{suff}_{i-d}$; actually, it is the longest common prefix of these two suffixes, and by maximality, there is no other previous suffix $\mathit{suff}_j$ (with $j < i$) that shares a longer prefix with $\mathit{suff}_i$. By properties of suffix trees, the lowest-common-ancestor of leaves $i$ and $j$ spells out $\pi_i$. However we cannot compute `lca`$(i, j)$ by issuing a query to the data structure of Theorem 7.8 because we do not know $j$, which is exactly the information we wish to compute. Similarly we cannot trace a downward path from the root of $ST_T$ trying to match

[6]We are not going to discuss the integer-encoding issue, since it will be the topic of a next chapter, we just mention here that efficiency is obtained in `gzip` by taking the rightmost copy and by encoding the values $d$ and $\ell$ via a Huffman coder.

$suff_i$ because all suffixes of $T$ are indexed in the suffix tree and thus we could detect a longer copy which follows position $i$.

To circumvent these problems we preprocess $ST_T$ via a post-order visit that computes for every node $u$ its minimum leaf $\min(u)$. Clearly $\min(u)$ is the leftmost position from which we can copy the substring $s[u]$. Given this information we can determine easily $\pi_i$, just trace a downward path from the root of $ST_T$ scanning $suff_i$ and stopping as soon as the reached node $v$ is such that $\min(v) \geq i$. At this point we take $u$ as the parent of $v$ and set $\pi_i = s[u]$, and $d = i - \min(u)$. Clearly, the chosen copy of $\pi_i$ is the farthest one and not the closest one: this does not impact in the number of phrases in which $T$ is parsed by LZ77, but possibly influences the magnitude of these distances and thus their succinct encoding. Devising an LZ77-parser that efficiently determines the closest copy of each $\pi_i$ is non trivial and needs much more sophisticated data structures.

Take $T$ = `mississippi` as the string to be parsed (see above) and consider its suffix tree $ST_T$ in Figure *fig:mississippi*. Assume that the parsing is at the the suffix $suff_2$ = `ississippi`. Its tracing down $ST_T$ stops immediately at the root of the suffix tree because the node to be visited next would be $u$, for which $\min(u) = 2$ which is not smaller than the current suffix position. Then consider the parsing at suffix $suff_6$ = `ssippi`. We trace down $ST_T$ and stop at node $z$, $s[z]$ = `ssi`, for which $\min(z) = 3 < 6$. The emitted triple is correctly $< 3, 3, p >$.

The time complexity of this implementation of the LZ77-algorithm is $O(n)$ because the traversal of the suffix tree advances over the string $T$, and this may occur only $n$ times. Branching out of suffix-tree nodes can be implemented in $O(1)$ time via perfect hash tables, as observed for the substring-search problem. The construction of the suffix tree costs $O(n)$ time, by using one of the algorithms we described in the previous sections. The computation of the values $\min(u)$, over all nodes $u$, takes $O(n)$ time via a post-order visit of $ST_T$.

***THEOREM 7.9*** *The LZ77-parsing of a string $T[1, n]$ can be computed in $O(n)$ time and space. Each substring of the parsing is copied from its farthest previous occurrence.*

### 7.4.3 Text Mining

In this section we briefly survey two examples of uses of suffix arrays and `lcp`-arrays in the solution of sophisticated text mining problems.

Let us consider the following question: *Check whether there exists a substring of $T[1, n]$ that repeats at least twice and has length L.* Solving this problem in a brute-force way would mean to take every text substring of length $L$, and count its number of occurrences in $T$. These substrings are $\Theta(n)$, searching each of them takes $O(L)$ time, hence the overall time complexity of this trivial algorithm would be $O(nL)$. A smarter and faster, actually optimal, solution comes from the use of the lcp-array $\mathtt{lcp}_T$, built on the input text $T$. Recall that suffixes in $SA_T$ are lexicographically ordered, so the longest common prefix shared by suffix $SA_T[i]$ is with its adjacent suffixes, namely either with suffix $SA_T[i-1]$ or with suffix $SA_T[i+1]$. The length of these lcps is stored in the entries $\mathtt{lcp}_T[i-1, i]$. Now, if the repeated substring of length $L$ does exist, and it occurs e.g. at text positions $x$ and $y$, then we have $\mathtt{lcp}(T[x, n], T[y, n]) \geq L$. These two suffixes not necessarily are contiguous in $SA_T$ (this is the case when the substring occurs more than twice), nonetheless all suffixes occurring among them in $SA$ will surely share a prefix of length $L$, because of their lexicographic order. Hence, if suffix $T[x, n]$ occurs at position $q$ of the suffix array, i.e. $SA_T[q] = x$, then we have that either $\mathtt{lcp}_T[q-1] \geq L$ or $\mathtt{lcp}_T[q] \geq L$, depending on the fact that $T[y, n] < T[x, n]$ or vice versa, respectively. Hence we can solve the question stated above by scanning $\mathtt{lcp}_T$ and searching for an entry $\geq L$. This takes $O(n)$ optimal time.

Let us now ask a more sophisticated question: *Check whether there exists a text substring that repeats at least C times and has length L.* This is the typical query in a text mining scenario,

where we are interested not just in a repetitive event but in an event occurring with some *statistical evidence*. We can again solve this problem by trying all possible substrings and counting their occurrences. Again, a faster solution comes from the use of the array $\texttt{lcp}_T$. Following the argument provided in the solution of the previous question we note that, if a substring of length $L$ occurs (at least) $C$ times, then it does exist (at least) $C$ text suffixes that share (at least) $L$ characters. So there do exist a sub-array in $\texttt{lcp}_T$ of length $\geq C - 1$ that consists of entries $\geq L$. A simple scan of this array can determine the occurrence of this event and thus provide an answer to the above question in $O(n)$ time.

## 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. String search in external memory: algorithms and data structures. *Handbook of Computational Molecular Biology*, edited by Srinivas Aluru. Chapman & Hall/CRC Computer and Information Science Series, chapter 35, Dicembre 2005.

[3] Paolo Ferragina and Travis Gagie and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica: Special issue on selected papers of LATIN 2010*, 63(3): 707-730, 2012.

[4] 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.

[5] 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.

[6] 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.

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

[8] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2): 262-272, 1976.