Skriptum VL *Text Indexing*

Sommersemester 2012

Johannes Fischer (KIT)

Disclaimer

Students attending my lectures are often astonished that I present the material in a much livelier form than in this script. The reason for this is the following:

This is a *script*, not a *text book*.

It is meant to *accompany* the lecture, not to *replace* it! We do examples on all concepts, definitions, theorems and algorithms in the lecture, but usually not not this script. In this sense, it is **not a good idea** to study the subject soley by reading this script.

1 Recommended Reading

- D. Gusfield: *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
- M. Crochemore, W. Rytter, *Jewels of Stringology.* World Scientific, 2002.
- M. Crochemore, C. Hancart, T. Lecroq, *Algorithms on Strings.* Cambridge UP, 2007.
- D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching.* Springer, 2008.

2 Suffix Trees and Arrays

2.1 Suffix Trees

In this section we will introduce suffix trees, which, among many other things, can be used to solve the string matching task (find pattern P of length m in a text T of length n in $O(n+m)$ time). We already know that other methods (Boyer-Moore, e.g.) solve this task in the same time. So why do we need suffix trees?

The advantage of suffix trees over the other string-matching algorithms (Boyer-Moore, KMP, etc.) is that suffix trees are an *index* of the text. So, if T is *static* and there are several patterns to be matched against T, the $O(n)$-task for building the index needs to be done only once, and subsequent matching-tasks can be done in $O(m)$ time. If $m \ll n$, this is a clear advantage over the other algorithms.

Definition 1. *Given a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings over Σ. A* compact trie *on S is a rooted tree $S = (V, E)$ with edge labels from Σ^+ that fulfills the following two constraints:*

- *$\forall v \in V$: all outgoing edges from v start with a different $a \in \Sigma$.*
- *Apart from the root, all nodes have out-degree $\neq 1$.*

- *For all $S_i \in \mathcal{S}$ there is a leaf ℓ such that S_i is a prefix of the concatenation of the labels on the root-to-ℓ path.*

- *For all leaves $\ell \in V$ there is a string $S_i \in \mathcal{S}$ such the root-to-ℓ path spells out exactly S_i.*

We are now ready to define suffix trees. Throughout this section, let $T = t_1 t_2 \dots t_n$ be a text over an alphabet Σ of size $|\Sigma| =: \sigma$. We use the notation $T_{i \dots j}$ as an abbreviation of $t_i t_{i+1} \dots t_j$, the substring of T ranging from i to j.

Definition 2. *The i'th* suffix *of T is the substring $T_{i \dots n}$ and is denoted by T^i.*

Definition 3. *The* suffix tree *of T is a compact trie over all suffixes $\{T^1, T^2, \dots, T^n\}$.*

The following definitions make it easier to argue about suffix trees and compact tries in general:

Definition 4. *Let $S = (V, E)$ be a compact trie.*

- *For $v \in V$, $\overline{v}$ denotes the concatenation of all path labels from the root of S to v.*

- *$|\overline{v}|$ is called the* string-depth *of v and is denoted by $d(v)$.*

- *S is said to* display *$\alpha \in \Sigma^*$ iff $\exists v \in V, \beta \in \Sigma^* : \overline{v} = \alpha\beta$.*

- *If $\overline{v} = \alpha$ for $v \in V, \alpha \in \Sigma^*$, we also write $\overline{\alpha}$ to denote v.*

- *words(S) denotes all strings in Σ^* that are displayed by S: words$(S) = \{\alpha \in \Sigma^* : S$ displays $\alpha\}$*

[NB. With these new definitions, an alternative definition of suffix trees would be: "The *suffix tree* of T is a compact trie that displays exactly the subwords of T."]

For several reasons, we shall find it useful that each suffix ends in a leaf of S. This can be accomplished by adding a new character $\$ \notin \Sigma$ to the end of T, and build the suffix tree over $T\$$.

From now on, we assume that T terminates with a \$, and we define \$ to be lexicographically smaller than all other characters in Σ: $\$ < a$ for all $a \in \Sigma$. This gives a one-to-one correspondence between T's suffixes and the leaves of S, which implies that we can *label the leaves* with a function l by the start index of the suffix they represent: $l(v) = i \iff \overline{v} = T^i$.

Remark: The outgoing edges at internal nodes v of the suffix tree can be implemented in two fundamentally different ways:

1. as arrays of size σ

2. as arrays of size s_v, where s_v denotes the number of v's children

Approach (1) has the advantage that the outgoing edge whose edge label starts with $\alpha \in \Sigma$ can be located in $O(1)$ time, but the complete suffix tree uses space $O(n\sigma)$, which can be as bad as $O(n^2)$. Hence, we assume that approach (2) is used, which implies that locating the correct outgoing edge takes $O(\log \sigma)$ time (using binary search). Note that the space consumption of approach (2) is always $O(n)$, independent of σ.

2.2 Searching in Suffix Trees

Let P be a pattern of length m. Throughout the whole lecture, we will be concerned with the two following problems:

Problem 1. Counting*: Return the number of matches of P in T. Formally, return the* size *of $O_P = \{i \in [1,n] : T_{i\ldots i+m-1} = P\}$*

Problem 2. Reporting*: Return all occurrences of P in T, i. e., return the set O_P.*

With suffix trees, the *counting-problem* can be solved in $O(m \log \sigma)$ time: traverse the tree from the root downwards, in each step locating the correct outgoing edge, until P has been scanned completely. More formally, suppose that $P_{1\ldots i-1}$ have already been parsed for some $1 \leq i < m$, and our position in the suffix tree S is at node v ($\overline{v} = P_{1\ldots i-1}$). We then find v's outgoing edge e whose label starts with P_i. This takes $O(\log \sigma)$ time. We then compare the label of e character-by-character with $P_{i\ldots m}$, until we have read all of P ($i = m$), or until we have reached position $j \geq i$ for which $\overline{P_{1\ldots j}}$ is a node v' in S, in which case we continue the procedure at v'. This takes a total of $O(m \log \sigma)$ time. Suppose the search procedure has brought us successfully to a node v, or to the incoming edge of node v. We then output the size of S_v, the subtree of S rooted at v. This can be done in constant time, assuming that we have labeled all nodes in S with their subtree sizes. This answers the *counting query*. For the *reporting query*, we output the labels of all leaves in S_v (recall that the leaves are labeled with text positions).

Theorem 1. *The suffix tree allows to answer counting queries in $O(m \log \sigma)$ time, and reporting queries in $O(m \log \sigma + |O_P|)$ time.*

An important *implementation detail* is that the edge labels in a suffix tree are represented by a pair (i,j), $1 \leq i \leq j \leq n$, such that $T_{i\ldots j}$ is equal to the corresponding edge label. This ensures that an edge label uses only a constant amount of memory.

From this implementation detail and the fact that S contains exactly n leaves and hence less than n internal nodes, we can formulate the following theorem:

Theorem 2. *A suffix tree occupies $O(n)$ space in memory.*

2.3 Suffix- and LCP-Arrays

We will now introduce two arrays that are closely related to the suffix tree, the *suffix array A* and the *LCP-array H*.

Definition 5. *The* suffix array *A of T is a permutation of $\{1, 2, \ldots, n\}$ such that $A[i]$ is the i-th smallest suffix in lexicographic order: $T^{A[i-1]} < T^{A[i]}$ for all $1 < i \leq n$.*

The following observation relates the suffix array A with the suffix tree S.

Observation 1. *If we do a lexicographically-driven depth-first search through S (visit the children in lexicographic order of the first character of their corresponding edge-label), then the leaf-labels seen in this order give the suffix-array A.*

The second array H builds on the suffix array:

Definition 6. *The* LCP-array H *of* T *is defined such that* $H[1] = 0$*, and for all* $i > 1$*,* $H[i]$ *holds the length of the longest common prefix of* $T^{A[i]}$ *and* $T^{A[i-1]}$*.*

To relate the LCP-array H with the suffix tree S, we need to define the concept of lowest common ancestors:

Definition 7. *Given a tree* $S = (V, E)$ *and two nodes* $v, w \in V$*, the* lowest common ancestor *of* v *and* w *is the deepest node in* S *that is an ancestor of both* v *and* w*. This node is denoted by* $\text{LCA}(v, w)$*.*

Observation 2. *The string-depth of the lowest common ancestor of the leaves labeled* $A[i]$ *and* $A[i-1]$ *is given by the corresponding entry* $H[i]$ *of the LCP-array, in symbols:* $\forall i > 1 : H[i] = d(\text{LCA}(\overline{T^{A[i]}}, \overline{T^{A[i-1]}}))$*.*

2.4 Searching in Suffix Arrays

We can use a *plain suffix array* A to search for a pattern P, using the ideas of *binary search*, since the suffixes in A are *sorted* lexicographically and hence the occurrences of P in T form an *interval* in A. The algorithm below performs two binary searches. The first search locates the starting position s of P's interval in A, and the second search determines the end position r. A *counting query* returns $r - s + 1$, and a *reporting* query returns the numbers $A[s], A[s+1], \ldots, A[r]$.

```
Algorithm 1: function SAsearch($P_{1\ldots m}$)
  $l \leftarrow 1$; $r \leftarrow n+1$;
  while $l < r$ do
      $q \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
      if $P >_{\text{lex}} T_{A[q]\ldots\min\{A[q]+m-1,n\}}$ then
          $l \leftarrow q+1$;
      else
          $r \leftarrow q$;
      end
  end
  $s \leftarrow l$; $l--$; $r \leftarrow n$;
  while $l < r$ do
      $q \leftarrow \lceil \frac{l+r}{2} \rceil$;
      if $P =_{\text{lex}} T_{A[q]\ldots\min\{A[q]+m-1,n\}}$ then
          $l \leftarrow q$;
      else
          $r \leftarrow q-1$;
      end
  end
  return $[s, r]$;
```

Note that both while-loops in Alg. 1 make sure that either l is increased or r is decreased, so they are both guaranteed to terminate. In fact, in the first while-loop, r always points one position *behind* the current search interval, and r is *decreased* in case of equality (when $P = T_{A[q]\ldots\min\{A[q]+m-1,n\}}$).

This makes sure that the first while-loop finds the *leftmost* position of P in A. The second loop works symmetrically. Note further that in the second while-loop it is enough to check for lexicographical equality, as the whole search is done in the interval of A where all suffixes are lexicographically no less than P.

Theorem 3. *The suffix array allows to answer counting queries in $O(m \log n)$ time, and reporting queries in $O(m \log n + |O_P|)$ time.*

2.5 Construction of Suffix Trees from Suffix- and LCP-Arrays

Assume for now that we are given T, A, and H, and we wish to construct S, the suffix tree of T. We will show in this section how to do this in $O(n)$ time. Later, we will also see how to construct A and H only from T in linear time. In total, this will give us an $O(n)$-time construction algorithm for suffix trees.

The idea of the algorithm is to insert the suffixes into S in the order of the suffix array: $T^{A[1]}, T^{A[2]}, \ldots, T^{A[n]}$. To this end, let S_i denote the partial suffix tree for $0 \leq i \leq n$ (S_i is the compact Σ^+-tree with $\mathit{words}(S_i) = \{T_{A[k]\ldots j} : 1 \leq k \leq i, A[k] \leq j \leq n\}$). In the end, we will have $S = S_n$.

We start with S_0, the tree consisting only of the root (and thus displaying only ϵ). In step $i+1$, we climb up the *rightmost path* of S_i (i.e., the path from the leaf labeled $A[i]$ to the root) until we meet the deepest node v with $d(v) \leq H[i+1]$. If $d(v) = H[i+1]$, we simply insert a new leaf x to S_i as a child of v, and label (v, x) by $T^{A[i+1]+H[i+1]}$. Leaf x is labeled by $A[i+1]$. This gives us S_{i+1}.

Otherwise (i.e., $d(v) < H[i+1]$), let w be the child of v on S_i's rightmost path. In order to obtain S_{i+1}, we *split up* the edge (v, w) as follows.

1. Delete (v, w).
2. Add a new node y and a new edge (v, y). (v, y) gets labeled by $T_{A[i]+d(v)\ldots A[i]+H[i+1]-1}$.
3. Add (y, w) and label it by $T_{A[i]+H[i+1]\ldots A[i]+d(w)-1}$.
4. Add a new leaf x (labeled $A[i+1]$) and an edge (y, x). Label (y, x) by $T^{A[i+1]+H[i+1]}$.

The correctness of this algorithm follows from observations 1 and 2 above. Let us now consider the execution time of this algorithm. Although climbing up the rightmost path could take $O(n)$ time in a single step, a simple amortized argument shows that the running time of this algorithm can be bounded by $O(n)$ in total: each node traversed in step i (apart from the last) is *removed* from the rightmost path and will not be traversed again for all subsequent steps $j > i$. Hence, at most $2n$ nodes are traversed in total.

Theorem 4. *We can construct T's suffix tree in linear time from T's suffix- and LCP-array.*

2.6 Linear-Time Construction of Suffix Arrays

Now we explain the *induced sorting* algorithm for constructing suffix arrays. Its basic idea is to sort a certain subset of suffixes recursively, and then use this result to *induce* the order of the remaining suffixes.

- Ge Nong, Sen Zhang, Wai Hong Chan: *Two Efficient Algorithms for Linear Time Suffix Array Construction.* IEEE Trans. Computers **60**(10): 1471–1484 (2011).

Definition 8. *For $1 \leq i < n$, suffix T^i is said to be* S-type *if $T^i <_{\text{lex}} T^{i+1}$, and* L-type *otherwise. The last suffix is defined to be S-type. For brevity, we also use the terms* S- *and* L-suffixes *for suffixes of the corresponding type.*

The type of each suffix can be determined in linear time by a right-to-left scan of T: first, T^n is declared as S-type. Then, for every i from $n-1$ to 1, T^i is classified by the following rule:

$$T^i \text{ is S-type iff either } t_i < t_{i+1}, \text{ or } t_i = t_{i+1} \text{ and } T^{i+1} \text{ is S-type.}$$

We further say that an S-suffix T^i is of *type S^** iff T^{i-1} is of type L. (Note that the S-suffixes still include the S*-suffixes in what follows.)

In A, all suffixes starting with the same character $c \in \Sigma$ form a consecutive interval, called the *c-bucket* henceforth. Observe that in any c-bucket, the L-suffixes precede the S-suffixes. Consequently, we can sub-divide buckets into S-type buckets and L-type buckets.

Now the induced sorting algorithm can be explained as follows:

1. Sort the S*-suffixes. This step will be explained in more detail below.
2. Put the sorted S*-suffixes into their corresponding S-buckets, without changing their order.
3. Induce the order of the L-suffixes by scanning A from left to right: for every position i in A, if $T^{A[i]-1}$ is L-type, write $A[i]-1$ to the current head of the L-type c-bucket ($c = t_{A[i]-1}$), and increase the current head of that bucket by one. Note that this step can only induce "to the right" (the current head of the c-bucket is larger than i).
4. Induce the order of the S-suffixes by scanning A from *right to left*: for every position i in A, if $T^{A[i]-1}$ is S-type, write $A[i]-1$ to the current *end* of the S-type c-bucket ($c = t_{A[i]-1}$), and *de*crease the current end of that bucket by one. Note that this step can only induce "to the left," and might intermingle S-suffixes with S*-suffixes.

It remains to explain how the S*-suffixes are sorted (step 1 above). To this end, we define:

Definition 9. *An* S*-substring *is a substring $T_{i..j}$ with $i \neq j$ of T such that both T^i and T^j are S*-type, but no suffix in between i and j is also of type S*.*

Let $R_1, R_2, \ldots, R_{n'}$ denote these S*-substrings, and σ' be the number of different S*-substrings. We assign a *name* $v_i \in [1, \sigma']$ to any such R_i, such that $v_i < v_j$ if $R_i <_{\text{lex}} R_j$ and $v_i = v_j$ if $R_i = R_j$. We then construct a new text $T' = v_1 \ldots v_{n'}$ over the alphabet $[1, \sigma']$, and build the suffix array A' of T' by applying the inducing sorting algorithm *recursively* to T' if $\sigma' < n'$ (otherwise there is nothing to sort, as then the order of the S*-suffixes is given by the order of the S*-substrings). The crucial property to observe here is that the order of the suffixes in T' is the same as the order of the respective S*-suffixes in T; hence, A' determines the sorting of the S*-suffixes in T. Further, as at most every second suffix in T can be of type S*, the complete algorithm has worst-case running time $T(n) = T(n/2) + O(n) = O(n)$, provided that the *naming* of the S*-substrings also takes linear time, which is what we explain next.

The naming of the S*-substrings is similar to the inducing of the S-suffixes in the induced sorting algorithm (steps 2–4 above), with the difference that in step 2 we put the *unsorted* S*-suffixes into their corresponding buckets (hence they are only sorted according to their first character). Steps 3 and 4 work exactly as described above. At the end of step 4, we can assign names to the S*-substrings by comparing adjacent S*-suffixes naively until we find a mismatch or reach their end; this takes overall linear time.

Theorem 5. *We can construct the suffix array for a text of length n in $O(n)$ time.*

2.7 Linear-Time Construction of LCP-Arrays

We now explain how the induced sorting algorithm (Sect. 2.6) can be modified to also compute the LCP-array. The basic idea is that whenever we place two S- or L-suffixes T^{i-1} and T^{j-1} at adjacent places $k-1$ and k in the final suffix array (steps 3 and 4 in the algorithm), the length of their longest common prefix can be induced from the longest common prefix of the suffixes T^i and T^j. As the latter suffixes are exactly those that caused the inducing of T^{i-1} and T^{j-1}, we already know their LCP-value ℓ (by the order in which we fill A), and can hence set $H[k]$ to $\ell+1$.

2.7.1 Basic Algorithm

We now describe the algorithm in more detail. We augment the steps of the induced sorting algorithm as follows:

1′. Compute the LCP-values of the S*-suffixes (see Sect. 2.7.3).

2′. Whenever we place an S*-suffix into its S-bucket, we also store its LCP-value at the corresponding position in H.

3′. Suppose that the inducing step just put suffix $T^{A[i]-1}$ into its L-type c-bucket at position k. If $T^{A[i]-1}$ is the first suffix in its L-bucket, we set $H[k]$ to 0. Otherwise, suppose further that in a previous iteration $i' < i$ the inducing step placed suffix $T^{A[i']-1}$ at $k-1$ in the same c-bucket. Then if i' and i are in different buckets, the suffixes $T^{A[i]}$ and $T^{A[i']}$ start with different characters, and we set $H[k]$ to 1, as the suffixes $T^{A[i]-1}$ and $T^{A[i']-1}$ share only a common character c at their beginnings. Otherwise (i' and i are in the same c'-bucket), the length ℓ of the longest common prefix of the suffixes $T^{A[i]}$ and $T^{A[i']}$ is given by the *minimum* value in $H[i'+1, i]$, all of which are in the same c'-bucket and have therefore already been computed in previous iterations. We can hence set $H[k]$ to $\ell+1$.

4′. As in the previous step, suppose that the inducing step just put suffix $T^{A[i]-1}$ into its S-type c-bucket at position k. Suppose further that in a previous iteration $i' > i$ the inducing step placed suffix $T^{A[i']-1}$ at $k+1$ in the same c-bucket (if k is the last position in its S-bucket, we skip the following steps). Then if i' and i are in different buckets, their suffixes start with different characters, and we set $H[k+1]$ to 1, as the suffixes $T^{A[i]-1}$ and $T^{A[i']-1}$ share only a common character c at their beginnings. Otherwise (i' and i are in the same c'-bucket), the length ℓ of the longest common prefix of the suffixes $T^{A[i]}$ and $T^{A[i']}$ is given by the *minimum* value in $H[i+1, i']$, all of which are in the same c'-bucket and have therefore already been computed. We can hence set $H[k+1]$ to $\ell+1$.

(We will resolve the problem of computing the LCP-value between the last L-suffix and the first S-suffix in a bucket at the end of this section.)

2.7.2 Finding Minima

To find the minimum value in $H[i'+1,i]$ or $H[i+1,i']$ (steps $3'$ and $4'$ above), we have several alternatives. The simplest idea is to scan the whole interval from $i'+1$ to i; this results in overall $O(n^2)$ running time. A better alternative would be to keep an array M of size σ, such that the minimum is always given by $M[c]$ if we induce an LCP-value in bucket c. To keep M up-to-date, after each step i we first set $M[c]$ to $H[i]$, and further update all other entries in M that are larger than $H[i]$ by $H[i]$; this approach has $O(n\sigma)$ running time. A further refinement of this technique stores the values in M in sorted order and uses binary search on M to find the minima; this results in overall $O(n \log \sigma)$ running time.

In subsequent sections, we will see how to preprocess an arbitrary *static* array of length n into a data structure of size $O(n)$ such that subsequently we can find the minimum in arbitrary ranges in the array in $O(1)$ time (so called *range minimum queries* or RMQs). With some changes, this data structure can also be applied to our *dynamic* LCP-array, as briefly outlined next for step $3'$ (the adjustments for step $4'$ or obvious). Recall that the queries lie within a single bucket (called c'), and every bucket is subdivided into an L- and an S-bucket. The idea is to also subdivide the query into an L- and an S-query, and return the minimum of the two. The S-queries are simple to handle: in step $3'$, only S*-suffixes will be scanned, and these are static. Hence, we can preprocess every S-type bucket with a static data structure for constant-time range minima, using overall linear space. The L-queries are more difficult, as elements keep being written to them during the scan. However, these updates occur in a very regular fashion, namely in a left-to-right manner. This makes the problem much simpler. Also, the sizes of the arrays to be prepared for RMQs are known in advance (namely the sizes of the L-buckets). As we shall see later, the data structure for constant-time RMQs consists of precomputed queries for suitably-sized subarrays. Hence, whenever a subarray to be precomputed for RMQs has been entirely filled, we precompute the answer to this RMQ. As only completely filled subarrays will be queried, this is enough to answer all occurring RMQs.

2.7.3 Computing LCP-Values of S*-suffixes

This section describes how to compute the LCP-values of the suffixes in the sample set (step $1'$ above). The recursive call to compute the suffix array A' for the text T' (the text formed by the names of the S*-substrings) also yields the LCP-array H' for T'. The problem is that these LCP-values refer to characters v_i in the reduced alphabet $[1,\sigma']$, which correspond to S*-substrings R_i in T. Hence, we need to "scale" every LCP-value in H' by the lengths of the actual S*-substrings that constitute this longest common prefix: a value $H'[k]$ refers to the substring $v_{A'[k]} \dots v_{A'[k]+H'[k]-1}$ of T', and actually implies an LCP-value of at least $\sum_{i=0}^{H'[k]-1} |R_{A'[k]+i}|$ between the corresponding S*-suffixes in T. We say "at least" because also the first mismatching reduced characters $v_{A'[k]+H'[k]}$ and $v_{A'[k-1]+H'[k]}$ could contribute some actual characters to the true LCP; we will come back to this issue at the end of this section.

A naive implementation of the sum $\sum_{i=0}^{H'[k]-1} |R_{A'[k]+i}|$ could again result in $O(n^2)$ running time, consider the text $T = \texttt{abab}\dots\texttt{ab}$. However, we can make use of the fact that the suffixes of T' appear lexicographically ordered in T': when "scaling" $H'[k]$, we know that the first $m =$

$\min(H'[k-1], H'[k])$ S*-substrings match, and can hence compute the actual LCP-value as

$$\sum_{i=0}^{H'[k]-1} |R_{A'[k]+i}| = \underbrace{\sum_{i=0}^{m-1} |R_{A'[k]+i}|}_{\text{already computed}} + \sum_{i=m}^{H'[k]-1} |R_{A'[k]+i}| \ .$$

This way, by an amortized argument it is easy to see that each character in T contributes to at most 2 additions, resulting in an overall $O(n)$ running time.

The final thing to do is to account for the fact that the first mismatching reduced characters $v_{A'[k]+H'[k]}$ and $v_{A'[k-1]+H'[k]}$ could contribute some actual characters to the true LCP. Stated differently, we need to know the LCP-values of arbitrary pairs of S*-*substrings.* To this end, we take the sorted S*-substrings in order, and compute the LCP of neighbouring entries naively in overall $O(n)$ time (this step can actually be incorporated without any extra costs into the naming step of the SA-construction). Then the LCP of any two S*-substrings corresponds to the minimum LCP-value between the two strings in the sorted list, which can be found in $O(1)$ time with a data structure for constant-time RMQs (to be discussed later).

2.7.4 Computing LCP-values at the L/S-Seam

There is one subtlety in the above inducing algorithm we have withheld so far, namely that of computing the LCP-values between the last L-suffix and the first S-suffix in a given c-bucket (we call this position the *L/S-seam*). More precisely, when reaching an L/S-seam in step $3'$, we have to re-compute the LCP-value between the first S*-suffix in the c-bucket (if it exists) and the last L-suffix in the same c-bucket (the one that we just induced), in order to induce correct LCP-values when stepping through the S*-suffixes in subsequent iterations. Likewise, when placing the very first S-suffix in its c-bucket in step $4'$, we need to compute the LCP-value between this induced S-suffix and the largest L-suffix in the same c-bucket. (Note that step 4 might place an S-suffix before all S*-suffixes, so we cannot necessarily re-use the LCP-value computed at the L/S-seam in step $3'$.)

The following lemma shows that the LCP-computation at L/S-seams is particularly easy:

Lemma 6. *Let T^i be an L-suffix, T^j an S-suffix, and $t_i = c = t_j$ (the suffixes are in the same c-bucket in A). Further, let $\ell \geq 1$ denote the length of the longest common prefix of T^i and T^j. Then*

$$T_{i\ldots i+\ell-1} = c^\ell = T_{j\ldots j+\ell-1} \ .$$

Proof. Assume that $t_{i+k} = c' = t_{i+k}$ for some $2 \leq k < \ell$ and $c' \neq c$. Then if $c' < c$, both T^i and T^j are of type L, and otherwise ($c' > c$), they are both of type S. In any case, this is a contradiction to the assumption that T^i is of type L, and T^j of type S. □

In words, the above lemma states that the longest common prefix at the L/S-seam can only consist of equal characters. Therefore, a *naive* computation of the LCP-values at the L/S-seam is sufficient to achieve overall linear running time: every character t_i contributes at most to the computation at the L/S-seam in the t_i-bucket, and not in any other c-bucket for $c \neq t_i$.

Theorem 7. *We can construct the LCP array for a text of length n in $O(n)$ time.*

3 Range Minimum Queries

Range Minimum Queries (RMQs) are a versatile tool for many tasks in exact and approximate pattern matching, as we shall see at various points in this lecture. They ask for the position of the minimum element in a specified sub-array, formally defined as follows.

Definition 10. *Given an array $H[1,n]$ of n integers (or any other objects from a totally ordered universe) and two indices $1 \leq i \leq j \leq n$, $\text{RMQ}_H(i,j)$ is defined as the position of the minimum in H's sub-array ranging from i to j, in symbols: $\text{RMQ}_H(i,j) = \text{argmin}_{i \leq k \leq j} H[k]$.*

We often omit the subscript H if the array under consideration is clear from the context.

Of course, an RMQ can be answered in a trivial manner by *scanning* $H[i,j]$ (H's sub-array ranging from position i to j) for the minimum each time a query is posed. In the worst case, this takes $O(n)$ query time.

However, if H is static and known in advance, and there are several queries to be answered on-line, it makes sense to *preprocess* H into an auxiliary data structure (called *index* or *scheme*) that allows to answer future queries faster. As a simple example, we could precompute all possible $\binom{n+1}{2}$ RMQs and store them in a table M of size $O(n^2)$ — this allows to answer future RMQs in $O(1)$ time by a single lookup at the appropriate place in M.

We will show in this section that this naive approach can be dramatically improved, as the following proposition anticipates:

Proposition 8. *An array of length n can be preprocessed in time $O(n)$ such that subsequent range minimum queries can be answered in optimal $O(1)$ time.*

3.1 Linear Equivalence of RMQs and LCAs

Recall the definition of range minimum queries (RMQs): $\text{RMQ}_D(\ell, r) = \text{argmin}_{\ell \leq k \leq r} D[k]$ for an array $D[1,n]$ and two indices $1 \leq \ell \leq r \leq n$. We show in this section that a seemingly unrelated problem, namely that of computing *lowest common ancestors* (LCAs) in static rooted trees, can be reduced quite naturally to RMQs.

Definition 11. *Given a rooted tree T with n nodes, $\text{LCA}_T(v,w)$ for two nodes v and w denotes the unique node ℓ with the following properties:*

1. *Node ℓ is an ancestor of both v and w.*
2. *No descendant of ℓ has property (1).*

Node ℓ is called the lowest common ancestor *of v and w.*

The reduction of an LCA-instance to an RMQ-instance works as follows:

- Let r be the root of T with children $u_1, \ldots, u_k$.
- Define T's *inorder tree walk array* $I = I(T)$ recursively as follows:
  - If $k = 0$, then $I = [r]$.
  - If $k = 1$, then $I = I(T_{u_1}) \circ [r]$.

  - Otherwise, $I = I(T_{u_1}) \circ [r] \circ I(T_{u_2}) \circ [r] \circ \cdots \circ [r] \circ I(T_{u_k})$, where "$\circ$" denotes array concatenation. Recall that T_v denotes T's subtree rooted at v.

- Define T's *depth array* $D = D(T)$ (of the same length as I) such that $D[i]$ equals the tree-depth of node $I[i]$.
- Augment each node v in T with a "pointer" p_v to an arbitrary occurrence of v in I ($p_v = j$ only if $I[j] = v$).

Lemma 9. *The length of I (and of D) is between n (inclusively) and $2n$ (exclusively).*

Proof. By induction on n.

$n = 1$: The tree T consists of a single leaf v, so $I = [v]$ and $|I| = 1 < 2n$.

$\leq n \to n + 1$: Let r be the root of T with children $u_1, \ldots, u_k$. Let n_i denote the number of nodes in T_{u_i}. Recall $I = I(T_{u_1}) \circ [r] \circ \cdots \circ [r] \circ I(T_{u_k})$. Hence,

$$\begin{aligned}
|I| &= \max(k-1, 1) + \sum_{1 \leq i \leq k} |I(T_{u_i})| \\
&\leq \max(k-1, 1) + \sum_{1 \leq i \leq k} (2n_i - 1) \qquad \text{(by the induction hypothesis)} \\
&= \max(k-1, 1) - k + 2 \sum_{1 \leq i \leq k} n_i \\
&\leq 1 + 2 \sum_{1 \leq i \leq k} n_i \\
&= 1 + 2(n-1) \\
&< 2n\ . \quad \square
\end{aligned}$$

Here comes the desired connection between LCA and RMQ:

Lemma 10. *For any pair of nodes v and w in T, $\text{LCA}_T(v, w) = I[\text{RMQ}_D(p_v, p_w)]$.*

Proof. Consider the inorder tree walk $I = I(T)$ of T. Assume $p_v \leq p_w$ (otherwise swap). Let ℓ denote the LCA of v and w, and let $u_1, \ldots, u_k$ be ℓ's children. Look at

$$I(T_\ell) = I(T_{u_1}) \circ \cdots \circ I(T_{u_x}) \circ [\ell] \circ \cdots \circ [\ell] \circ I(T_{u_y}) \circ \cdots \circ I(T_{u_k})$$

such that $v \in T_{u_x}$ and $w \in T_{u_y}$ ($v = \ell$ or $w = \ell$ can be proved in a similar manner).

Note that $I(T_\ell)$ appears in I exactly the same order, say from a to b: $I[a, b] = I(T_\ell)$. Now let d be the tree depth of ℓ. Because ℓ's children u_i have a greater tree depth than d, we see that D attains its minima in the range $[a, b]$ only at positions i where the corresponding entry $I[i]$ equals ℓ. Because $p_v, p_w \in [a, b]$, and because the inorder tree walk visits ℓ between u_x and u_y, we get the result. $\square$

To summarize, if we can solve RMQs in $O(1)$ time using $O(n)$ space, we also have a solution for the LCA-problem within the same time- and space-bounds.

Interestingly, this reduction also works the other way around: a linear-space data structure for $O(1)$ LCAs implies a linear-space data structure for $O(1)$ RMQs. To this end, we need the concept of Cartesian Trees:

Definition 12. *Let $A[1, s]$ be an array of size n. The* Cartesian Tree $\mathcal{C}(A)$ *of A is a labelled binary tree, recursively defined as follows:*

- *Create a root node r and label it with $p = \operatorname{argmin}_{1 \le i \le n} A[i]$.*
- *The left and right children of r are the roots of the Cartesian Trees $\mathcal{C}(A[1, p-1])$ and $\mathcal{C}(A[p+1, n])$, respectively (if existent).*

Constructing the Cartesian Tree according to this definition requires $O(n^2)$ time (scanning for the minimum in each recursive step), or maybe $O(n \log n)$ time after an initial sorting of A. However, there is also a linear time **algorithm** for constructing $\mathcal{C}(A)$, which we describe next.

Let $\mathcal{C}_i$ denote the Cartesian Tree for $A[1, i]$. Tree $\mathcal{C}_1$ just consists of a single node r labelled with 1. We now show how to obtain $\mathcal{C}_{i+1}$ from $\mathcal{C}_i$. Let the *rightmost path* of $\mathcal{C}_i$ be the path $v_1, \ldots, v_k$ in $\mathcal{C}_i$, where v_1 is the root, and v_k is the node labelled i. Let l_i be the label of node v_i for $1 \le i \le k$.

To get $\mathcal{C}_{i+1}$, climb up the rightmost path (from v_k towards the root v_1) until finding the first node v_y where the corresponding entry in A is not larger than $A[i+1]$:

$$A[l_y] \le A[i+1], \text{and } A[l_z] > A[i+1] \text{ for all } y < z \le k \ .$$

Then insert a new node w as the right child of v_y (or as the root, if v_y does not exist), and label w with $i + 1$. Node v_{y+1} becomes the left child of w. This gives us $\mathcal{C}_{i+1}$.

The linear running time of this algorithm can be seen by the following amortized argument: each node is inserted onto the rightmost path exactly once. All nodes on the rightmost path (except the last, v_y) traversed in step i are removed from the rightmost path, and will never be traversed again in steps $j > i$. So the running time is proportional to the total number of removed nodes from the rightmost path, which is $O(n)$, because we cannot remove more nodes than we insert.

How is the Cartesian Tree related to RMQs?

Lemma 11. *Let A and B be two arrays with equal Cartesian Trees. Then $\text{RMQ}_A(\ell, r) = \text{RMQ}_B(\ell, r)$ for all $1 \le \ell \le r \le n$.*

Proof. By induction on n.

$n = 1$: $\mathcal{C}(A) = \mathcal{C}(B)$ consists of a single node labelled 1, and $\text{RMQ}(1, 1) = 1$ in both arrays.

$\le n \to n + 1$: Let v be the root of $\mathcal{C}(A) = \mathcal{C}(B)$ with label μ. By the definition of the Cartesian Tree,

$$\operatorname*{argmin}_{1 \le k \le n} A[k] = \mu = \operatorname*{argmin}_{1 \le k \le n} B[k] \ . \tag{1}$$

Because the left (and right) children of $\mathcal{C}(A)$ and $\mathcal{C}(B)$ are roots of the same tree, this implies that the Cartesian Trees $\mathcal{C}(A[1, \mu-1])$ and $\mathcal{C}(B[1, \mu-1])$ (and $\mathcal{C}(A[\mu+1, n])$ and $\mathcal{C}(B[\mu+1, n])$) are equal. Hence, by the induction hypothesis,

$$\text{RMQ}_A(\ell, r) = \text{RMQ}_B(\ell, r) \forall 1 \le \ell \le r < \mu, \text{and } \text{RMQ}_A(\ell, r) = \text{RMQ}_B(\ell, r) \forall \mu < \ell \le r \le n. \tag{2}$$

In total, we see that $\text{RMQ}_A(\ell, r) = \text{RMQ}_B(\ell, r)$ for all $1 \le \ell \le r \le n$, because a query must either contain position μ (in which case, by (1), μ is the answer to both queries), or it must be completely to the left/right of μ (in which case (2) gives what we want). □

3.2 $O(1)$-RMQs with $O(n \log n)$ Space

We already saw that with $O(n^2)$ space, $O(1)$-RMQs are easy to realize by simply storing the answers to all possible RMQs in a two-dimensional table of size $n \times n$. We show in this section a little trick that lowers the space to $O(n \log n)$.

The basic idea is that it suffices to precompute the answers only for query lengths that are a *power of 2*. This is because an arbitrary query $\text{RMQ}_D(l, r)$ can be decomposed into two overlapping sub-queries of equal length 2^h with $h = \lfloor \log_2(r - l + 1) \rfloor$:

$$m_1 = \text{RMQ}_D(l, l + 2^h - 1) \text{ and } m_2 = \text{RMQ}_D(r - 2^h + 1, r)$$

The final answer is then given by $\text{RMQ}_D(l, r) = \operatorname{argmin}_{\mu \in \{m_1, m_2\}} D[\mu]$. This means that the precomputed queries can be stored in a two-dimensional table $M[1, n][1, \lfloor \log_2 n \rfloor]$, such that

$$M[x][h] = \text{RMQ}_D(x, x + 2^h - 1)$$

whenever $x + 2^h - 1 \leq n$. Thus, the size of M is $O(n \log n)$. With the identity

$$\begin{aligned} M[x][h] &= \text{RMQ}_D(x, x + 2^h - 1) \\ &= \operatorname{argmin}\{D[i] : i \in \{x, \ldots, x + 2^h - 1\}\} \\ &= \operatorname{argmin}\{D[i] : i \in \{\text{RMQ}_D(x, x + 2^{h-1} - 1), \text{RMQ}_D(x + 2^{h-1}, x + 2^h - 1)\}\} \\ &= \operatorname{argmin}\{D[i] : i \in \{M[x][h-1], M[x + 2^{h-1}][h-1]\}\} , \end{aligned}$$

we can use *dynamic programming* to fill M in optimal $O(n \log n)$ time.

3.3 $O(1)$-RMQs with $O(n)$ Space

We divide the input array D into blocks $B_1, \ldots, B_m$ of size $s := \frac{\log_2 n}{4}$ (where $m = \lceil \frac{n}{s} \rceil$ denotes the number of blocks): $B_1 = D[1, s]$, $B_2 = D[s + 1, 2s]$, and so on. The reason for this is that any query $\text{RMQ}_D(l, r)$ can be decomposed into at most three non-overlapping sub-queries:

- At most one query spanning exactly over several blocks.
- At most two queries completely inside of a block.

We formalize this as follows: Let $i = \lceil \frac{l}{s} \rceil$ and $j = \lceil \frac{r}{s} \rceil$ be the block numbers where l and r occur, respectively. If $i = j$, then we only need to answer one in-block-query to obtain the final result. Otherwise, $\text{RMQ}_D(l, r)$ is answered by $\text{RMQ}_D(l, r) = \operatorname{argmin}_{\mu \in \{m_1, m_2, m_3\}} D[\mu]$, where the m_i's are obtained as follows:

- $m_1 = \text{RMQ}_D(l, is)$
- $m_2 = \text{RMQ}_D(is + 1, (j - 1)s)$ (only necessary if $j > i + 1$)
- $m_3 = \text{RMQ}_D((j - 1)s + 1, r)$

We first show how to answer queries spanning exactly over several blocks (i.e., finding m_2).

3.3.1 Queries Spanning Exactly over Blocks

Define a new array $D'[1,m]$, such that $D'[i]$ holds the minimum inside of block B_i: $D'[i] = \min_{(i-1)s<j\leq is} D[j]$. We then prepare D' for constant-time RMQs with the algorithm from Sect. 3.2, using

$$O(m\log m) = O(\frac{n}{s}\log(\frac{n}{s})) = O(\frac{n}{\log n}\log\frac{n}{\log n}) = O(n)$$

space.

We also define a new array $W[1,m]$, such that $W[i]$ holds the position where $D'[i]$ occurs in D: $W[i] = \operatorname{argmin}_{(i-1)s<j\leq is} D[j]$. A query of the form $\text{RMQ}_D(is+1,(j-1)s)$ is then answered by $W[\text{RMQ}_{D'}(i+1,j-1)]$.

3.3.2 Queries Completely Inside of Blocks

We are left with answering "small" queries that lie completely inside of blocks of size s. These are actually more complicated to handle than the "long" queries from Sect. 3.3.1.

The consequence of this is that we only have to precompute in-block RMQs for blocks with different Cartesian Trees, say in a table called P. But how do we know in $O(1)$ time where to look up the results for block B_i? We need to store a "number" for each block in an array $T[1,m]$, such that $T[i]$ gives the corresponding row in the lookup-table P.

Lemma 12. *A binary tree T with s nodes can be represented uniquely in $2s+1$ bits.*

Proof. We first label each node in T with a '1' (these are not the same labels as for the Cartesian Tree!). In a subsequent traversal of T, we add "missing children" (labelled '0') to every node labelled '1', such that in the resulting tree T' all leaves are labelled '0'. We then list the 0/1-labels of T' *level-wise* (i.e., first for the root, then for the nodes at depth 1, then for depth 2, etc.). This uses $2s+1$ bits, because in a binary tree without nodes of out-degree 1, the number of leaves equals the number of internal nodes plus one.

It is easy to see how to reconstruct T from this sequence. Hence, the encoding is unique. □

So we perform the following steps:

1. For every block B_i, we compute the bit-encoding of $\mathcal{C}(B_i)$ and store it in $T[i]$. Because $s = \frac{\log n}{4}$, every bit-encoding can be stored in a single computer word.

2. For every *possible* bit-vector t of length $2s+1$ that describes a binary tree on s nodes, we store the answers to all RMQs in the range $[1,s]$ in a table:

   $P[t][l][r] = \text{RMQ}_B(l,r)$ for some array B of size s whose Cartesian Tree has bit-encoding t

Finally, to answer a query $\text{RMQ}_D(l,r)$ which is completely contained within a block $i = \lceil\frac{l}{s}\rceil = \lceil\frac{r}{s}\rceil$, we simply look up the result in $P[T[i]][l-(i-1)s][r-(i-1)s]$.

To analyze the space, we see that T occupies $m = n/\log n = O(n)$ words. It is perhaps more surprising that also P occupies only a linear number of words, namely order of

$$2^{2s}\cdot s\cdot s = \sqrt{n}\cdot\log^2 n = O(n)\ .$$

Construction time of the data structures is $O(ms) = O(n)$ for T, and $O(2^{2s}\cdot s\cdot s\cdot s) = O(\sqrt{n}\cdot\log^3 n) = O(n)$ for P (the additional factor s accounts for finding the minimum in each precomputed query interval).

This finishes the description of the algorithm.

4 Repeats

Definition 13. *Let $T = t_1 t_2 \dots t_n$ be a text of length n. A triple (i, j, ℓ) with $1 \leq i < j \leq n - \ell + 1$ is called* repeat *if $T_{i \dots i+\ell-1} = T_{j \dots j+\ell-1}$.*

We look at three different tasks:

1. Output all triples (i, j, ℓ) that are a repeat according to Def. 13.
2. Output all strings $\alpha \in \Sigma^\star$ such that there is a repeat (i, j, ℓ) with $T_{i \dots i+\ell-1} = \alpha (= T_{j \dots j+\ell-1})$.
3. Like (2), but instead of outputting α just output a pair (l, r) with $\alpha = T_{l \dots r}$.

Task (1) yields probably many triples (consider $T = \mathtt{a}^n$), whereas the returned strings in task (2) may probably be very long. The ouput in task (3) may be seen as a space-efficient representation of (2). Nevertheless, in most cases we are happy with repeats that cannot be extended, as captured in the following section.

4.1 Maximal Repeats

Definition 14. *A repeat (i, j, ℓ) is called*

- left-maximal*, if $t_{i-1} \neq t_{j-1}$.*
- right-maximal*, if $t_{i+\ell} \neq t_{j+\ell}$.*
- maximal*, if it is both left- and right-maximal.*

To make this definition valid for the text borders, we extend T with $t_0 = \pounds$ to the left, and with $t_{n+1} = \$$ to the right ($\pounds, \$ \notin \Sigma$).

Observation 3. *If (i, j, ℓ) is a right-maximal repeat, then there must be an* internal *node v in T's suffix tree S with $\overline{v} = T_{i \dots i+\ell-1}$. For otherwise S could not display both $T_{i \dots i+\ell}$ and $T_{j \dots j+\ell}$, which must be different due to $t_{i+\ell} \neq t_{j+\ell}$.*

Let

$$\mathcal{R}_T = \{\alpha \in \Sigma^\star \ : \ \text{there is a maximal repeat } (i, j, \ell) \text{ with } T_{i \dots i+\ell-1} = \alpha\}$$

be the set of T's maximal repeat strings (task (2) above). Then the observation above shows that $|\mathcal{R}_T| < n$, as there are only n leaves and hence less than n internal nodes in the suffix tree. Hence, for task (3) we should be able to come up with a $O(n)$-time algorithm.

It remains to show how left-maximality can be checked efficiently.

Definition 15. *Let S be T's suffix tree. A node v in S is called* left-diverse *if there are at least two leaves b_1 and b_2 below v such that $t_{\ell(b_1)-1} \neq t_{\ell(b_2)-1}$. [Recall that $\ell(\cdot)$ denotes the leaf label (=suffix number)]. Character $t_{\ell(v)-1}$ is called v's* left-character*.*

Lemma 13. *A repeat (i, j, ℓ) is maximal iff there is left-diverse node v in T's suffix tree S with $\overline{v} = T_{i \dots i+\ell-1}$.*

Proof. It remains to care about left-maximality.

"$\Rightarrow$" Let (i, j, ℓ) be maximal. Let v be the node in S with $\overline{v} = T_{i \dots i+\ell-1}$, which must exist due to right-maximality. Due to left-maximality, we know $t_{i-1} \neq t_{j-1}$. Hence there are two different leaves b_1 and b_2 below with $\ell(b_1) = i$ and $\ell(b_2) = j$. So v is left-diverse.

"$\Leftarrow$" analogous. □

This yields the following **algorithm** to compute $\mathcal{R}_T$:

In a depth-first search through S do:

- Let v be the current node.
- If v is a leaf: propagate v's left-character to its parent.
- If v is internal with children $v_1, \dots, v_k$:
  - If one of the v_i's is left-diverse, or if at least two of the v_i's have a different left-character: output $\overline{v}$ and propagate "left-diverse" to the parent.
  - Otherwise, propagate the unique left-character of the v_i's to the parent.

We formulated the above algorithm to solve task (2) above, but it can easily be adapted to task (1) or (3). Note in particular the *linear* running time for (3). For (1), we also have to propagate lists of positions $L_a(v)$ to the parent, where $L_a(v)$ contains all leaf labels below v that have $a \in \Sigma$ as their left-character. These lists have to be concatenated in linear time (using linked lists with additional pointers to the ends), and in each step we have to output the Cartesian product of $L_a(v)$ and $L_b(v)$ for all $a, b \in \Sigma$, $a \neq b$. The resulting algorithm is still optimal (in an output-sensitive meaning).

4.2 Super-Maximal Repeats

The maximal repeats in Def. 14 can still contain other maximal repeats, as the example $T =$ `axybxxyyyaxyb` shows (both `xy` and `axyb` are maximal repeats, for example). This is prevented by the following definition:

Definition 16. *A maximal repeat (i, j, ℓ) is called* super-maximal *if there is no maximal repeat (i', j', ℓ') such that $T_{i \dots i+\ell-1}$ is a proper subword of $T_{i' \dots i'+\ell-1}$.*

The algorithmic difference to the previous section is that we only have to consider internal nodes whose children are all *leaves*. Hence, we can also report all k super-maximal repeats in output-optimal time $O(n + k)$.

4.3 Longest Common Substrings

As a last simple example of repeated sequences, consider the following problem: We are given two strings T_1 and T_2. Our task is to return the longest string $\alpha \in \Sigma^\star$ which occurs in both T_1 and T_2.

Computer-science pioneer Don Knuth conjectured in the late 60's that no linear-time algorithm for this problem can exist. However, he was deeply wrong, as suffix trees make the solution almost trivial: Build a suffix tree S for $T = T_1\#T_2$. In a DFS through S (where v is the current node), propagate to v's parent from which of the T_i's the suffixes below v come (either from T_1, T_2, or from both). During the DFS, remember the node w of greatest string depth which has suffixes from both T_1 and T_2 below it. In the end, $\overline{w}$ is the solution. Total time is $O(n)$ for $n = |T_1| + |T_2|$.

5 Tandem Repeats and Related Repetitive Structures

As usual, let $T = t_1 t_2 \dots t_n$ be a string of length n over an alphabet Σ. Our ultimate goal will be to find all *tandem repeats* in T, i.e. subwords of the form $\alpha\alpha$ for some $\alpha \in \Sigma^+$. Recall that $T_{i\dots j}$ is a shorthand for $t_i t_{i+1} \dots t_j$, and is defined to be the empty string ϵ if $j < i$.

5.1 Longest Common Prefixes and Suffixes

An indispensable tool in pattern matching are efficient implementations of functions that compute *longest common prefixes* and *longest common suffixes* of two strings. We will be particularly interested in longest common prefixes of suffixes from the same string T:

Definition 17. *For a text T of length n and two indices $1 \leq i, j \leq n$, $\textsc{lcp}_T(i,j)$ denotes the length of the longest common prefix of the suffixes starting at position i and j in T, in symbols: $\textsc{lcp}_T(i,j) = \max\{\ell \geq 0 \ : \ T_{i\dots i+\ell-1} = T_{j\dots j+\ell-1}\}$.*

Note that $\textsc{lcp}(\cdot)$ only gives the *length* of the matching prefix; if one is actually interested in the *prefix* itself, this can be obtained by $T_{i\dots i+\textsc{lcp}(i,j)-1}$.

Note also that the LCP-array H from Sect. 2.3 holds the lengths of longest common prefixes of *lexicographically consecutive suffixes*: $H[i] = \textsc{lcp}(A[i], A[i-1])$. Here and in the remainder of this chapter, A is again the suffix array of text T.

But how do we get the lcp-values of suffixes that are *not* in lexicographic neighborhood? The key to this is to employ RMQs over the LCP-array, as shown in the next lemma (recall that A^{-1} denotes the *inverse suffix array* of T).

Lemma 14. *Let $i \neq j$ be two indices in T with $A^{-1}[i] < A^{-1}[j]$ (otherwise swap i and j). Then $\textsc{lcp}(i,j) = H[\textsc{rmq}_H(A^{-1}[i]+1, A^{-1}[j])]$.*

Proof. First note that any common prefix ω of T^i and T^j must be a common prefix of $T^{A[k]}$ for all $A^{-1}[i] \leq k \leq A^{-1}[j]$, because these suffixes are lexicographically *between* T^i and T^j and must hence start with ω. Let $m = \textsc{rmq}_H(A^{-1}[i]+1, A^{-1}[j])$ and $\ell = H[m]$. By the definition of H, $T_{i\dots i+\ell-1}$ is a common prefix of all suffixes $T^{A[k]}$ for $A^{-1}[i] \leq k \leq A^{-1}[j]$. Hence, $T_{i\dots i+\ell-1}$ is a common prefix of T^i and T^j.

Now assume that $T_{i\dots i+\ell}$ is also a common prefix of T^i and T^j. Then, by the lexicographic order of A, $T_{i\dots i+\ell}$ is also a common prefix of $T^{A[m-1]}$ and $T^{A[m]}$. But $|T_{i\dots i+\ell}| = \ell + 1$, contradicting the fact that $H[m] = \ell$ tells us that $T^{A[m-1]}$ and $T^{A[m]}$ share no common prefix of length more than ℓ. $\square$

The above lemma implies that with the inverse suffix array A^{-1}, the LCP-array H, and constant-time RMQs on H, we can answer lcp-queries for arbitrary suffixes in $O(1)$ time.

Now consider the "reverse" problem, that of finding *longest common suffixes* of prefixes.

Definition 18. *For a text T of length n and two indices $1 \leq i, j \leq n$, $\textsc{lcs}_T(i,j)$ denotes the length of the* longest common suffix *of the prefixes ending at position i and j in T, in symbols: $\textsc{lcs}_T(i,j) = \max\{k \geq 0 : T_{i-k+1\dots i} = T_{j-k+1\dots j}\}$.*

For this, it suffices to build the *reverse* string $\tilde{T}$, and prepare it for lcp-queries as shown before. Then $\textsc{lcs}_T(i,j) = \textsc{lcp}_{\tilde{T}}(n-i+1, n-j+1)$.

5.2 Tandem Repeats and Runs

Definition 19. *A string $S \in \Sigma^*$ that can be written as $S = \omega^k$ for $\omega \in \Sigma^+$ and $k \geq 2$ is called a* tandem repeat*.*

The usual task is to extract all tandem repeats from a given sequence of nucleotides (or, less common, amino acids). We refer the reader to Sect. 7.11.1 of D. Gusfield's textbook for the significance of tandem repeats in computational biology.

To cope with the existence of overlapping tandem repeats, the concept of runs turns out to be useful.

Definition 20. *Given indices b and e with $1 \leq b \leq e \leq n$ and an integer $\ell \geq 1$, the triple (b, e, ℓ) is called a* run *in T if*

1. *$T_{b\ldots e} = \omega^k \nu$ for some $\omega \in \Sigma^\ell$ and $k \geq 2$, and $\nu \in \Sigma^*$ a (possibly empty) proper prefix of ω.*
2. *If $b > 1$, then $t_{b-1} \neq t_{b+\ell-1}$ ("maximality to the left").*
3. *If $e < n$, then $t_{e+1} \neq t_{e-\ell+1}$ ("maximality to the right").*
4. *There is no $\ell' < \ell$ such that (b, e, ℓ') is also a run ("primitiveness of ω").*

Integer ℓ is called the period *of the run, and $\frac{e-b+1}{\ell}$ its (rational)* exponent*.*

It should be clear that each tandem repeat is contained in a run, and that tandem repeats can be easily extracted from runs. Hence, from now on we concentrate on the *computation of all runs* in T.

The following lemma, which is sometimes taken as the definition of runs, follows easily from Definition 20.

Lemma 15. *Let (b, e, ℓ) be a run in T. Then $t_{j-\ell} = t_j$ for all $b + \ell \leq j \leq e$.*

Proof. We know that $T_{b\ldots e} = \omega^k \nu$ for some ω with $|\omega| = \ell$ and ν a proper prefix of ω. If j is inside of the i'th occurrence of ω in $T_{b\ldots e}$ ($i \geq 2$), going ℓ positions to the left gives the same character in the $(i-1)$'th occurrence of ω in $T_{b\ldots e}$. The same is true if j is inside of ν, because ν is a prefix of ω. $\square$

The following proposition has a venerable history in computer science. Proving it has become simpler since the original publication of Kolpakov and Kucherov from 1999, but it would still take about 4 pages of dense mathematics, so we omit the proof in this lecture.

Proposition 16. *The number of runs in a text of length n is in $O(n)$.*

We now give a first hint at how we can use our previous knowledge to tackle the computation of all runs. From this point onwards, we will not require the period of a run be minimal (fourth point in Definition 20) — this condition can be easily incorporated into the whole algorithm.

Lemma 17. *For $\ell \leq \lfloor \frac{n}{2} \rfloor$ and an index j with $\ell < j \leq n - \ell + 1$, let $s = \text{LCS}(j-1, j-\ell-1)$ and $p = \text{LCP}(j, j-\ell)$. Then the following three statements are equivalent:*

1. *There is a run (b, e, ℓ) in T with $b + \ell \leq j \leq e + 1$*
2. *$s + p \geq \ell$*

3. $(j-\ell-s, j+p-1, \ell)$ *is a run in* T

Proof. (1)⇒(2): Let (b,e,ℓ) be the run. We decompose $T_{b\dots e}$ into three strings u,v,w with $u = T_{b\dots j-\ell-1}$, $v = T_{j-\ell\dots j-1}$, and $w = T_{j\dots e}$, so that $T_{b\dots e} = uvw$ (u and w are possibly empty). Due to Lemma 15, we have that $|u| = s$ and $|w| = p$. Because $|v| = \ell$ by construction and $|T_{b\dots e}| = e-b+1$, we get

$$s+p+\ell = |T_{b\dots e}| = e-b+1 \geq 2\ell ,$$

where the last inequality follows from the "$k \geq 2$" in Definition 20. Hence, $s+p \geq \ell$.
(2)⇒(3): Let $s+p \geq \ell$. First note that due to the definition of lcp/lcs, we get that

$$T_{j-\ell-s\dots j-\ell+p-1} = T_{j-s\dots j+p-1} . \qquad (*)$$

Let $\omega = T_{j-\ell-s\dots j-s-1}$ be the prefix of $T_{j-\ell-s\dots j-\ell+p-1}$ of length ℓ. Due to $(*)$ and the fact that $s+p \geq \ell$ implies that $j-\ell-s+(s+p) \geq j-s$, we get $T_{j-s\dots j+\ell-s-1} = \omega$. This continues until no additional ω fits before position $j+p-1$. We then define the remaining prefix of ω that fits before position $j+p-1$ as ν — this shows that $T_{j-\ell-s\dots j+p-1} = \omega^k\nu$ for some $k \geq 2$. Further, because of the maximality of lcp/lcs we see that $t_{j-\ell-s-1} \neq t_{j-s-1}$, and likewise $t_{j+p} \neq t_{j+p-\ell}$. Hence, $(j-\ell-s, j+p-1, \ell)$ is a run in T.
(3)⇒(1): This is obvious. □

5.3 Algorithm

Lemma 17 gives rise to a first algorithm for computing all the runs in T: simply check for every possible period $\ell \leq \lfloor \frac{n}{2} \rfloor$ and every j if there is a run (b,e,ℓ) with $b+\ell \leq j \leq e+1$, using the lemma.

A key insight is that $e+1 \geq b+2\ell$ for every run (b,e,ℓ), and hence we can increase j by ℓ after every check at position j, without missing any runs. The following pseudo-code incorporates these ideas.

Algorithm 2: $O(n \log n)$-algorithms for locating all runs in a string

```
for $\ell = 1, \dots, \lfloor \frac{n}{2} \rfloor$ do
    $j \leftarrow 2\ell + 1$;
    while $j \leq n+1$ do
        $s \leftarrow \textsc{lcs}(j-1, j-\ell-1)$;
        $p \leftarrow \textsc{lcp}(j, j-\ell)$;
        if $s+p \geq \ell$ and $p < \ell$ then output the run $(j-\ell-s, j+p-1, \ell)$;
        $j \leftarrow j + \ell$;
    end
end
```

In step ℓ of the outer for-loop, $\lceil \frac{n}{\ell} \rceil$ positions j are tested in the inner while-loop. Hence, the total running time is order of

$$\sum_{\ell=1}^{n/2} \frac{n}{\ell} \leq n \sum_{\ell=1}^{n} \frac{1}{\ell} = O(n \log n) .$$

Hence, we get:

Theorem 18. *We can find all runs in a string of length* n *in* $O(n \log n)$ *time.*

In fact, some subtle refinements of the above algorithm (which we do not discuss in this lecture due to lack of time) lead to:

Proposition 19. *We can find all runs in a string of length n in $O(n)$ time.*

6 Lempel-Ziv Compression

6.1 Longest Previous Substring

We now show how to compute an array L of *longest previous substrings*, where $L[i]$ holds the length of the longest prefix of T^i that has another occurrence in T starting *strictly* before i.

Definition 21. *The* longest-previous-substring-array $L[1, n]$ *is defined such that* $L[i] = \max\{\ell \geq 0 : \exists k < i \text{ with } T_{i\ldots i+\ell-1} = T_{k\ldots k+\ell-1}\}$.

Note that for a character $a \in \Sigma$ which has its first occurrence in T at position i, the above definition correctly yields $L[i] = 0$, as in this case any position $k < i$ satisfies $T_{i\ldots i-1} = \epsilon = T_{k\ldots k-1}$.

If we are also interested in the *position* of the longest previous substring, we need another array:

Definition 22. *The array $O[1, n]$ of* previous occurrences *is defined by:*

$$O[i] = \begin{cases} k & \text{if } T_{i\ldots i+L[i]-1} = T_{k\ldots k+L[i]-1} \neq \epsilon \\ \perp & \text{otherwise} \end{cases}$$

A first approach for computing L is given by the following lemma, which follows directly from the definition of L and LCP:

Lemma 20. *For all $2 \leq i \leq n$: $L[i] = \max\{\text{LCP}(i, j) : 1 \leq j < i\}$.* □

For convenience, from now on we assume that both A and H are padded with 0's at their beginning and end: $A[0] = H[0] = A[n + 1] = H[n + 1] = 0$. We further define T^0 to be the empty string ϵ.

Definition 23. *Given the suffix array A and an index $1 \leq i \leq n$ in A, the* previous smaller value *function $PSV_A(\cdot)$ returns the nearest preceding position where A is strictly smaller, in symbols: $PSV_A(i) = \max\{k < i : A[k] < A[i]\}$. The* next smaller value *function $NSV(\cdot)$ is defined similarly for nearest succeeding positions: $NSV_A(i) = \min\{k > i : A[k] < A[i]\}$.*

The straightforward solution that stores the answers to all PSV-/NSV-queries in two arrays $P[1, n]$ and $N[1, n]$ is sufficient for our purposes. Both arrays can be computed from left to right, setting $P[i]$ to $i-1$ if $A[i-1] < A[i]$. Otherwise, continue as follows: if $A[P[i-1]] < A[i]$, set $P[i]$ to $P[i-1]$. And so on ($P[P[i-1]], P[P[P[i-1]]], \ldots$), until reaching the beginning of the array (set $P[0] = -\infty$ for handling the border case). By a similar argument we used for constructing Cartesian trees, this algorithms takes $O(n)$ time.

The next lemma shows how PSVs/NSVs can be used to compute L efficiently:

Lemma 21. *For all $1 \leq i \leq n$, $L[A[i]] = \max(\text{LCP}(A[PSV_A(i)], A[i]), \text{LCP}(A[i], A[NSV_A(i)]))$.*

Proof. Rewriting the claim of Lemma 20 in terms of the suffix array, we get

$$L[A[i]] = \max\{\textsc{lcp}(A[i], A[j]) : A[j] < A[i]\}$$

for all $1 \leq i \leq n$. This can be split up as

$$\begin{aligned} L[A[i]] &= \max(\max\{\textsc{lcp}(A[i], A[j]) : 0 \leq j < i \text{ and } A[j] < A[i]\}, \\ &\qquad \max\{\textsc{lcp}(A[i], A[j]) : i < j \leq n \text{ and } A[j] < A[i]\}) \ . \end{aligned}$$

To complete the proof, we show that $\textsc{lcp}(A[PSV(i)], A[i]) = \max\{\textsc{lcp}(A[i], A[j]) : 0 \leq j < i \text{ and } A[j] < A[i]\}$ (the equation for NSV follows similarly). To this end, first consider an index $j < PSV(i)$. Because of the lexicographic order of A, any common prefix of $T^{A[j]}$ and $T^{A[i]}$ is also a prefix of $T^{A[PSV(i)]}$. Hence, the indices $j < PSV(i)$ need not be considered for the maximum. For the indices j with $PSV(i) < j < i$, we have $A[j] \geq A[i]$ by the definition of PSV. Hence, the maximum is given by $\textsc{lcp}(A[PSV(i)], A[i])$. □

To summarize, we build the array L of longest common substrings in $O(n)$ time as follows:

- Build the suffix array A and the LCP-array H.
- Calculate two arrays P and N such that $PSV_A(i) = P[i]$ and $NSV_A(i) = N[i]$.
- Prepare H for $O(1)$-RMQs, as $\textsc{lcp}(A[PSV(i)], A[i]) = H[\textsc{rmq}_H(P[i] + 1, i)]$ by Lemma 14.
- Build L by applying Lemma 21 to all positions i.

The array O of previous occurrences can be filled along with L, by writing to $O[A[i]]$ the value $A[P[i]]$ if $\textsc{lcp}(A[P[i]], A[i]) \geq \textsc{lcp}(A[N[i]], A[i])$, and the value $A[N[i]]$ otherwise.

6.2 Lempel-Ziv Factorization

Although the Lempel-Ziv factorization is usually introduced for data compression purposes (gzip, WinZip, etc. are all based on it), it also turns out to be useful for efficiently finding repetitive structures in texts, due to the fact that it "groups" repetitions in some useful way.

Definition 24. *Given a text T of length n, its* LZ-decomposition *is defined as a sequence of k strings $s_1, \dots, s_k$, $s_i \in \Sigma^+$ for all i, such that $T = s_1 s_2 \dots s_k$, and s_i is either a single letter not occurring in $s_1 \dots s_{i-1}$, or the longest factor occurring at least twice in $s_1 s_2 \dots s_i$.*

Note that the "overlap" in the definition above exists on purpose, and is not a typo!

We describe the LZ-factorization by a list of k pairs $(b_1, e_1), \dots, (b_k, e_k)$ such that $s_i = T_{b_i \dots e_i}$. We now observe that given our array L of longest previous substrings from the previous section, we can obtain the LZ-factorization quite easily in linear time:

Algorithm 3: $O(n)$-computation of the LZ-factorization

```
$i \leftarrow 1$, $e_0 \leftarrow 0$;
while $e_{i-1} < n$ do
    $b_i \leftarrow e_{i-1} + 1$;
    $e_i \leftarrow b_i + \max(0, L[b_i] - 1)$;
    $++i$;
end
```

7 Burrows Wheeler Transformation

The Burrows-Wheeler Transformation was originally invented for text *compression*. Nonetheless, it was noted soon that it is also a very useful tool in text *indexing*.

7.1 The Transformation

Definition 25. *Let $T = t_1 t_2 \ldots t_n$ be a text of length n, where $t_n = \$$ is a unique character lexicographically smaller than all other characters in Σ. Then the* i-th cyclic shift *of T is $T_{i \ldots n} T_{1 \ldots i-1}$. We denote it by $T^{(i)}$.*

Example 1.

$$\begin{array}{l} \scriptstyle 1\,2\,3\,4\,5\,6\,7\,\,8\,9\,10 \\ T = \text{CACAACCAC\$} \end{array}$$

$$T^{(6)} = \text{CCAC\$CACAA}$$

The *Burrows-Wheeler-Transformation* (BWT) is obtained by the following steps:

1. Write all cyclic shifts $T^{(i)}$, $1 \leq i \leq n$, column-wise next to each other.
2. Sort the columns lexicographically.
3. Output the last row. This is T^{BWT}.

Example 2.

$T = \text{CACAACCAC\$}$

$$\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & A & C & A & A & C & C & A & C & \$ \\
A & C & A & A & C & C & A & C & \$ & C \\
C & A & A & C & C & A & C & \$ & C & A \\
A & A & C & C & A & C & \$ & C & A & C \\
A & C & C & A & C & \$ & C & A & C & A \\
C & C & A & C & \$ & C & A & C & A & A \\
C & A & C & \$ & C & A & C & A & A & C \\
A & C & \$ & C & A & C & A & A & C & C \\
C & \$ & C & A & C & A & A & C & C & A \\
\$ & C & A & C & A & A & C & C & A & C \\
\uparrow & & & & & \uparrow & & & & \\
T^{(1)} & & & & & T^{(6)} & & & &
\end{array}
\quad \overset{\Rightarrow}{\begin{array}{c}\text{sort}\\ \text{columns}\\ \text{lexicogr.}\end{array}} \quad
\begin{array}{ccccccccccl}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \\
\$ & A & A & A & A & C & C & C & C & C & \longrightarrow F \text{ (first)} \\
C & A & C & C & C & \$ & A & A & A & C & \\
A & C & \$ & A & C & C & A & C & C & A & \\
C & C & C & A & A & A & C & \$ & A & C & \\
A & A & A & C & C & C & C & C & A & \$ & \\
A & C & C & C & \$ & A & A & A & C & C & \\
C & \$ & A & C & C & A & C & C & C & A & \\
C & C & A & A & A & C & \$ & A & A & C & \\
A & A & C & \$ & C & C & C & A & C & A & \\
C & C & C & C & A & A & A & C & \$ & A & \longrightarrow T^{\text{BWT}} = L \text{ (last)} \\
 & & & & & & & & \uparrow & & \\
 & & & & & & & & T^{(1)} & &
\end{array}$$

The text T^{BWT} in the last row is also denoted by L (last), and the text in the first row by F (first). Note:

- Every row in the BWT-matrix is a permutation of the characters in T.
- Row F is a sorted list of all characters in T.
- In row $L = T^{\text{BWT}}$, similar characters are grouped together. This is why T^{BWT}can be compressed more easily than T.

7.2 Construction of the BWT

The BWT-matrix needs *not* to be constructed *explicitly* in order to obtain T^{BWT}. Since T is terminated with the special character \$, which is lexicographically smaller than any $a \in \Sigma$, the shifts $T^{(i)}$ are sorted exactly like T's suffixes. Because the last row consists of the characters *preceding* the corresponding suffixes, we have

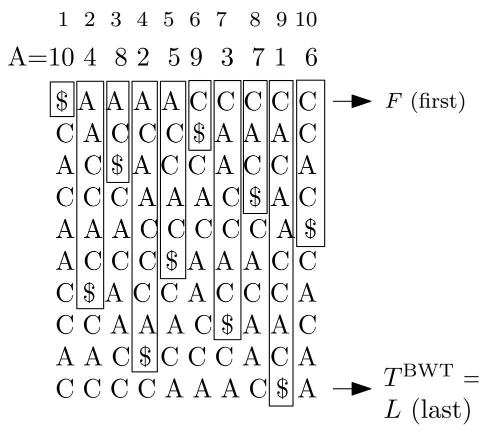
$$T^{\text{BWT}}[i] = t_{A[i]-1} (= T^{(A[i])}[n]) \ ,$$

where A denotes again T's suffix array, and t_0 is defined to be t_n (read T cyclically!). Because the suffix array can be constructed in linear time (Thm. 7), we get:

Theorem 22. *The BWT of a text length-n text over an integer alphabet can be constructed in $O(n)$ time.* $\square$

Example 3.

$T = \text{CACAACCAC\$}$



7.3 The Reverse Transformation

The amazing property of the BWT is that it is not a random permutation of T's letters, but that it can be *transformed back* to the original text T. For this, we need the following definition:

Definition 26. *Let F and L be the strings resulting from the* BWT*. Then the* last-to-front *mapping* LF *is a function* LF $: [1, n] \to [1, n]$*, defined by*

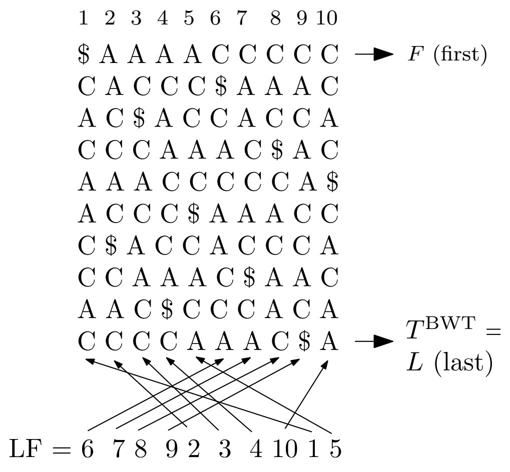
$$\text{LF}(i) = j \iff T^{(A[j])} = (T^{(A[i])})^{(n)} (\iff A[j] = A[i] + 1) \ .$$

(Remember that $T^{(A[i])}$ is the i'th column in the BWT*-matrix, and $(T^{(A[i])})^{(n)}$ is that column rotated by one character downwards.)*

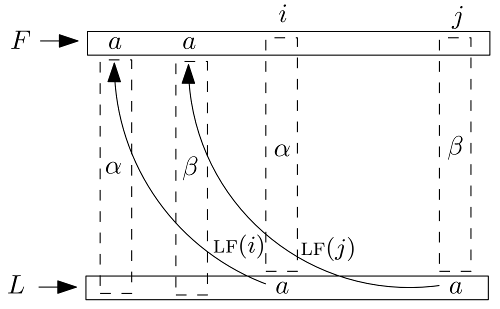
Thus, LF(i) tells us the position in F where $L[i]$ occurs.

Example 4.

$T = $ CACAACCAC\$



Observation 4. *Equal characters preserve the same order in F and L. That is, if $L[i] = L[j]$ and $i < j$, then $\textsc{lf}(i) < \textsc{lf}(j)$. To see why this is so, recall that the* BWT*-matrix is sorted lexicographically. Because both the $\textsc{lf}(i)$'th and the $\textsc{lf}(j)$'th column start with the same character $a = L[i] = L[j]$, they must be sorted according to what follows this character a, say α and β. But since $i < j$, we know $\alpha <_{lex} \beta$, hence $\textsc{lf}(i) < \textsc{lf}(j)$.*



This observation allows us to compute the LF-mapping *without knowing the suffix array* of T.

Definition 27. *Let T be a text of length n over an alphabet Σ, and let $L = T^{\text{BWT}}$ be its* BWT.

- *Define $C : \Sigma \to [1, n]$ such that $C(a)$ is the number of occurrences in T of characters that are lexicographically smaller than $a \in \Sigma$.*
- *Define $\textsc{occ} : \Sigma \times [1, n] \to [1, n]$ such that $\textsc{occ}(a, i)$ is the number of occurrences of a in L's length-i-prefix $L[1, i]$.*

Lemma 23. *With the definitions above,*

$$\textsc{lf}(i) = C(L[i]) + \textsc{occ}(L[i], i) \ .$$

Proof: Follows immediately from the observation above. □

This gives rise to the following algorithm to recover T from $L = T^{\text{BWT}}$.

1. Scan $L = T^{\text{BWT}}$ and compute array $C[1, \sigma]$.
2. Compute the first row F from C; as F consists of all characters in L sorted lexicographically, this step is trivial.
3. Compute $\text{OCC}(L[i], i)$ for all $1 \le i \le n$.
4. Recover $T = t_1 t_2 \dots t_n$ *from right to left*: we know that $t_n = \$$, and the corresponding cyclic shift $T^{(n)}$ appears in column 1 in BWT. Hence, $t_{n-1} = L[1]$. Shift $T^{(n-1)}$ appears in column $\text{LF}(1)$, and thus $t_{n-2} = L[\text{LF}(1)]$. This continues until the whole text has been recovered:

$$t_{n-i} = L[\underbrace{\text{LF}(\text{LF}(\dots(\text{LF}(1))\dots))}_{i-1 \text{ applications of LF}}]$$

Example 5.

$$C = \begin{array}{ccc} \scriptstyle\$ & \scriptstyle A & \scriptstyle C \\ 0 & 1 & 5 \end{array}$$

$$F = \$\ \mathrm{A\ A\ A\ A\ C\ C\ C\ C\ C}$$

$$L = \mathrm{C\ C\ C\ C\ A\ A\ A\ C}\ \$\ \mathrm{A}$$

$$\text{OCC}(L[i], i) = 1\ 2\ 3\ 4\ 1\ 2\ 3\ 5\ 1\ 4$$

$$\begin{array}{r|c|l} T_n = & \$ & , k = 1 \\ L[1] = \mathrm{C} \Rightarrow T_{n-1} = & \mathrm{C} & , k = \mathrm{LF}(1) = 6 \\ L[6] = \mathrm{C} \Rightarrow T_{n-2} = & \mathrm{A} & , k = \mathrm{LF}(6) = 3 \\ L[3] = \mathrm{C} \Rightarrow T_{n-3} = & \mathrm{C} & , k = \mathrm{LF}(3) = 8 \\ L[8] = \mathrm{C} \Rightarrow T_{n-4} = & \mathrm{C} & , k = \mathrm{LF}(8) = 10 \\ L[10] \text{ etc.} & & \\ \hline \end{array}$$

$$\uparrow$$

T reversed

8 Backwards Search and *FM*-Indices

We are now going to explore how the BW-transformed text is helpful for (indexed) pattern matching. Indices building on the BWT are called *FM*-indices, most likely in honor of their inventors P. Ferragina and G. Manzini. From now on, we shall always assume that the alphabet Σ is good-natured: $\sigma = o(n / \log \sigma)$.

Recommended Reading

- G. Navarro and V. Mäkinen: *Compressed Full-Text Indexes.* ACM Computing Surveys **39**(1), Article no. 2 (61 pages), 2007. Sect. 4.1, 4.2, 5.1, 5.4, 6.1, and 9.1.

8.1 Model of Computation and Space Measurement

For the rest of this lecture, we work with the *word-RAM* model of computation. This means that we have a processor with registers of *width* w (usually $w = 32$ or $w = 64$), where usual arithmetic operations (additions, shifts, comparisons, etc.) on w-bit wide words can be computed in constant time. Note that this matches all current computer architectures. We further assume that n, the input size, satisfies $n \leq 2^w$, for otherwise we could not even address the whole input.

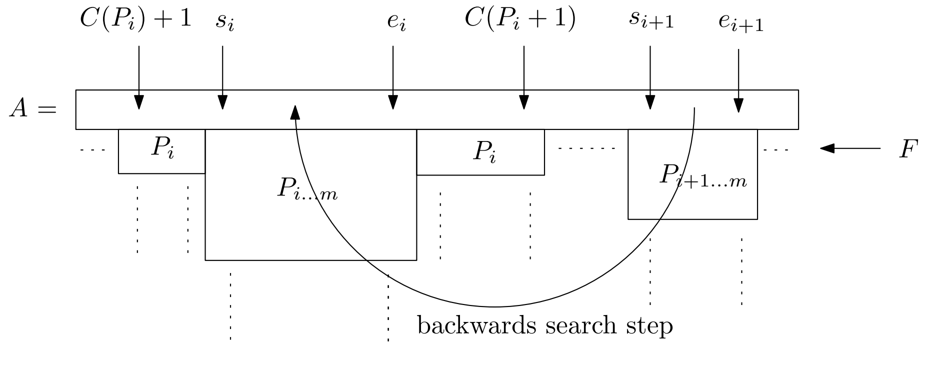
From now on, we measure the space of all data structures in *bits* instead of words, in order to be able to differentiate between the various text indexes. For example, an array of n numbers from the range $[1, n]$ occupies $n\lceil \log n \rceil$ bits, as each array cell stores a binary number consisting of $\lceil \log n \rceil$ bits. As another example, a length-n text over an alphabet of size σ occupies $n\lceil \log \sigma \rceil$ bits. In this light, all text indexes we have seen so far (suffix trees, suffix arrays, suffix trays) occupy $O(n \log n + n \log \sigma)$ bits. Note that the difference between $\log n$ and $\log \sigma$ can be quite large, e. g., for the human genome with $\sigma = 4$ and $n = 3.4 \times 10^9$ we have $\log \sigma = 2$, whereas $\log n \approx 32$. So the suffix array occupies about 16 times more memory than the genome itself!

8.2 Backward Search

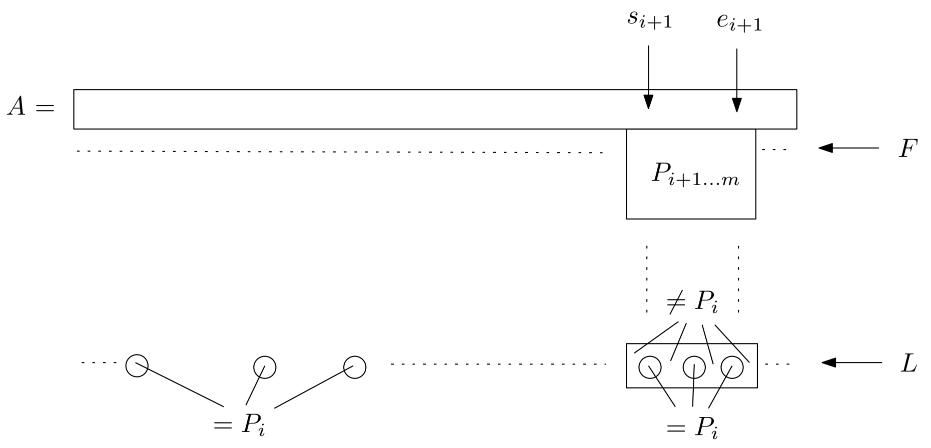
We first focus our attention on the *counting problem* (p. 3); i.e., on finding the number of occurrences of a pattern $P_{1\dots m}$ in $T_{1\dots n}$. Recall from Chapter 7 that

- A denotes T's suffix array.
- L/F denotes the first/last row of the BWT-matrix.
- LF$(\cdot)$ denotes the last-to-front mapping.
- $C(a)$ denotes the number of occurrences in T of characters lexicographically smaller than $a \in \Sigma$.
- OCC(a, i) denotes the number of occurrences of a in $L[1, i]$.

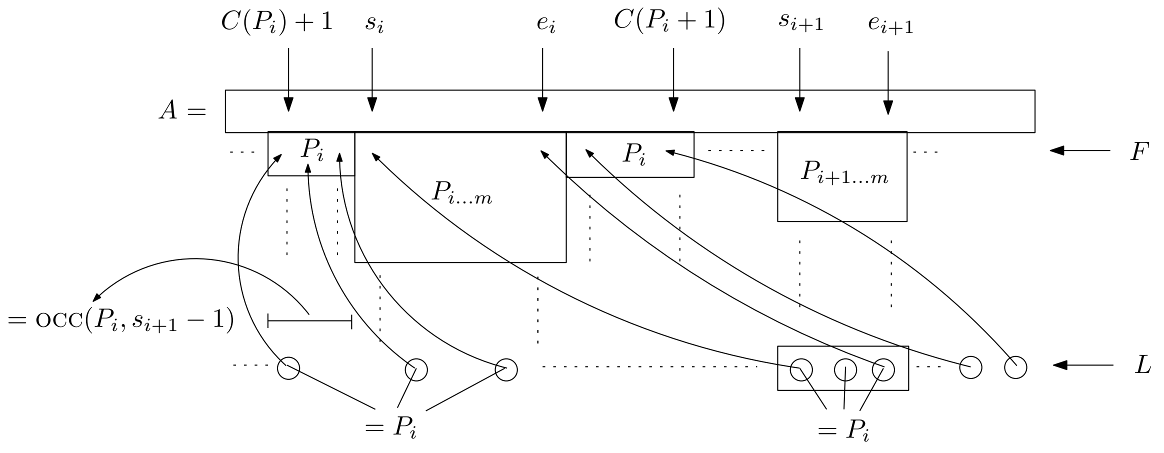
Our aim is identify the interval of P in A by searching P from right to left (= backwards). To this end, suppose we have already matched $P_{i+1\dots m}$, and know that the suffixes starting with $P_{i+1\dots m}$ form the interval $[s_{i+1}, e_{i+1}]$ in A. In a *backwards search step*, we wish to calculate the interval $[s_i, e_i]$ of $P_{i\dots m}$. First note that $[s_i, e_i]$ must be a sub-interval of $[C(P_i) + 1, C(P_i + 1)]$, where $(P_i + 1)$ denotes the character that follows P_i in Σ.



So we need to identify, from those suffixes starting with P_i, those which continue with $P_{i+1\ldots m}$. Looking at row L in the range from s_{i+1} to e_{i+1}, we see that there are exactly $e_i - s_i + 1$ many positions $j \in [s_{i+1}, e_{i+1}]$ where $L[j] = P_i$.



From the BWT decompression algorithm, we know that characters preserve the same order in F and L. Hence, if there are x occurrences of P_i before s_{i+1} in L, then s_i will start x positions behind $C(P_i) + 1$. This x is given by $\textsc{occ}(P_i, s_{i+1} - 1)$. Likewise, if there are y occurrences of P_i within $L[s_{i+1}, e_{i+1}]$, then $e_i = s_i + y - 1$. Again, y can be computed from the $\textsc{occ}$-function.



This gives rise to the following, elegant algorithm for backwards search:

The reader should compare this to the "normal" binary search algorithm in suffix arrays. Apart from matching backwards, there are two other notable deviations:

1. The suffix array A is not accessed during the search.
2. There is no need to access the input text T.

Hence, T and A can be deleted once T^{BWT} has been computed. It remains to show how array C and $\textsc{occ}$ are implemented. Array C is actually very small and can be stored plainly using $\sigma \log n$ bits.[1] Because $\sigma = o(n/\log n)$, $|C| = o(n)$ bits. For $\textsc{occ}$, we have several options that are explored

[1]More precisely, we should say $\sigma\lceil\log n\rceil$ bits, but we will usually omit floors and ceilings from now on.

Algorithm 4: function `backwards-search`($P_{1 \ldots m}$)

```
$s \leftarrow 1$; $e \leftarrow n$;
for $i = m \ldots 1$ do
    $s \leftarrow C(P_i) + \textsc{occ}(P_i, s-1) + 1$;
    $e \leftarrow C(P_i) + \textsc{occ}(P_i, e)$;
    if $s > e$ then
        return "no match";
    end
end
return $[s, e]$;
```

in the rest of this chapter. This is where the different *FM*-Indices deviate from each other. In fact, we will see that there is a natural trade-off between time and space: using more space leads to a faster computation of the OCC-values, while using less space implies a higher query time.

Theorem 24. *With backwards search, we can solve the counting problem in $O(m \cdot t_{\textsc{occ}})$ time, where $t_{\textsc{occ}}$ denotes the time to answer an $\textsc{occ}(\cdot)$-query.*

8.3 First Ideas for Implementing Occ

For answering $\textsc{occ}(c, i)$, there are two simple possibilities:

1. Scan L every time an $\textsc{occ}(\cdot)$-query has to be answered. This occupies no space, but needs $O(n)$ time for answering a single $\textsc{occ}(\cdot)$-query, leading to a total query time of $O(mn)$ for backwards search.

2. Store all answers to $\textsc{occ}(c, i)$ in a two-dimensional table. This table occupies $O(n\sigma \log n)$ bits of space, but allows constant-time $\textsc{occ}(\cdot)$-queries. Total time for backwards search is *optimal* $O(m)$.

For more *more practical implementation* between these two extremes, let us define the following:

Definition 28. *Given a bit-vector $B[1, n]$, $rank_1(B, i)$ counts the number of 1's in B's prefix $B[1, i]$. Operation $rank_0(B, i)$ is defined similarly for 0-bits.*

In the lecture "Advanced Data Structures" (every winter semester) it is shown that a bit-vector B, together with additional information for constant-time *rank*-operations, can be stored in $n + o(n)$ bits. This can be used as follows for implementing OCC: For each character $c \in \Sigma$, store an *indicator* bit vector $B_c[1, n]$ such that $B_c[i] = 1$ iff $L[i] = c$. Then

$$\textsc{occ}(c, i) = rank_1(B_c, i) \; .$$

The total space for all σ indicator bit vectors is thus $\sigma n + o(\sigma n)$ bits. Note that for *reporting* queries, we still need the suffix array to output the values in $A[s, e]$ after the backwards search.

Theorem 25. *With backwards search and constant-time rank operations on bit-vectors, we can answer counting queries in optimal $O(m)$ time. The space (in bits) is $\sigma n + o(\sigma n) + \sigma \log n$.* $\square$

Example 6.

$$\begin{array}{l} \quad\ \ {\scriptstyle 1\,2\,3\,4\,5\ \,6\,7\,8\,9\,10} \\ L = \text{CCCCAAAC\$A} \end{array}$$

$$\begin{aligned} B_{\$} &= 0000000010 \\ B_A &= 0000111001 \\ B_C &= 1111000100 \end{aligned}$$

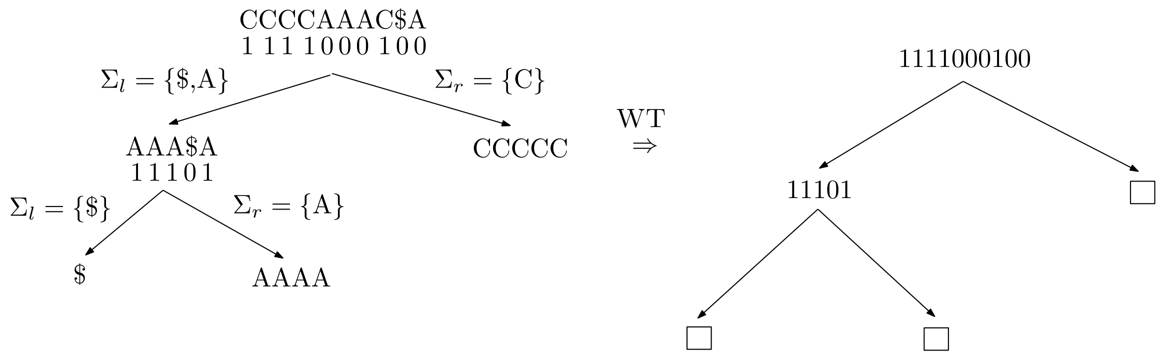
8.4 Wavelet Trees

Armed with constant-time *rank*-queries, we now develop a more space-efficient implementation of the OCC-function, sacrificing the optimal query time. The idea is to use a *wavelet tree* on the BW-transformed text.

The wavelet tree of a sequence $L[1, n]$ over an alphabet $\Sigma[1, \sigma]$ is a balanced binary search tree of height $O(\log \sigma)$. It is obtained as follows. We create a root node v, where we divide Σ into two halves $\Sigma_l = \Sigma[1, \lceil \frac{\sigma}{2} \rceil]$ and $\Sigma_r = \Sigma[\lceil \frac{\sigma}{2} \rceil + 1, \sigma]$ of roughly equal size. Hence, Σ_l holds the lexicographically first half of characters of Σ, and Σ_r contains the other characters. At v we store a bit-vector B_v of length n (together with data structures for $O(1)$ *rank*-queries), where a $'0'$ of position i indicates that character $L[i]$ belongs to Σ_l, and a $'1'$ indicates the it belongs to Σ_r. This defines two (virtual) sequences L_v and R_v, where L_v is obtained from L by concatenating all characters $L[i]$ where $B_v[i] = 0$, in the order as they appear in L. Sequence R_v is obtained in a similar manner for positions i with $B_v[i] = 1$. The left child l_v is recursively defined to be the root of the wavelet tree for L_v, and the right child r_v to be the root of the wavelet tree for R_v. This process continues until a sequence consists of only one symbol, in which case we create a leaf.

Example 7.

L=CCCCAAAC\$A $\quad \Sigma$ ={\$,A,C}



Note that the sequences themselves are *not* stored explicitly; node v only stores a bit-vector B_v and structures for $O(1)$ *rank*-queries.

Theorem 26. *The wavelet tree for a sequence of length n over an alphabet of size σ can be stored in $n \log \sigma \times (1 + o(1))$ bits.*

Proof: We concatenate all bit-vectors at the same depth d into a single bit-vector B_d of length n, and prepare it for $O(1)$-*rank*-queries. Hence, at any level, the space needed is $n + o(n)$ bits.

Because the depth of the tree is $\lceil \log \sigma \rceil$ the claim on the space follows. In order to "know" the sub-interval of a particular node v in the concatenated bit-vector B_d at level d, we can store two indices α_v and β_v such that $B_d[\alpha_v, \beta_v]$ is the bit-vector B_v associated to node v. This accounts for additional $O(\sigma \log n)$ bits. Then a rank-query is answered as follows ($b \in \{0,1\}$):

$$rank_b(B_v, i) = rank_b(B_d, \alpha_v + i - 1) - rank_b(B_d, \alpha_v - 1) \ ,$$

where it is assumed that $i \leq \beta_v - \alpha_v + 1$, for otherwise the result is not defined. □

How does the wavelet tree help for implementing the OCC-function? Suppose we want to compute OCC(c, i), i. e., the number of occurrences of $c \in \Sigma$ in $L[1, i]$. We start at the root r of the wavelet tree, and check if c belongs to the first or to the second half of the alphabet. In the first case, we know that the c's are "stored" in the *left* child of the root, namely L_r. Hence, the number of c's in $L[1, i]$ corresponds to the number of c's in $L_r[1, rank_0(B_r, i)]$. If, on the hand, c belongs to the second half of the alphabet, we know that the c's are "stored" in the subsequence R_r that corresponds to the *right* child of r, and hence compute the number of occurrences of c in $R_r[1, rank_1(B_r, i)]$ as the number of c's in $L[1, i]$. This leads to the following recursive procedure for computing OCC(c, i), to be invoked with WT-OCC$(c, i, 1, \sigma, r)$, where r is the root of the wavelet tree. (Recall that we assume that the characters in Σ can be accessed as $\Sigma[1], \ldots, \Sigma[\sigma]$.)

Algorithm 5: function `WT-occ`$(c, i, \sigma_l, \sigma_r, v)$

```
if $\sigma_l = \sigma_r$ then
    return $i$;
end
$\sigma_m = \lfloor \frac{\sigma_l + \sigma_r}{2} \rfloor$;
if $c \leq \Sigma[\sigma_m]$ then
    return WT-occ$(c, rank_0(B_v, i), \sigma_l, \sigma_m, l_v)$;
else
    return WT-occ$(c, rank_1(B_v, i), \sigma_m + 1, \sigma_r, r_v)$;
end
```

Due to the depth of the wavelet tree, the time for `WT-occ`$(\cdot)$ is $O(\log \sigma)$. This leads to the following theorem.

Theorem 27. *With backward-search and a wavelet-tree on T^{BWT}, we can answer counting queries in $O(m \log \sigma)$ time. The space (in bits) is*

$$\underbrace{O(\sigma \log n)}_{|C| + \text{ space for } \alpha_v\text{'s}} + \underbrace{n \log \sigma}_{\text{wavelet tree}} + \underbrace{o(n \log \sigma)}_{\textit{rank} \text{ data structure}} \ .$$

8.5 Sampling the Suffix Array

If we also want to solve the *reporting problem* (outputting all starting positions of P in T, see p. 3), we *do* need the actual suffix array values. A simple way to solve this is to sample regular text positions in A, and use the LF-function to recover unsampled values. More precisely, we choose a sampling parameter s, and in an array A' we write the values $1, s, 2s, 3s, \ldots$ in the order as they appear in the full suffix array A. Array A' takes $O(n/s \log n)$ bits. In a bit-vector S of length

n, we mark the sampled suffix array values with a '1', and augment S with constant-time rank information. Now let i be a position for which we want to find the value of $A[i]$. We first check if $S[i] = 1$, and if so, return the value $A'[rank_1(S, i)]$. If not ($S[i] = 0$), we go to position $\textsc{lf}(i)$ in time $t_{\textsc{lf}}$, making use of the fact that if $A[i] = j$, then $A[\textsc{lf}(i)] = j - 1$. This processes continues until we hit a sampled position d, which takes at most s steps. We then add the number of times we followed $\textsc{lf}$ to the sampled value of $A'[d]$; the result is $A[i]$. The overall time for this process is $O(s \cdot t_{\textsc{occ}})$ for a single suffix array value. Choosing $s = \log_\sigma n$ and wavelet trees for implementing the $\textsc{occ}$-function, we get an index of $O(n \log \sigma)$ space, $O(m \log \sigma)$ counting time, and $O(k \log n)$ reporting time for k occurrences to be reported.

9 Simulation of Suffix Trees

So far, we have seen compressed text indices that have only one functionality: locating all occurrences of a search pattern P in a text T. In some cases, however, more functionality is required. From other courses you might know that many sequence-related problems are solved efficiently with *suffix trees* (e. g., computing tandem repeats, MUMs, ...). However, the space requirement of a suffix tree is huge: it is at least 20–40 times higher then the space of the text itself, using very proprietary implementations that support only a very small number of all conceivable suffix tree operations. In this chapter, we present a generic approach that allows for the simulation of *all* suffix tree operations, by using only compressed data structures. More specifically, we will build on the compressed suffix array from Chapter 8, and show how all suffix tree operations can be simulated by computations on *suffix array intervals* (the same intervals that we used for suffix trays). Space-efficient data structures that facilitate these computations will be handled in subsequent chapters.

Recommended Reading

- J. Fischer, V. Mäkinen, and G. Navarro: *Faster Entropy-Bounded Compressed Suffix Tree.* Theor. Comput. Sci. **410**(51), 5354–5364, 2009.

9.1 Basic Concepts

The reader is encouraged to recall the definitions from Sect. 2.1, in particular Def. 4. From now on, we regard the suffix tree as an abstract data type that supports the following operations.

Definition 29. *A* suffix tree S *supports the following operations.*

- $\textsc{Root}()$*: returns the* root *of the suffix tree.*
- $\textsc{IsLeaf}(v)$*: true iff v is a* leaf*.*
- $\textsc{LeafLabel}(v)$*: returns $l(v)$ if v is a leaf, and* NULL *otherwise.*
- $\textsc{IsAncestor}(v, w)$*: true iff v is an* ancestor *of w.*
- $\textsc{SDepth}(v)$*: returns $d(v)$, the* string-depth *of v.*
- $\textsc{Count}(v)$*: the number of leaves in S_v.*

- $\textsc{Parent}(v)$*: the* parent *node of* v*.*
- $\textsc{FirstChild}(v)$*: the alphabetically* first child *of* v*.*
- $\textsc{NextSibling}(v)$*: the alphabetically* next sibling *of* v*.*
- $\textsc{lca}(v)$*: the* lowest common ancestor *of* v *and* w*.*
- $\textsc{Child}(v, a)$*: node* w *such that the edge-label of* (v, w) *starts with* $a \in \Sigma$*.*
- $\textsc{EdgeLabel}(v, i)$ *the* i*'th letter on the edge* $(\textsc{Parent}(v), v)$*.*

We recall from from previous chapters that A denotes the suffix array, H the LCP-array, and RMQ a range minimum query. Because we will later be using *compressed* data structures (which not necessarily have constant access times), we use variables t_{SA}, t_{LCP} and t_{RMQ} for the access time to the corresponding array/function. E. g., with uncompressed (plain) arrays, we have $t_{\text{SA}} = t_{\text{LCP}} = t_{\text{RMQ}} = O(1)$, while with the sampled suffix array from Sect. 8.5 we have $t_{\text{SA}} = O(\log n)$.

We represent a suffix tree node v by the *interval* $[v_\ell, v_r]$ such that $A[v_\ell], \ldots, A[v_r]$ are exactly the labels of the leaves below v. For such a representation we have the following basic lemma (from now on we assume $H[1] = H[n+1] = -1$ for an easy handling of border cases):

Lemma 28. *Let* $[v_\ell, v_r]$ *be the interval of an internal node* v*. Then*

(1) For all $k \in [v_\ell + 1, v_r] : H[k] \geq d(v)$*.*

(2) $H[v_\ell] < d(v)$ *and* $H[v_r + 1] < d(v)$*.*

(3) There is a $k \in [v_\ell + 1, v_r]$ *with* $H[k] = d(v)$*.*

Proof: Condition (1) follows because all suffixes $T^{A[k]}$, $k \in [v_\ell, v_r]$, have $\overline{v}$ as their prefix, and hence $H[k] = \textsc{lcp}(T^{A[k]}, T^{A[k-1]}) \geq |\overline{v}| = d(v)$ for all $k \in [v_\ell + 1, v_r]$. Property (2) follows because otherwise suffix $T^{A[v_\ell]}$ or $T^{A[v_r+1]}$ would start with $\overline{v}$, and hence leaves labeled $A[v_\ell]$ or $A[v_r + 1]$ would also be below v. For proving property (3), for the sake of contradiction assume $H[k] > d(v)$ for all $k \in [v_\ell + 1, v_r]$. Then all suffixes $T^{A[k]}$, $k \in [v_\ell, v_r]$, would start with $\overline{v}a$ for some $a \in \Sigma$. Hence, v would only have one outgoing edge (whose label starts with a), contradicting the fact that the suffix tree is compact (has no unary nodes). $\square$

As a side remark, this is actually an "if and only if" statement, as every interval satisfying the three conditions from Lemma 28 corresponds to an internal node.

Definition 30. *Let* $[v_\ell, v_r]$ *be the interval of an internal node* v*. Any position* $k \in [v_\ell + 1, v_r]$ *satisfying point (3) in Lemma 28 is called a* $d(v)$-index *of* v*.*

Our aim is to simulate all suffix tree operations by computations on suffix intervals: given the interval $[v_\ell, v_r]$ corresponding to node v, compute the interval of $w = f(v)$ from the values v_ℓ and v_r alone, where f can be any function from Def. 29; e.g., $f = \textsc{Parent}$. We will see that most suffix tree operations follow a generic approach: first locate a $d(w)$-index p of w, and then search for the (yet unknown) delimiting points w_ℓ and w_r of w's suffix interval. For this latter task (computation of w_ℓ and w_r from p), we also need the previous- and next-smaller-value functions as already defined in Def. 23 in Sect. 6.1. However, this time we define them to work on the LCP-array:

Definition 31. *Given the* LCP*-array* H *and an index* $1 \leq i \leq n$*, the* previous smaller value *function* $PSV_H(i) = \max\{k < i : H[k] < H[i]\}$*. The* next smaller value *function* $NSV_H(i)$ *is defined similarly for* succeeding *positions:* $NSV_H(i) = \min\{k > i : H[k] < H[i]\}$*.*

We use t_{PNSV} to denote the time to compute a value $NSV_H(i)$ or $PSV_H(i)$. In what follows, we often use simply PSV and NSV instead of PSV_H and NSV_H, implicitly assuming that array H is the underlying array. The following lemma shows how these two functions can be used to compute the delimiting points w_ℓ and w_r of w's suffix interval:

Lemma 29. *Let* p *be a* $d(w)$*-index of an internal node* w*. Then* $w_\ell = PSV(p)$*, and* $w_r = NSV(p)-1$*.*

Proof: Let $l = PSV(p)$, and $r = NSV(p)$. We must show that all three conditions in Lemma 29 are satisfied by $[l, r-1]$. Because $H[l] < H[p]$ by the definition of PSV, and likewise $H[r] < H[p]$, point (1) is clear. Further, because l and r are the *closest* positions where H attains a smaller value, condition (2) is also satisfied. Point (3) follows from the assumption that p is a $d(w)$-index. We thus conclude that $w_\ell = l$ and $w_r = r - 1$. □

9.2 Suffix Tree Operations

We now step through the operations from Def. 29 and show how they can be simulated by computations on the suffix array intervals. Let $[v_\ell, v_r]$ denote the interval of an arbitrary node v. The most easy operations are:

- Root(): returns the interval $[1, n]$.
- IsLeaf(v): true iff $v_\ell = v_r$.
- Count(v): returns $v_r - v_\ell + 1$.
- IsAncestor(v, w): true iff $v_\ell \leq w_r \leq v_r$.

Time is $O(1)$ for all four operations.

- LeafLabel(v): If $v_\ell \neq v_r$, return NULL. Otherwise, return $A[v_\ell]$ in $O(t_{\text{SA}})$ time.
- SDepth(v): If $v_\ell = v_r$, return $n - A[v_\ell] + 1$ in time $O(t_{\text{SA}})$, as this is the length of the $A[v_\ell]$'th suffix. Otherwise from Lemma 28 we know that $d(v)$ is the minimum LCP-value in $H[v_\ell + 1, v_r]$. We hence return $H[\text{RMQ}_H(v_\ell + 1, v_r)]$ in time $O(t_{\text{RMQ}} + t_{\text{LCP}})$.
- Parent(v): Because S is a compact tree, either $H[v_\ell]$ or $H[v_r + 1]$ equals the string-depth of the parent-node, whichever is greater. Hence, we first set $p = \text{argmax}\{H[k] : k \in \{v_\ell, v_r+1\}\}$, and then, by Lemma 29, return $[PSV(p), NSV(p) - 1]$. Time is $O(t_{\text{LCP}} + t_{\text{PNSV}})$.
- FirstChild(v): If v is a leaf, return NULL. Otherwise, locate the first $d(v)$-value in $H[v_\ell, v_r]$ by $p = \text{RMQ}_H(v_\ell + 1, v_r)$. Here, we assume that RMQ returns the position of the *leftmost* minimum, if it is not unique. The final result is $[v_\ell, p - 1]$, and the total time is $O(t_{\text{RMQ}})$.
- NextSibling(v): First, compute v's parent as $w = \text{Parent}(v)$. Now, if $v_r = w_r$, return NULL, since v does not have a next sibling in this case. If $w_r = v_r + 1$, then v's next sibling is a leaf, so we return $[w_r, w_r]$. Otherwise, try to locate the first $d(w)$-value after $v_r + 1$ by $p = \text{RMQ}_H(v_r + 2, w_r)$. If $H[p] = d(w)$, we return $[v_r + 1, p - 1]$ as the final result. Otherwise ($H[p] > d(w)$), the final result is $[v_r + 1, w_r]$. Time is $O(t_{\text{LCP}} + t_{\text{PNSV}} + t_{\text{RMQ}})$.

- LCA(v,w): First check if one of v or w is an ancestor of the other, and return that node in this case. Otherwise, assume $v_r < w_\ell$ (otherwise swap v and w). Let u denote the (yet unknown) LCA of v and w, so that our task is to compute u_ℓ and u_r. First note that all suffixes $T^{A[k]}$, $k \in [v_\ell, v_r] \cup [w_\ell, w_r]$, must be prefixed by $\overline{u}$, and that u is the deepest node with this property. Further, because none of v and w is an ancestor of the other, v and w must be contained in subtrees rooted at two *different* children $\hat{u}$ and $\grave{u}$ of u, say v is in $\hat{u}$'s subtree and w in the one of $\grave{u}$. Because $v_r \leq w_\ell$, we have $\hat{u}_r \leq \grave{u}_\ell$, and hence there must be a $d(u)$-index in H between $\hat{u}_r$ and $\grave{u}_\ell$, which can be found by $p = \text{RMQ}_H(v_r+1, w_\ell)$. The endpoints of u's interval are again located by $u_\ell = PSV(p)$ and $u_r = NSV(p) - 1$. Time is $O(t_{\text{RMQ}} + t_{\text{PNSV}})$.

- EdgeLabel(v,i): First, compute the string-depth of v by $d_1 = \text{SDepth}(v)$, and that of $u = \text{Parent}(v)$ by $d_2 = \text{SDepth}(u)$, in total time $O(t_{\text{RMQ}} + t_{\text{LCP}} + t_{\text{PNSV}})$. Now if $i > d_1 - d_2$, return NULL, because i exceeds the length of the label of (u,v) in this case. Otherwise, the result is given by $t_{A[v_\ell]+d_2+i-1}$, since the edge-label of (u,v) is $T_{A[k]+d_2 \dots A[k]+d_1-1}$ for an arbitrary $k \in [v_\ell, v_r]$. Total time is thus $O(t_{\text{SA}} + t_{\text{RMQ}} + t_{\text{LCP}} + t_{\text{PNSV}})$.

A final remark is that we can also simulate many other operations in suffix trees not listed here, e.g. suffix links, Weiner links, level ancestor queries, and many more.

9.3 Compressed LCP-Arrays

We now show how to reduce the space for the LCP-array H from $n \log n$ to $O(n)$ bits. To this end, we first note that the LCP-value can decrease by at most 1 when moving from suffix $A[i]-1$ to $A[i]$ in H (i. e., when enumerating the LCP-values in *text* order):

Lemma 30. *For all* $1 < i \leq n$, $H[i] \geq H[A^{-1}[A[i]-1] - 1$.

Proof: If $H[i] = 0$, the claim is trivial. Hence, suppose $H[i] > 0$, and look at the two suffixes starting at positions $A[i]$ and $A[i-1]$, which must start with the same character. Suppose $T^{A[i]} = a\alpha$ and $T^{A[i-1]} = a\beta$ for $a \in \Sigma$, $\alpha, \beta \in \Sigma^*$.

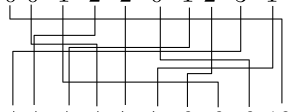
Because the suffixes are sorted lexicographically in A, and $a\alpha >_{\text{lex}} a\beta$, we know $\alpha >_{\text{lex}} \beta$, and that α and β share a common prefix of length $H[i]-1$, call it γ. Now note that all suffixes between β and α in A must also start with γ, as otherwise the suffixes would not be in lexicographic order. In particular, suffix $T^{A[A^{-1}[A[i]+1]-1]}$ must be prefixed by γ, and hence $H[A^{-1}[A[i]+1] = \text{LCP}(T^{A[i]+1}, T^{A[A^{-1}[A[i]+1]-1]}) = \text{LCP}(\alpha, T^{A[A^{-1}[A[i]+1]-1]}) \geq |\gamma| = H[i] - 1$. □

From the above lemma, we can conclude that $I[1,n] = [H[A^{-1}[1]]+1, H[A^{-1}[2]]+2, H[A^{-1}[3]]+3, \dots, H[A^{-1}[n]]+n]$ is an array of *increasing* integers. Further, because no LCP-value can exceed the length of corresponding suffixes, we see that $H[A^{-1}[i]] \leq n-i+1$. Hence, sequence I must be in range $[1,n]$. We encode I *differentially*: writing $\Delta[i] = I[i] - I[i-1]$ for the difference between entry i and $i-1$, and defining $I[0] = 0$ for handling the border case, we encode $\Delta[i]$ in *unary* as $0^{\Delta[i]}1$. Let the resulting sequence be S.

$T =$ C A C A A C C A C \$

$A =$ 10 4 8 2 5 9 3 7 1 6

$H =$ 0 0 1 2 2 0 1 2 3 1



$I =$ 4 4 4 4 7 7 9 9 9 10

$S =$ 00001 1 1 1 0001 1 001 1 1 01

Note that the number of 1's in S is exactly n, and that the number of 0's is at most n, as the $\Delta[i]$'s sum up to at most n. Hence, the length of S is at most $2n$ bits. We further prepare S for constant-time *rank*$_0$- and *select*$_1$-queries, using additional $o(n)$ bits. Then $H[i]$ can be retrieved by

$$H[i] = \mathit{rank}_0(S, \mathit{select}_1(S, A[i])) - A[i] \ .$$

This is because the *select*-statement points to the position of the terminating '1' of $0^{\Delta[A[i]]}1$ in S, and the *rank*-statement counts the sum of Δ-values before that position, which is $I[A[i]]$. From this, in order to get $H[i]$, we need to subtract $A[i]$, which has bin "artificially" added when deriving I from H.

By noting that there are exactly $A[i]$ 1's up to position $\mathit{select}_1(S, A[i])$ in S (and therefore $\mathit{select}_1(S, A[i]) - A[i]$ 0's), the calculation can be further simplified to

$$H[i] = \mathit{select}_1(S, A[i]) - 2A[i] \ .$$

We have proved:

Theorem 31. *The* LCP*-array H can be stored in $2n + o(n)$ bits such that retrieving an arbitrary entry $H[i]$ takes $t_{\text{LCP}} = O(t_{\text{SA}})$ time.*

Note that with the sampled suffix array from Sect. 8.5, this means that we no more have constant-time access to H, as $t_{\text{SA}} = O(\log n)$ in this case.

10 Succinct Data Structures for RMQs and PSV/NSV Queries

This chapter shows that $O(n)$ bits are sufficient to answer RMQs and *PSV*/*NSV*-queries in constant time. For our compressed suffix tree, we assume that all three queries are executed on the LCP-array H, although the data structures presented in this chapter are applicable to any array of ordered objects.

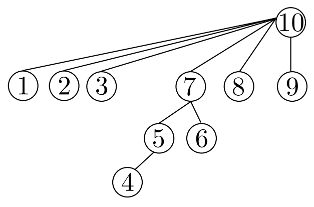
10.1 2-Dimensional Min-Heaps

We first define a tree that will be the basis for answering RMQs and *NSV*-queries. The solution for *PSV*-queries is symmetric. The following definition assumes that $H[n + 1]$ is always the smallest value in H, what can be enforced by introducing a "dummy" element $H[n + 1] = -\infty$.

Definition 32. *Let $H[1, n + 1]$ be an array of totally ordered objects, with the property that $H[n + 1] < H[i]$ for all $1 \leq i \leq n$. The* 2-dimensional Min-Heap *$\mathcal{M}_H$ of H is a tree an n nodes $1, \ldots, n$, defined such that $NSV(i)$ is the parent-node of i for $1 \leq i \leq n$.*

Note that $\mathcal{M}_H$ is a well-defined tree whose root is $n+1$.

Example 8.

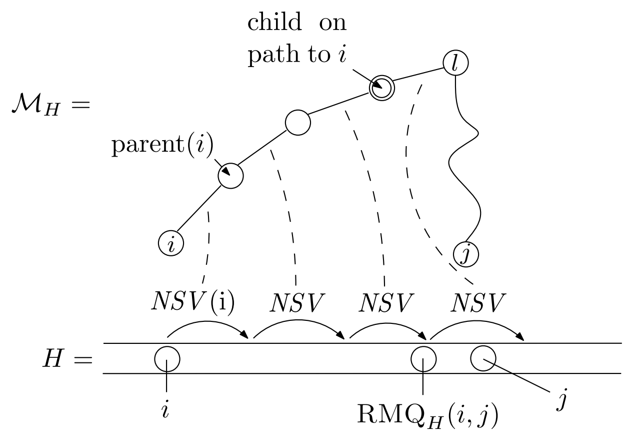


$$H = \underset{1}{\text{-1}}\ \underset{2}{0}\ \underset{3}{0}\ \underset{4}{3}\ \underset{5}{1}\ \underset{6}{2}\ \underset{7}{0}\ \underset{8}{1}\ \underset{9}{1}\ \underset{10}{-\infty}$$

From the definition of $\mathcal{M}_H$, it is immediately clear that the value $NSV(i)$ is given by the parent node of i ($1 \leq i \leq n$). The next lemma shows that $\mathcal{M}_H$ is also useful for answering RMQs on H.

Lemma 32. *For $1 \leq i < j \leq n$, let $l = \text{LCA}_{\mathcal{M}_H}(i,j)$. Then if $l = j$, $\text{RMQ}_H(i,j) = j$. Otherwise, $\text{RMQ}_H(i,j)$ is given by the child of l that is on the path from l to i.*

Proof: "graphical proof":



Example 9. *Continuing the example above, let $i = 4$ and $j = 6$. We have $\text{LCA}_{\mathcal{M}_H}(4,6) = 7$, and 5 is the child of 7 on the path to 4. Hence, $\text{RMQ}_H(4,6) = 5$.*

10.2 Balanced Parentheses Representation of Trees

Any ordered tree T on n nodes can be represented by a sequence B of $2n$ parentheses as follows: in a depth-first traversal of T, write an opening parenthesis '(' when visiting a node v for the first time, and a closing parenthesis ')' when visiting v for the last time (i. e., when all nodes in T_v have been traversed).

Example 10. *Building on the 2d-Min-Heap from the Example 8, we have $B = (()()()((())())()())$.*

In a computer, a '(' could be represented by a '1'-bit, and a ')' by a '0'-bit, so the space for B is $2n$ bits. In the lecture "Advanced Data Structures" it is shown that this representation allows us to answer queries like $rank_{(}(B,i)$ and $select_{)}(B,i)$, by using only $o(n)$ additional space.

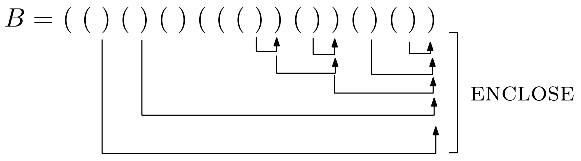
Note that the sequence B is *balanced*, in the sense that in each prefix the number of closing parentheses is no more than the number of opening parenthesis, and that there are n opening and closing parentheses each in total. Hence, this representation of trees is called *balanced parentheses sequence* (BPS).

We also need the following operation.

Definition 33. *Given a sequence $B[1, 2n]$ of balanced parentheses and a position i with $B[i] =$ ')', enclose(B, i) returns the position of the closing parenthesis of the* nearest enclosing '()'-pair.

In other words, if v is a node with closing parenthesis at position $i < 2n$ in B, and w is the parent of v with closing parenthesis at position j in B, then $enclose(B, i) = j$. Note that $enclose(i) > i$ for all i, because of the order in which nodes are visited in a depth first traversal.

Example 11.



We state the following theorem that is also shown in the lecture "Advanced Data Structures."

Theorem 33. *There is a data structure of size $O\left(\frac{n \log\log n}{\log n}\right) = o(n)$ bits that allows for constant-time enclose-queries.*

(The techniques are roughly similar to the techniques for *rank*- and *select*-queries.)

Now look at an arbitrary position i in B, $1 \le i \le 2n$. We define the *excess-value* $E[i]$ at position i as the number of opening parenthesis in $B[1, i]$ minus the number of closing parenthesis in $B[1, i]$. Note that the excess-values do not have to be stored explicitly, as

$$\begin{aligned}|E[i]| &= rank_{(}(B, i) - rank_{)}(B, i)\\ &= i - rank_{)}(B, i) - rank_{)}(B, i)\\ &= i - 2rank_{)}(B, i)\ .\end{aligned}$$

Example 12.

$$\begin{array}{rcccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20\\ B = & (& (&) & (&) & (&) & (& (& (&) &) & (&) &) & (&) & (&) &)\\ E = & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 4 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 0\end{array}$$

Note:

1. $E[i] > 0$ for all $1 \le i < 2n$
2. $E[2n] = 0$
3. If i is the position of the closing parenthesis of node v, then $E[i]$ is the *depth* of v. (Counting starts at 0, so the root has depth 0.)

We also state the following theorem without proof.

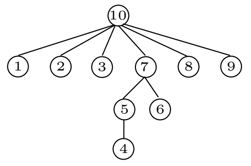
Theorem 34. *Given a sequence B of balanced parentheses, there is a data structure of size $O\left(\frac{n \log\log n}{\log n}\right) = o(n)$ bits that allows to answer* RMQ*s on the associated excess-sequence E in constant time.* $\square$

(The techniques are again similar to *rank* and *select*: *blocking* and *table-lookups*. Note in particular that $\frac{\log n}{2}$ excess-values $E[x], E[x+1], \ldots, E\left[x + \frac{\log n}{2} - 1\right]$ are encoded in a *single* computer-word $B\left[x, x + \frac{\log n}{2} - 1\right]$, and hence it is again possible to apply the Four-Russians-Trick!)

10.3 Answering Queries

We represent $\mathcal{M}_H$ by its BPS B, and identify each node i in $\mathcal{M}_H$ by the position of its *closing* parenthesis in B.

Example 13.

$\mathcal{M}_H=$ 

$$\begin{array}{rcccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
B = & (& (&) & (&) & (&) & (& (& (&) &) & (&) &) & (&) & (&) &) \\
 & & & \textcircled{1} & & \textcircled{2} & & \textcircled{3} & & & & \textcircled{4} & \textcircled{5} & & \textcircled{6} & \textcircled{7} & & \textcircled{8} & & \textcircled{9} & \textcircled{10} \\
E = & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 4 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 0
\end{array}$$

Note that the (closing parenthesis of) nodes appear in B in *sorted* order - this is simply because in $\mathcal{M}_H$ node i hast *post-order* number i, and the closing parenthesis appear in post-order by the definition of the BPS. This fact allows us to jump back and forth between indices in H and positions of closing parentheses ')' in B, by using rank- and select-queries in the appropriate sequences.

Answering *NSV-queries* is now simple. Suppose we wish to answer $NSV_H(i)$. We then move to the position of the i'th ')' by

$$x \leftarrow select_{)}(B, i) \ ,$$

and then call

$$y \leftarrow enclose(B, x)$$

in order to move to the position y of the closing parenthesis of the parent j of i in $\mathcal{M}_H$. The (yet unknown) value j is computed by

$$j \leftarrow rank_{)}(B, y) \ .$$

Example 14. *We want to compute $NSV(7)$. First compute $x \leftarrow select_{)}(B, 7) = 15$, and then $y \leftarrow enclose(15) = 20$. The final result is $j \leftarrow rank_{)}(B, 20) = 10$.*

Answering RMQ*s* is only slightly more complicated. Suppose we wish to answer $\text{RMQ}_H(i, j)$ for $1 \le i < j \le n$. As before, we go to the appropriate positions in B by

$$x \leftarrow select_{)}(B, i) \text{ and}$$

$$y \leftarrow select_{)}(B, j) \ .$$

We then compute the position of the minimum excess-value in the range $[x, y]$ by

$$z \leftarrow \text{RMQ}_E(x, y) \ ,$$

and map it back to a position in H by

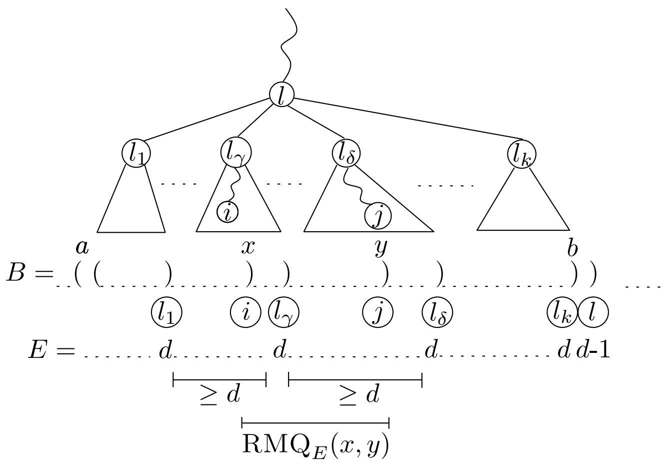
$$m \leftarrow \mathit{rank}_{)}(B, z) \ .$$

This is the final answer.

Example 15. *We want to compute* $\text{RMQ}_H(4, 9)$*. First, compute* $x \leftarrow \mathit{select}_{)}(B, 4) = 11$ *and* $y \leftarrow \mathit{enclose}(B, 9) = 19$*. The range minimum query yields* $z \leftarrow \text{RMQ}_E(11, 19) = 15$*. Finally,* $m \leftarrow \mathit{rank}_{)}(B, 15) = 7$ *is the result.*

We now justify the correctness of this approach. First assume that $\ell = \text{LCA}_{\mathcal{M}_H}(i, j)$ is different from j. Let $\ell_1, \ldots, \ell_k$ be the children of ℓ, and assume $i \in T_{\ell_\gamma}$ and $j \in T_{\ell_\delta}$ for some $1 \leq \gamma < \delta \leq k$. By Lemma 32, we thus need to show that the position of the closing parenthesis of ℓ_γ is the position where E attains the minimum in $E[x, y]$.

Example 16.



Let $d - 1$ be the tree-depth of ℓ, and let $B[a, b]$ denote the part of B that "spells out" T_ℓ (i.e., $B[a, b]$ is the BPS of the sub-tree of T rooted at ℓ). Note that $a < x < y < b$, as i and j are both below ℓ in T.

Because $B[a]$ is the opening parenthesis of node ℓ, we have $E[a] = d$. Further, because B is balanced, we have $E[c] \geq d$ for all $a < c < b$. But E assumes the values d at the positions of the closing parenthesis of nodes ℓ_β $(1 \leq \beta \leq k)$, in particular for ℓ_γ. Hence, the leftmost minimum in $E[x, y]$ is attained at the position z of the closing parenthesis of node ℓ_γ, which is computed by an RMQ in E. The case where $\ell = j$ is similar (and even simpler to prove). Thus, we get:

Theorem 35. *With a data structure of size* $2n + o(n)$ *bits, we can answer* RMQ*s and NSV-queries on an array of* n *ordered objects on* $O(1)$ *time.* □

The *drawback* of the 2d-Min-Heap, however, is that it is inherently asymmetric (as the parent-relationship is defined by the minimum to the *right*), and cannot be used for answering *PSV*-queries as well. For this, we could build another 2d-Min-Heap $\mathcal{M}_H^R$ on the *reversed* sequence H^R, using another $2n + o(n)$ bits. (Note that an interesting side-effect of this $\mathcal{M}_H^R$ is that it would allow to compute the *rightmost* minimum in any query range, instead of the leftmost, which could have interesting applications in compressed suffix trees.)

In the lecture we also discussed the possibility to just add another bit-vector of length n bits — however, this seems only to work if we represent the 2d-Min-Heap by DFUDS (instead of BPS). If we plug all these structures into the compressed suffix tree from Chapter 9 (which was indeed the reason for developing the solutions for RMQs and PNSVs), we get:

Theorem 36. *A suffix tree on a text of length n over an alphabet of size σ can be stored in $|SA| + 3n+o(n)$ bits of space (where $|SA|$ denotes the space for the suffix array), such that operations* Root*,* IsLeaf*,* Count*,* IsAncestor*,* FirstChild*, and* lca *take $O(1)$ time, and operations* LeafLabel*,* SDepth*,* Parent*,* NextSibling *and* EdgeLabel *take $O(t_{\text{SA}})$ time (where t_{SA} denotes the time to retrieve an element from the suffix array).* □

11 Inside Google*

Recommended Reading

- S. Muthukrishnan: *Efficient Algorithms for Document Retrieval Problems.* Proc. of the 13th Annual Symposium on Discrete Algorithms, 657–666. ACM/SIAM, 2002.

11.1 The Task

You are given a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of sequences $S_i \in \Sigma^*$ (web pages, protein or DNA-sequences, or the like). Your task is to build an index on $\mathcal{S}$ such that the following type of on-line queries can be answered *efficiently*:

given: a pattern $P \in \Sigma^*$.

return: all $j \in [1, m]$ such that S_j contains P.

Exercise: What has this to do with Google?

11.2 The Straight-Forward Solution

Define a string

$$T = S_1\#S_2\# \ldots \#S_m\#$$

of length $n := \sum_{1\le i\le m}(|S_i| + 1) = m + \sum_{1\le i\le m} |S_i|$. Build the suffix array A on T. In an array $D[1, n]$ remember from which string in $\mathcal{S}$ the corresponding suffix comes from:

$$D[i] = j \text{ iff } \sum_{k=1}^{j-1}(|S_k| + 1) < A[i] \le \sum_{k=1}^{j}(|S_k| + 1)\ .$$

When a query pattern P arrives, first locate the interval $[\ell, r]$ of P in A. Then output all numbers in $D[\ell, r]$, removing the duplicates (how?).

11.3 The Problem

Even if we can efficiently remove the duplicates, the above query algorithm is *not* output sensitive. To see why, consider the situation where P occurs many (say x) times in S_1, but never in S_j for $j > 1$. Then the query takes $O(|P| + x)$ time, just to output *one* sequence identifier (namely nr. 1). Note that x can be as large as $\Theta(n)$, e.g., if $|S_1| \geq \frac{n}{2}$.

11.4 An Optimal Solution

The following algorithm solves the queries in optimal $O(|P| + d)$ time, where d denotes the number of sequences in $\mathcal{S}$ where P occurs.

We set up a new array $E[1, n]$ such that $E[i]$ points to the nearest previous occurrence of $D[i]$ in D:

$$E[i] = \begin{cases} j & \text{if there is a } j < i \text{ with } D[j] = D[i], \text{ and } D[k] \neq D[i] \text{ for all } j < k < i \ , \\ -1 & \text{if no such } j \text{ exists.} \end{cases}$$

It is easy to compute E with a single left-to-right scan of D. We further process E for constant-time RMQs.

When a query pattern P arrives, we first locate P's interval $[\ell, r]$ in A in $O(|P|)$ time (as before). We then call `report`(ℓ, r), which is a procedure defined as follows.

Algorithm 6: Document Reporting

```
procedure report $(i, j)$;
    $m \leftarrow \text{RMQ}_E(i, j)$;
    if $E[m] \leq \ell$ then
        output $D[m]$;
        if $m - 1 \geq i$ then report$(i, m - 1)$;
        if $m + 1 \leq j$ then report$(m + 1, j)$;
end
```

The claimed $O(d)$ running time of the call to `report`(ℓ, r) relies on the following observation. Consider the range $[\ell, r]$. Note that P is a prefix of $T^{A[i]}$ for all $\ell \leq i \leq r$. The idea is that the algorithm visits and outputs only those suffixes $T^{A[i]}$ with $i \in [\ell, r]$ such that the corresponding suffix σ_i of $S_{D[i]}$ ($\sigma_i = T^{A[i]\dots e}$, where $e = \sum_{1 \leq j \leq D[i]}(|S_j| + 1)$ is the end position of $S_{D[j]}$ in T) is the *lexicographically smallest* among those suffixes of $S_{D[i]}$ that are prefixed by P. Because the suffix array orders the suffixes lexicographically, we must have $E[i] \leq \ell$ for such suffixes σ_i. Further, there is at most one such position i in $[\ell, r]$ for each string S_j. Because the recursion searches the whole range $[\ell, r]$ for such positions i, no string $S_j \in \mathcal{S}$ is missed by the procedure.

Finally, when the recursion stops (i.e., $E[m] > \ell$), because $E[m]$ is the minimum in $E[i, j]$, we must have that the identifiers of the strings $S_{D[k]}$ for all $k \in [i, j]$ have already been output in a previous call to `report`(i', j') for some $\ell \leq i' \leq j' < i$. Hence, we can safely stop the recursion at this point.