Algorithmische Grundlagen und Vermittlung der Informatik
Lehrstuhl 11 (Algorithm Engineering)
Prof. Dr. Johannes Fischer

technische universität
dortmund

Vorläufiges Skriptum VL
Text-Indexierung und Information Retrieval

Wintersemester 2016/17
last update: December 6, 2016

Disclaimer 1

Dieses Skript wird den Studierenden an der TU Dortmund im Voraus zur Verfügung gestellt. Die Inhalte werden im Laufe des Semesters aber noch angepasst. Die horizontale Linie kennzeichnet den bisher tatsächlich behandelten Stoff. Mit (*) markierte Abschnitte wurden in diesem Semester nicht in der Vorlesung behandelt (können aber durchaus prüfungsrelevant sein, z.B. wenn Teile daraus in den Übungen behandelt wurden).

Disclaimer 2

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 this script. In this sense, it is **not a good idea** to study the subject soley by reading this script.

1 Recommended Reading

More or less in the order of relevance for this lecture:

1. V. Mäkinen, D. Belazzougui, F. Cunial, A. Tomescu: *Genome-Scale Algorithm Design.* Cambridge University Press, 2015.

2. G. Navarro: *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

3. E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

4. D. Gusfield: *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

5. R. Baeza-Yates and B. Ribeiro-Neto: *Modern information retrieval* (2nd edition). Person Education Limited, 2011.

6. M. Crochemore, C. Hancart, T. LeCroq: *Algorithms on Strings.* Cambridge University Press, 2001.

7. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching.* Springer, 2008.

2 Tries

Definition 1. *Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a set of k prefix-free strings (meaning that no string is a prefix of another string) over the alphabet Σ of size $\sigma = |\Sigma|$. A* trie *on S is a rooted tree $S = (V, E)$ with edge labels from Σ that fulfills the following two constraints:*

1. *$\forall v \in V$: all outgoing edges from v start with a different $a \in \Sigma$.*

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

3. *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 often deal with *compacted* tries, which can be defined similarly to Def. 1, with the difference that the edge labels are now from Σ^+, and with an additional constraint:

4. *Apart from the root, all nodes have out-degree $\neq 1$.*

Tries support existential queries ("Is pattern P one of the strings in $\mathcal{S}$?"), prefix queries ("Which strings in $\mathcal{S}$ have P as a prefix?"), and also predecessor queries ("If P is none of the strings in $\mathcal{S}$, which ones are lexicographically closest?"). All of those queries work in a top-down manner, starting at the root and trying to match further characters in P on the way down. The search time of all these operations depends mainly on the way the outgoing edges of a trie node are implemented; this is what we consider next.

Let v be a node in the trie.

1. We can simply scan all of v's outgoing edges to find the next character of P. This results in $O(|P| \cdot \sigma)$ search time. The space of the trie is $O(n+k) = O(n)$ for $n = \sum_{i=1}^{k} |s_i|$ being the total size of the strings in $\mathcal{S}$.

2. The outgoing edges are implemented as arrays of size σ. This results in optimal $O(|P|)$ search time, but the space shoots up to $O(n \cdot \sigma)$.

3. We can use either a hash table at every node, or a global hash table using (node,character) pairs as keys. In any case, this results in optimal $O(|P|)$ search time, but only with high probability. Also, predecessor searches are not supported. The space is $O(n)$.

4. The outgoing edges are implemented as arrays of size s_v, where s_v denotes the number of v's children. Using binary search over these arrays, this results in total $O(|P| \log \sigma)$ search time. The overall space is $O(n)$ (WHY?). Note that if the trie is dynamic, the arrays can be replaced by balanced binary search trees, yielding the same running times.

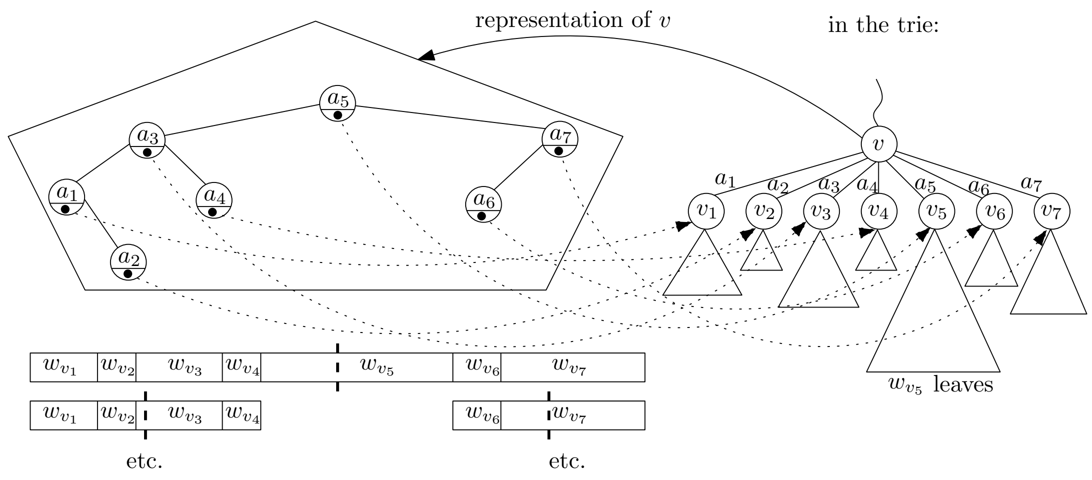

representation of v
in the trie:
a_5
a_3
a_7
a_1
a_4
a_6
a_2
v
a_1 a_2 a_3 a_4 a_5 a_6 a_7
v_1 v_2 v_3 v_4 v_5 v_6 v_7
w_{v_1} w_{v_2} w_{v_3} w_{v_4} w_{v_5} w_{v_6} w_{v_7}
w_{v_1} w_{v_2} w_{v_3} w_{v_4} w_{v_6} w_{v_7}
etc.
etc.
w_{v_5} leaves

Figure 1: Representation of trie nodes with weight-balanced binary search trees.

5. Modifying the previous approach, we can use weight-balanced binary search trees (WB-BST), where each trie node v has a weight w_v equal to the number of leaves below v (hence, the number of strings stored in v's subtree). Then the binary search tree at every trie node v with children $v_1, \ldots, v_x$ is formed as follows (see also Fig. 1). Split the total weights $w_{v_1}, \ldots, w_{v_x}$ exactly in the middle (namely at $\sum w_{v_i}/2$), respecting the lexicographic order of the corresponding characters. This creates the root of the WB-BST (the character touching this middle). The process continues recursively in the left and right children of the root. It is then easy to see that one character comparison in any WB-BST either advances one character in P, or reduces the number of strings to be considered by at least 1/2. Since the latter situation can happen only $\log k$ times, this results in a total search time of $O(|P| + \log k)$, while the space remains linear.

6. Here comes the climax! Divide the trie into an upper top tree and several lower bottom trees by declaring all maximally deep nodes with weight at least σ as leaves of the top tree. Then use approach (5) for the nodes in the bottom trees; since their size is now $O(\sigma)$, this results in $O(|P| + \log \sigma)$ time. In the top tree, all branching nodes (meaning thay have at least 2 children) are handled by approach (2) above. Since the number of branching nodes in the top tree are at most $O(n/\sigma)$, this results in $O(n)$ total space for the entire trie. Non-branching nodes of the top tree are simply stored by noting the character of their only outgoing edge. In sum, we get $O(|P| + \log \sigma)$ time, and $O(n)$ space.

3 Suffix Trees and Arrays

In this section we will introduce suffix trees and suffix arrays, 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 and arrays over the other string-matching algorithms (Boyer-Moore, KMP, etc.) is that those structures 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 time proportional only to m, and only weakly depends on n ("weakly" meaning, for example, logarithmically). If $m \ll n$, this is a clear advantage over the other algorithms.

Throughout this section, let $T = t_1 t_2 \dots t_n$ be a text over an alphabet Σ of size σ. 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.

To make this more formal, let P be a pattern of length m. 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* $O_P = \{i \in [1, n] : T_{i \dots i+m-1} = P\}$

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

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

3.1 Suffix- and LCP-Arrays

Definition 3. *The* suffix array *A of T is a permutation of $\{1, 2, \dots, n\}$ such that $A[i]$ is the i-th smallest suffix in lexicographic order: $T^{A[i-1]} < T^{A[i]}$ for all $1 < i \leq n$.*

Hence, the suffix array is a compact representation ($O(n)$ space) of the sorted order of all suffixes of a text.

The second array H builds on the suffix array:

Definition 4. *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 $T^{A[i-1]}$.*

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 $a \in \Sigma$.

3.2 Construction of Suffix Arrays

The first task we consider is how to construct the suffix array; i.e., how to actually sort the suffixes of a text $T_{1 \dots n}$ lexicographically. We want to do this quicker than in $O(n^2 \lg n)$ time, which is what we would achieve by employing a comparison-based sorting algorithm like merge sort.

3.2.1 $O(n \lg n)$-Time Construction

The idea of a simple $O(n \lg n)$-time algorithm is to repeatedly bucket-sort the suffixes in an MSB-first-like fashion, in step j using the characters $T_{i+2^{j-1} \dots i+2^j-1}$ as the sort key for suffix T^i. Such a radix step is possible in $O(n)$ time, since by induction, all suffixes are already sorted by their first 2^{j-1} characters. This algorithm is often called *prefix doubling*. We did examples and pseudo-code in the lecture; more details can also be found in:

- V. Heun: *Skriptum zur Vorlesung Algorithmen auf Sequenzen.* LMU München, 2016. Chapter 5.1.3. Available at `https://www.bio.ifi.lmu.de/mitarbeiter/volker-heun/notes/as5.pdf`.

3.2.2 Linear-Time Construction

Now we explain the *induced sorting* algorithm for constructing suffix arrays (called *SAIS* in the literature). 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).
- E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013. Chapter 4.1.2.

Definition 5. *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 6. *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 could be done by any string sorting algorithm, e.g., the trie sorter we have seen in the exercises. But it is also possible to do something 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 1. *We can construct the suffix array for a text of length n in $O(n)$ time.*

For proving the correctness of this algorithm, we identified the following *key lemma* in the lecture:

Lemma 2. *If two suffixes T^i and T^j start with the same character, then their lexicographic order is determined by the lexicographic order of the suffixes T^{i+1} and T^{j+1}.*

(Note that for a full proof it remains to show that all suffixes are actually considered by the algorithm – see the cited literature for details.)

3.3 Linear-Time Construction of LCP-Arrays

- E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013. Chapters 4.2.1.

It remains to be shown how the LCP-array H can be constructed in $O(n)$ time. Here, we assume that we are given T and A, the text and the suffix array for T.

We will construct H *in text order*, which is also the order of the *inverse suffix array* A^{-1}, the latter defined by $A^{-1}[A[i]] = i$ for all $1 \le i \le n$, which is easily computable from A in linear time. In other words, we aim at filling $H[A^{-1}[i]]$ before $H[A^{-1}[i+1]]$, because in this case we know that H cannot decrease too much, as shown next.

Going from suffix T^i to T^{i+1}, we see that the latter equals the former, but with the first character t_i truncated. Let $h = H[i]$. Then the suffix T^j, $j = A[A^{-1}[i] - 1]$, has a longest common prefix with T^i of length h. So T^{i+1} has a longest common prefix with T^{j+1} of length $h - 1$. But every suffix T^k that is lexicographically between T^{j+1} and T^{i+1} must have a longest common prefix with T^{j+1} that is at least $h-1$ characters long (for otherwise T^k would not be in lexicographic order). In particular, the suffix right before T^{i+1} in A, which is suffix $T^{A[A^{-1}[i+1]-1]}$, must share a common prefix with S_{i+1} of length at least $h-1$. Hence, $H[A^{-1}[i+1]] \ge h - 1$. We have thus proved the following:

Lemma 3. *For all $1 \le i < n$: $H[A^{-1}[i+1]] \ge H[A^{-1}[i]] - 1$.*

This gives rise to the following elegant algorithm to construct H:

Algorithm 1: Linear-Time Construction of the LCP-Array

```
for $i = 1, \ldots, n$ do $A^{-1}[A[i]] \leftarrow i$;
$h \leftarrow 0, H[1] \leftarrow 0$;
for $i = 1, \ldots, n$ do
    if $A^{-1}[i] \neq 1$ then
        $j \leftarrow A[A^{-1}[i] - 1]$;
        while $t_{i+h} = t_{j+h}$ do $h \leftarrow h + 1$;
        $H[A^{-1}[i]] \leftarrow h$;
        $h \leftarrow \max\{0, h - 1\}$;
    end
end
```

The linear running time follows because h starts and ends at 0, is always less than n and decreased at most n times in line 8. Hence, the number of times where k is increased in line 6 is bounded by n, so there are at most $2n$ character comparisons in the whole algorithm. We have proved:

Theorem 4. *We can construct the LCP array for a text of length n in $O(n)$ time.*

3.4 Suffix Trees

Definition 7. *The* suffix tree *of T is a compact trie over all suffixes $\{T^1, T^2, \ldots, T^n\}$.*

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

Definition 8. *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."]

It is useful if each suffix ends in a leaf of S. This can again be accomplished by adding a new character $\$ \notin \Sigma$ to the end of T, and build the suffix tree over $T\$$. 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: $l(v) = i \iff \overline{v} = T^i$.

The following observations relate the suffix array A and the LCP-array H 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.*

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

Definition 9. *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: $\forall i > 1 : H[i] = d(\text{LCA}(\overline{T^{A[i]}}, \overline{T^{A[i-1]}}))$.*

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 5. *A suffix tree of a text of length n occupies $O(n)$ space in memory.*

3.5 Searching in Suffix Trees

Since the suffix tree is a trie, we can use any of the methods from the section on tries (Sect. 2) for navigation: for example, the *counting* the number of pattern matches can be done in $O(m \log \sigma)$ time (with outgoing-edge representation (4) from the previous chapter on tries): 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. With the more sophisticated representation (6) from Sect. 2 this time can be reduced to $O(m + \log \sigma)$.

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 *reporting* all positions where the pattern matches, we output the labels of all leaves in S_v (recall that the leaves are labeled with text positions).

Theorem 6. *The suffix tree can answer counting queries in $O(m + \log \sigma)$ time, and reporting queries in $O(m + \log \sigma + |O_P|)$ time.*

3.6 Linear-Time Construction of Suffix Trees

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 *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 $T_{A[i]+d(v)\dots A[i]+H[i+1]-1}$.
3. Add (y, w) and label it by $T_{A[i]+H[i+1]\dots A[i]+d(w)-1}$.
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 7. *We can construct T's suffix tree in linear time from T's suffix- and LCP-array.*

3.6.1 Practical Improvements*

Let us do some algorithm engineering on the LCP-array construction algorithm! The problem with this algorithm is its poor locality behavior, resulting in many *potential cache misses* ($4n$ in total). Our idea is now to rearrange the computations such that in the big for-loop accesses only one array in a random access manner, whereas all other arrays are scanned sequentially. To this end, we first compute a temporary array $\Phi[1, n]$ that at $\Phi[i]$ stores the lexicographic preceeding suffix of $T^{[}i]$. (This is exactly the suffix with whom we have to compare $T^{[}i]$ for longest common prefix computation.) Further, in the for-loop we write the computed LCP-values in *text order.* (This is exactly the order in which they are computed.) The resulting algorithm can be seen in Alg. 2.

Algorithm 2: More Cache-Efficient Linear-Time Construction of the LCP-Array

```
1  $\Phi[n] \leftarrow A[n]$;                    // assume that $T$ is $-terminated, so $A[1] = n$
2  for $i = 2, \dots, n$ do $\Phi[A[i]] \leftarrow A[i-1]$;
   // "with whom I want to be compared"
3  $h \leftarrow 0$;
4  for $i = 1, \dots, n$ do
5      $j \leftarrow \Phi[i]$;
6      while $t_{i+h} = t_{j+h}$ do $h \leftarrow h + 1$;
7      $H'[i] \leftarrow h$// $\Phi[i]$ can be overwritten by $H'$ (saves space)
8      $h \leftarrow \max\{0, h - 1\}$;
9  end
10 for $i = 1, \dots, n$ do $H[i] \leftarrow H'[A[i]]$;
   // put values back into suffix array order
```

In total, the algorithm now produces at most $3n$ cache misses (as opposed to $4n$ in Alg. 1). The practical running time of Alg. 2 is reported to be 1.5 times faster than Alg. 1.

3.7 Searching in Suffix Arrays

3.7.1 Exact Searches

- G. Navarro, V. Mäkinen: *Compressed Full-Text Indexes.* ACM Computing Surveys **39**(1), Article Article No. 2, 2007. Section 3.3.
- E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013. Chapter 5.1.3.

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 3: function `SAsearch`($P_{1\ldots m}$)

```
1  $l \leftarrow 1$; $r \leftarrow n + 1$;
2  while $l < r$ do
3      $q \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
4      if $P >_{\text{lex}} T_{A[q]\ldots\min\{A[q]+m-1,n\}}$ then
5          $l \leftarrow q + 1$;
6      else
7          $r \leftarrow q$;
8      end
9  end
10 $s \leftarrow l$; $l{-}{-}$; $r \leftarrow n$;
11 while $l < r$ do
12     $q \leftarrow \lceil \frac{l+r}{2} \rceil$;
13     if $P =_{\text{lex}} T_{A[q]\ldots\min\{A[q]+m-1,n\}}$ then
14         $l \leftarrow q$;
15     else
16         $r \leftarrow q - 1$;
17     end
18 end
19 return $[s, r]$;
```

Note that both while-loops in Alg. 3 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.

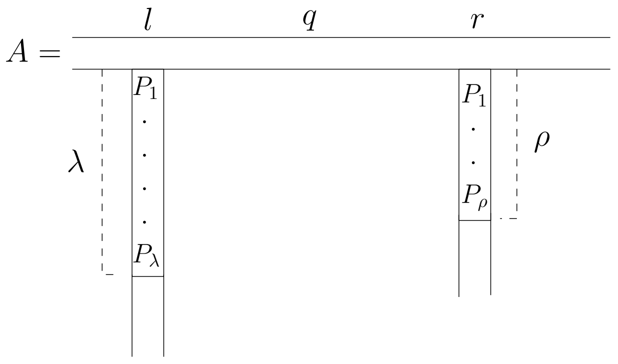
Theorem 8. *The suffix array allows to answer counting queries in $O(m \log n)$ time, and reporting queries in $O(m \log n + |O_P|)$ time.*

3.7.2 Accelerated Search in Suffix Arrays*

The simple binary search (Alg. 3) may perform many unnecessary character comparisons, as in every step it compares P from scratch. With the help of the LCP-function from the previous section, we can improve the search in suffix arrays from $O(m \log n)$ to $O(m + \log n)$ time. The idea is to *remember the number of matching characters* of P with $T_{A[l]\ldots n}$ and $T_{A[r]\ldots n}$, if $[l : r]$ denotes the current interval of the binary search procedure. Let λ and ρ denote these numbers,

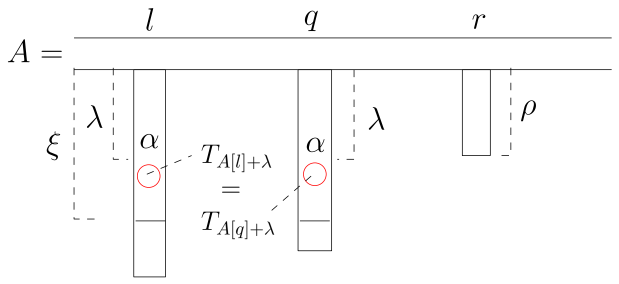
$$\lambda = \text{LCP}(P, T_{A[l]\ldots n}) \text{ and } \rho = \text{LCP}(P, T_{A[r]\ldots n}).$$

Initially, both λ and ρ are 0. Let us consider an iteration of the first while-loop in function $SAsearch(P)$, where we wish to determine whether to continue in $[l : q]$ or $[q, r]$. (Alg. 3 would actually continue searching in $[q + 1, r]$ in the second case, but this minor improvement is not possible in the accelerated search.) We are in the following situation:



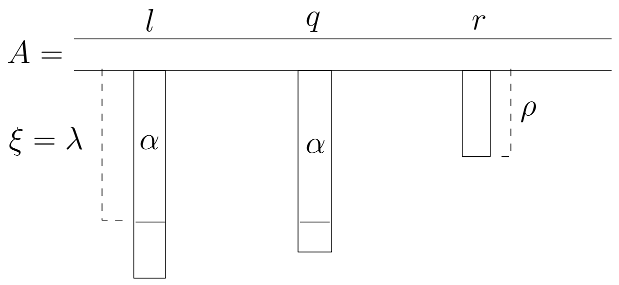
Without loss of generality, assume $\lambda \geq \rho$ (otherwise swap). We then look up $\xi = \text{LCP}(A[l], A[q])$ as the longest common prefix of the suffixes $T_{A[l]\ldots n}$ and $T_{A[q]\ldots n}$. We look at three different cases:

1. $\xi > \lambda$



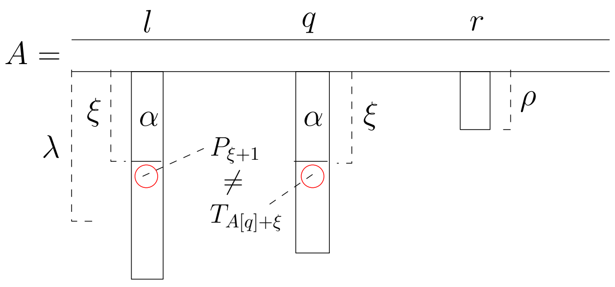
Because $P_{\lambda+1} >_{\text{lex}} T_{A[l]+\lambda} = T_{A[q]+\lambda}$, we know that $P >_{\text{lex}} T_{A[q]\ldots n}$, and can hence set $l \leftarrow q$, and continue the search *without any character comparison*. Note that ρ and in particular λ correctly remain unchanged.

2. $\xi = \lambda$



In this case we continue comparing $P_{\lambda+1}$ with $T_{A[q]+\lambda}$, $P_{\lambda+2}$ with $T_{A[q]+\lambda+1}$, and so on, until P is matched completely, or a mismatch occurs. Say we have done this comparison up to $P_{\lambda+k}$. If $P_{\lambda+k} >_{\text{lex}} T_{A[q]+\lambda+k-1}$, we set $l \leftarrow q$ and $\lambda \leftarrow k-1$. Otherwise, we set $r \leftarrow q$ and $\rho \leftarrow k-1$.

3. $\xi < \lambda$



First note that $\xi \geq \rho$, as $\text{LCP}(A[l], A[r]) \geq \rho$, and $T_{A[q]\dots n}$ lies lexicographically between $T_{A[l]\dots n}$ and $T_{A[r]\dots n}$. So we can set $r \leftarrow q$ and $\rho \leftarrow \xi$, and continue the binary search *without any character comparison.*

This algorithm either halves the search interval (case 1 and 3) without any character comparison, or increases either λ or ρ for each successful character comparison. Because neither λ nor ρ are ever decreased, and the search stops when $\lambda = \rho = m$, we see that the total number of character comparisons (= total work of case 2) is $O(m)$. So far we have proved the following theorem:

Theorem 9. *Together with* LCP*-information, the suffix array supports counting and reporting queries in $O(m + \log n)$ and $O(m + \log n + |O_P|)$ time, respectively (recall that O_P is the set of occurrences of P in T).*

4 Lempel-Ziv Compression

4.1 LZ77

Definition 10. *Given a text $T = t_1 \dots t_n$, its* LZ77-decomposition *is defined as a sequence of z strings $f_1, \dots, f_z$, $f_i \in \Sigma^+$ for all i, such that $T = f_1 f_2 \dots f_z$, and f_i is either a single letter not occurring in $f_1 \dots f_{i-1}$, or the longest factor occurring at least twice in $f_1 f_2 \dots f_i$.*

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

We usually describe the LZ77-factorization by a list of z pairs of integers $(\ell_1, p_1), \dots, (\ell_z, p_z)$ such that either (if $\ell_i > 0$) $f_i = T_{p_i \dots p_i+\ell_i-1}$, or (if $\ell_i = 0$) p_i is the (ASCII-)code of the single character f_i. In the former case, a pair (ℓ_i, p_i) can also be interpreted as “copy ℓ_i characters from position p_i.”

Using the suffix tree of T with the fast trie-implementation (approach (6) from Sect. 2), we could compute the factorization greedily in $O(\sum_{i=1}^{z}(|f_i| + \log\sigma)) = O(n + z\log\sigma)$ time. Next, we show how to improve this to optimal $O(n)$ time.

The algorithm consists of the following steps:

1. Construct the suffix tree S for T and annotate every internal node v with its string depth $d(v)$.

Algorithm 4: $O(n)$-computation of the LZ77-factorization for $T = t_1 \dots t_n$

```
$j \leftarrow 1, p \leftarrow 1$;   // $j$: position in $T$; $p$: position where next factor begins
while $p < n + 1$ do
    $v \leftarrow \tilde{A}^{-1}[j]$;                         // go to leaf labeled $j$
    while $v$ is unvisited do
        mark $v$ as visited;
        $v \leftarrow S.parent(v)$;                           // climb up
    end
    if $j = p$ then
        if $v = S.root()$ then
            output new LZ77-factor $(0, t_p)$;               // case 2: new character
            $p \leftarrow p + 1$;
        end
        else
            output new LZ77-factor $(d(v), s(v))$;     // case 1: ``copying'' factor
            $p \leftarrow p + d(v)$;
        end
    end
    $++j$;
end
```

2. Also, annotate every internal node v with $s(v)$, where $s(v)$ is the smallest leaf label in v's subtree.
3. Construct an array $\tilde{A}^{-1}$ such that $\tilde{A}^{-1}[j]$ points to the leaf labeled j (similar to the inverse suffix array).
4. Mark the root as `visited`, mark all other nodes as `unvisited`.
5. Call Alg. 4.

To analyse the running time, note that step 1 takes $O(n)$ time (see Sect. 3.4). Using a bottom-up approach, step 2 also takes $O(n)$ time. Steps 3 and 4 are trivially implemented in linear time. Finally, in step 5, each node is `visited` exactly once, and the inner while-loop stops when hitting the first `unvisited`'ed node, so the whole Alg. 4 runs in $O(n)$ time.

4.2 LZ78

Definition 11. *Given a text $T = t_1 \dots t_n$, its* LZ78-decomposition *is defined as a sequence of z strings $f_1, \dots, f_z$, $f_i \in \Sigma^+$ for all i, such that*

1. $T = f_1 f_2 \dots f_z$, and

2. if $f_1 \dots f_{i-1} = T_{1 \dots j-1}$, then f_i is the longest prefix of the suffix T^j such that $f_i = f_k a$ for some $k < i$ and $a \in \Sigma$ (define $f_0 = \epsilon$ to handle the "new character"-case).

We usually describe the LZ78-factorization as a sequence of z (int,char)-tuples $(k_1, a_1), \dots, (k_z, a_z)$, where the pair (k_i, a_i) is interpreted such that $f_i = f_{k_i} a_i$.

Algorithm 5: $O(n)$-computation of the LZ78-factorization for $T = t_1 \dots t_n$

```
$j \leftarrow 1$;                                    // $j$: position in $T$
while $j < n+1$ do
    $v \leftarrow \tilde{A}^{-1}[j]$;                 // go to leaf labeled $j$
    $u \leftarrow S.root()$;
    $x \leftarrow 0$;                                 // $x$ always holds the tree depth of $u$
    while $c(u) = e(u)$ do
        $++x$;
        $u \leftarrow \mathrm{LA}(v, x)$;             // climb down
    end
    $++c(u)$;          // increase counter aka append new factor to LZ78-trie
    $j \leftarrow d(S.parent(u)) + c(u)$;            // advance in text
end
```

A natural way to calculate the LZ78-factorization would be the use of a *trie*, the so-called *LZ78-trie*. If the text up to position $j-1$ has already been factorized into $f_1 f_2 \dots f_{i-1} = T_{1 \dots j-1}$, then the LZ78-trie stores all factors $f_1, \dots, f_{i-1}$. To find the next factor f_i, one descends greedily as far as possible in the trie to find the longest prefix of suffix T^j that already occurs in the trie (say as factor f_k for some $k < i$) and then extends the trie by a new leaf (whose incoming edge is labeled by the character $a \in \Sigma$ following this prefix in the text). Using approach 4 from Sect. 2 (the fastest for *dynamic* tries), this takes $O(\sum_{i=1}^{z}(|f_i| \log n)) = O(n \log n)$ time. Again, we want to improve this to $O(n)$ time.

The following linear-time algorithm superimposes the LZ78-trie on top of the suffix tree, circumventing the costly top-down navigation in the trie by employing another data structure. It consists of the following steps:

1. Construct the suffix tree S for T and annotate every node v with its string depth $d(v)$, and the number $e(v)$ of characters on v's incoming edge (note $d(v) = 0 = e(v)$ if $v = S.root()$).

2. Initialize a counter $c(v) = 0$ at every node v.

3. Construct an array $\tilde{A}^{-1}$ such that $\tilde{A}^{-1}[j]$ points to the leaf labeled j (similar to the inverse suffix array).

4. Compute a *level ancestor* data structure for S. Such a data structure prepares the tree S such that the following queries can later be answered in $O(1)$ time: given a node v and an integer $x \geq 0$, $\mathrm{LA}(v, x)$ returns the ancestor u of v at depth x (node depth, *not* string depth!). For example, $\mathrm{LA}(v, 0)$ returns the root node for every v, and if $v \neq$ *S.root()*, $\mathrm{LA}(v, 1)$ returns the child of the root on the path to v. See Sect. 5 for details.

5. Call Alg. 5.

The above algorithm does not output the factors as (int-char)-pairs (k_i, a_i), but we saw in the lecture that it is augment the algorithm to do exactly this.

5 Level Ancestor Queries (Deutsch, danke an Maximilian Schuler (KIT) für's texen!)

5.1 Literaturempfehlungen

- M.A. Bender, M. Farach-Colton: *The Level Ancestor Problem Simplified*, Theor. Comput. Sci. 321(1): 5–12 (2004).

5.2 Einführung

Zunächst sollte, zugunsten einer übersichtlicheren Notation, der Hyperfloor-Operator eingeführt werden: $\lfloor\lfloor x \rfloor\rfloor := 2^{\lfloor \lg x \rfloor}$ bezeichne die größte Zweierpotenz nicht größer als x.

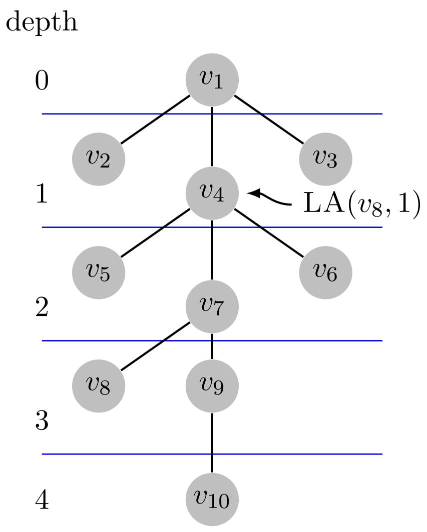
Für das Problem sei ein statischer Baum T mit n Knoten gegeben, der so *vorverarbeitet* werden soll, sodass Anfragen der folgenden Art möglichst effizient beantwortet werden können:

$$\text{LevelAncestor}_T(u, d) : \text{return } u\text{'s ancestor at depth } d \le \text{depth}(u)$$

Daraus folgt, dass $\text{LevelAncestor}_T(u, 0)$ die Wurzel und $\text{LevelAncestor}_T(u, \text{depth}(u))$ u selbst zurückliefert.

Direkt fallen zwei naive Ansätze ein, die in Laufzeit und Speicherverbrauch komplementär sind:

- speichere nur den Baum; suche nach Vorgänger mittels direkter Baumtraversierung
(Speicherverbrauch: *nicht zutreffend*, Laufzeit: $\mathcal{O}(n)$).
- speichere $\text{LevelAncestor}_T(u, d)$ für alle u und alle $0 \le d < \text{depth}(u)$
(Speicherverbrauch: $\mathcal{O}(n^2)$, Laufzeit: $\mathcal{O}(1)$).



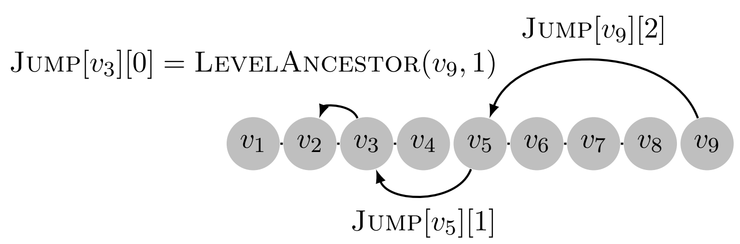
Nun ist die Herausforderung, die Vorteile beider Ansätze in einem einzigen Vorverarbeitungsverfahren zu vereinen. Es wird sich zeigen, dass sich in der Tat ein Verfahren konstruieren lässt, das sowohl linear im Platzverbrauch ist und mit welchem sich in konstanter Zeit das LevelAncestor-Problem beantworten lässt.

5.3 $\mathcal{O}(1)$ Level Ancestor mit $\mathcal{O}(n \lg n)$ Platzverbrauch

Wir werden uns - wie so häufig - schrittweiße der Lösung nähern, indem wir zunächst zwei Lösungen mit logarithmischer Laufzeit diskutieren und werden daraufhin einen Ansatz vorstellen, um diese beiden Techniken zu verbinden.

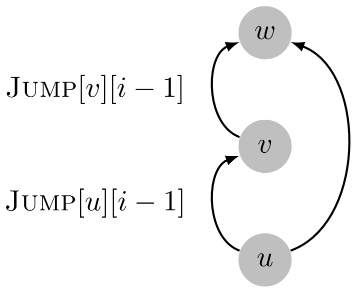
5.3.1 Jump-Pointer Algorithmus

Die grundlegende Idee ist es, ähnlich einer Skip Liste, gerade so viele Lösungen (oder Jump-Pointer) zu speichern, um gerade in logarithmisch vielen Schritten am Ziel anzukommen. In jedem Knoten u wird $\text{LevelAncestor}_T(u, d)$ also nur für $d = 1, 2, 4 \ldots, \lfloor\lfloor \text{depth}(u) \rfloor\rfloor$ anstatt für alle Vorgänger gespeichert. Hier bezeichne $\text{Jump}[u][i] := \text{LevelAncestor}_T(u, \text{depth}(u) - 2^i)$ den i-ten Jump-Pointer des Knotens u.



Die zentrale Beobachtung ist es nun, dass mit jedem *korrekten* Sprung mindestens die halbe Strecke zurückgelegt wird: Sei hierfür $\delta = \text{depth}(u) - d$ die zurückzulegende Distanz, dann überspringt Jump$[u][\lfloor \lg \delta \rfloor]$ eben $\lfloor\lfloor \delta \rfloor\rfloor \geq \frac{\delta}{2}$ Knoten. Der Algorithmus muss nun lediglich diesen Jump-Pointern folgen bis LevelAncestor$_T(u, d)$ erreicht ist.

Dies führt zu einer logarithmischen Laufzeit bei einem Platzverbrauch von $\mathcal{O}(n \lg n)$, wobei allerdings jeder Knoten $\leq \lg n$ zusätzlichen Speicher benötigt. Die Vorverarbeitung kann mittels dynamischer Programmierung (Top-Down) in $\mathcal{O}(n \lg n)$ Zeit durchgeführt werden:



Jump$[u][0] = \text{parent}(u)$

Jump$[u][i] = $ Jump[Jump$[u][i-1]][i-1], i > 0$

5.3.2 Ladder Algorithmus

Für den *Ladder Algorithmus* zerlegen wir zunächst den Baum in lange Pfade, indem wir schrittweiße den längsten Pfad entfernen. Das zerlegt den Baum in Teilbäume $T_1, T_2, \ldots$, die wiederum rekursiv zerlegt werden.

Dies kann in $\mathcal{O}(n)$ Zeit erreicht werden, indem zunächst für jeden Knoten, ausgehend von den Blättern (Bottom-Up), die größte Knoten-Blatt Entfernung berechnet wird. In einem weiteren Durchlauf, beginnend bei der Wurzel, kann nun für jeden Knoten der Nachfolger als das Kind mit der größten Knoten-Blatt Entfernung gewählt werden.

Entlang eines Pfades ist es *einfach*, den Level Ancestor zu finden: Falls die Knoten des Pfades π der Länge m in einem Array Ladder$_\pi[0, m-1]$ gespeichert sind und Ladder$_\pi[0]$ auf Tiefe h liegt, dann ist LevelAncestor$_T(u, d) = $ Ladder$_\pi[d-h]$ mit $d \geq h$ und $u \in \pi$. Ansonsten ist $d < h$ und wir können zu dem Elternpfad π' von π springen bis $d \geq h'$

Dies führt zu einer Laufzeit von $\mathcal{O}(\sqrt{n})$ da bis zu $\Theta(\sqrt{n})$ Pfade auf einem Weg von Wurzel zu Blatt liegen können.

Um dies auf $\mathcal{O}(\lg n)$ zu drücken, erweitern wir die Pfade zu Leitern (*Ladders*). Sei $|\pi| = m$, dann speichere in Ladder$_\pi$ nicht nur π, sondern zusätzlich die m direkten Vorgänger von $\pi[0]$.

Der Vorteil dieser Konstruktion ist, dass falls sich LevelAncestor$_T(u, d)$ nicht auf der Leiter befindet, dann ist Ladder$_\pi[0]$ mindestens *doppelt* so weit vom tiefsten Blatt im Teilbaum des Knotens u entfernt wie der Knoten u selbst.

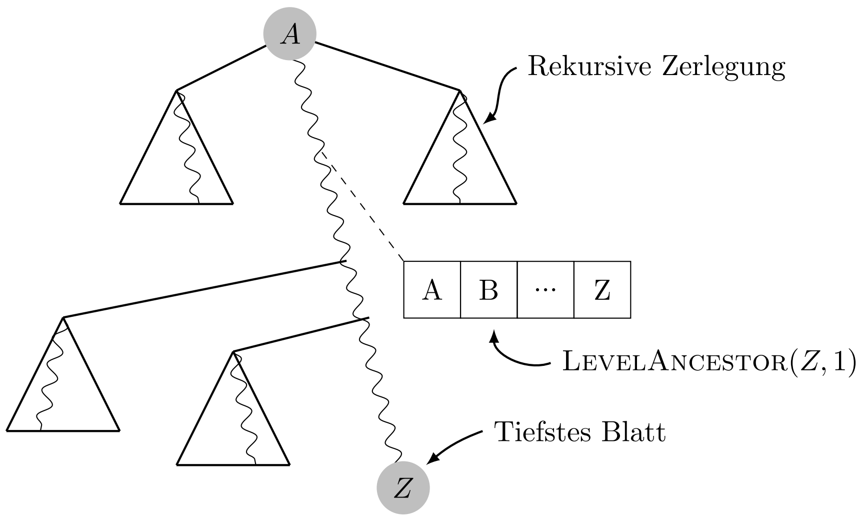


Figure 2: Ein längster Wurzel-Blatt Pfad, der den Baum T in vier Teilbäume zerlegt. Die Tabelle zeigt, wie eine LevelAncestor-Anfragen entlang des Pfades beantwortet werden kann.

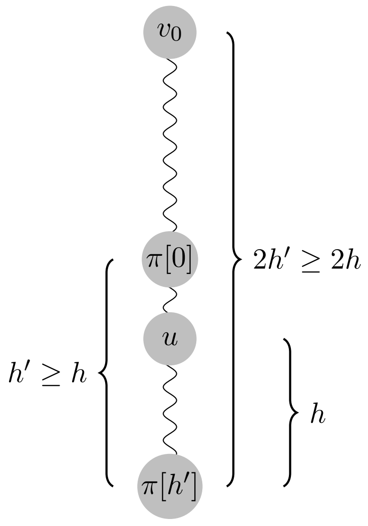


Figure 3: Diese Abbildung zeigt schematisch eine Leiter, die aus dem Pfad π entstanden ist. Hier sind h' die Länge des Pfades π, deshalb $2h'$ die Länge der Leiter und h die Entfernung von u zu $\pi[h']$.

Weiterhin kann wiederholt auf die nächsthöhere Leiter gewechselt werden, bis $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ auf der Leiter liegt, und da sich mit jedem Wechsel der Leiter die Entfernung zu den Blättern verdoppelt führt dies zu einer logarithmischen Laufzeit.

5.3.3 Beide Techniken verbinden

Konstante Laufzeit kann erreicht werden indem man beide Techniken verbindet. Hierfür springt man zunächst den halben Weg mit $\textsc{Jump}[u][\lfloor\delta\rfloor]$ nach oben. Man erreicht auf diese Weiße einen Knoten v mit $height(v) \geq \lfloor\lfloor \delta \rfloor\rfloor$. Da nun die Entfernung von v nach $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ kleiner als δ ist, ist $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ in der Leiter des Knotens v enthalten und man erreicht es in $\mathcal{O}(1)$.

5.4 $\mathcal{O}(1)$ Level Ancestor mit $\mathcal{O}(n)$ Platzverbrauch

Der aktuelle Platzverbrauch ist in den $\mathcal{O}(n \lg n)$ Jump-Pointern begründet, jedoch müssen diese nicht in *jedem* Knoten gespeichert werden, da $\textsc{LevelAncestor}_{\mathrm{T}}(v, d) = \textsc{LevelAncestor}_{\mathrm{T}}(w, d)$ für jeden Nachfolger w von v. Falls man also eine Menge von Jump Nodes, welche mit Pointern ausgestattet sind, auswählt, kann *jeder* Knoten *oberhalb* eines Jump Nodes diesen benutzen um die Anfrage zu beantworten. Alle anderen (also jene unterhalb der Jump Nodes) werden am Ende des Abschnitts behandelt.

Die Idee Jump Nodes zu bestimmen ist angelehnt an y-fast tries: *Wähle als Jump Nodes* die tiefsten Knoten die mindestens $s := \lg n/4$ Nachfolger besitzen (Was dazu führt, dass jedes Kind eines Jump Nodes weniger als s Knoten in seinem Teilbaum besitzt). Darüber hinaus muss für jeden Knoten überhalb der Jump Nodes ein Zeiger JUMP-DESC$[u]$ auf einen der nachfolgenden Jump Nodes gespeichert werden.

Wir werden im Folgenden den Teil des Baumes T oberhalb der Jump Nodes als *macro tree*, und die Teile unterhalb der Jump Nodes als *micro trees* bezeichnen. Da die Jump Nodes Wurzeln von disjunkten Teilbäumen mindestens der Größe s sind, gibt es höchstens n/s Jump Nodes. Deshalb ist der gesamte Platzverbrauch um alle Jump Pointer zu speichern höchstens

$$\frac{n}{s} \lg \frac{n}{s} \leq \frac{n}{s} \lg n = \mathcal{O}(\frac{n}{\lg n} \lg n) = \mathcal{O}(n)$$

und damit linear.

Die Jump Pointer können effizient in Linearzeit bestimmt werden, indem von dem jeweiligen Jump Node wiederholt den Leitern gefolgt wird. Das heißt, sobald die aktuelle Leiter nicht mehr ausreicht um die gerade zu speichernde Anfrage beantworten zu können, wechselt man auf die nächsthöhere. Da allerdings höchstens logarithmisch viele solcher Leiterwechsel benötigt werden um pro Knoten die logarithmisch vielen Jump Pointer zu bestimmen, kann jeder Jump Pointer in amortisiert konstanter Zeit bestimmt werden.

Um nun auch noch $\textsc{LevelAncestor}_{\mathrm{T}}(u, d)$ für alle Micro Trees beantworten zu können, kodiert man alle möglichen Bäume mit $s' < s$ Knoten als ein Bitmuster der Größe $2(s' - 1)$: Schreibe eine '0' falls man in einem DFS Schritt auf dem Baum nach unten geht und eine '1' falls man nach oben geht.

Nun speichert jeder Micro Tree sein Bitmuster (mit hinten aufgefüllten Nullen um eine Länge von $2(s - 1)$ zu erreichen), einen Zeiger auf seinen nächsten Vorgänger im Macro Tree und eine Tabelle um DFS Nummern in globale Schlüssel umzurechnen. Zusätzlich wird eine *globale Lookup-Tabelle* benötigt, die für jedes auftretende Bitmuster die Antworten zu allen möglichen $\textsc{LevelAncestor}$-Anfragen enthält, wobei jede Antwort als DFS-Nummer in diesem spezifischen Baum gegeben ist. Zusammen mit der Transformationstabelle lassen sich nun alle

LEVELANCESTOR$_T(u,d)$ Anfragen für u in einem Micro Tree und LEVELANCESTOR$_T(u,d)$ im selben Micro Tree beantworten. Falls LEVELANCESTOR$_T(u,d)$ sich nicht in diesem Micro Tree befindet, wird die Anfrage durch den im Micro Tree gespeicherten Vorgänger im Macro Tree beantwortet.

Es ist nun also möglich in konstanter Laufzeit LEVELANCESTOR-Anfragen für alle Knoten vom Typ LEVELANCESTOR$_T(u,d)$ zu beantworten, wobei sich für die Größe der Tabellen ergibt:

$$\# \text{ Bitmuster der Länge } 2s \times \# \text{ der } u\text{'s} \times \# \text{ der } d\text{'s}$$

$$2^{2s} \times s \times s$$

was gerade $\mathcal{O}(2^{\lg n/2}s^2) = \mathcal{O}(\sqrt{n}\lg^2 n) = o(n)$ ergibt.

6 The 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.*

6.1 The Transformation

Definition 12. *Let $T = t_1t_2\dots t_n$ 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 $T_{i\dots n}T_{1\dots i-1}$. We denote it by $T^{(i)}$.*

Example 1.

$$T = \overset{1}{\text{C}}\overset{2}{\text{A}}\overset{3}{\text{C}}\overset{4}{\text{A}}\overset{5}{\text{A}}\overset{6}{\text{C}}\overset{7}{\text{C}}\overset{8}{\text{A}}\overset{9}{\text{C}}\overset{10}{\$}$$

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

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

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

Example 2.

$T = \text{CACAACCAC\$}$

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

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.

6.2 Construction of the BWT

The BWT-matrix need *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

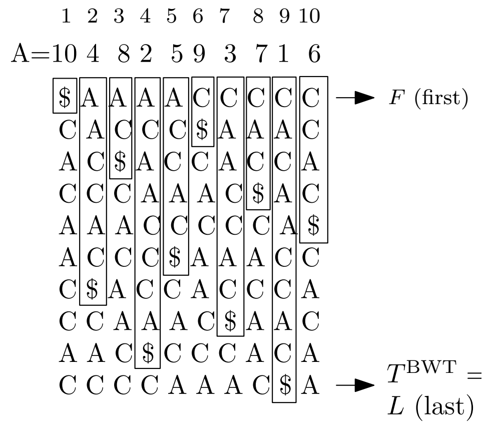
$$T^{\text{BWT}}[i] = t_{A[i]-1} \ ,$$

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. 1), we get:

Theorem 10. *The BWT of a text length-n text over an integer alphabet can be constructed in $O(n)$ time.* □

Example 3.

$T = \text{CACAACCAC\$}$



$$\begin{array}{r|cccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{A=} & 10 & 4 & 8 & 2 & 5 & 9 & 3 & 7 & 1 & 6 \\
\hline
 & \boxed{\$} & A & A & A & A & C & C & C & C & C \\
 & C & A & C & C & C & \$ & A & A & A & C \\
 & A & C & \$ & A & C & C & A & C & C & A \\
 & C & C & C & A & A & A & C & \$ & A & C \\
 & A & A & A & C & C & C & C & C & A & \$ \\
 & A & C & C & C & \$ & A & A & A & C & C \\
 & C & \$ & A & C & C & A & C & C & C & A \\
 & C & C & A & A & A & C & \$ & A & A & C \\
 & A & A & C & \$ & C & C & C & A & C & A \\
 & C & C & C & C & A & A & A & C & \$ & A
\end{array}$$

6.3 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 13. *Let F and L be the strings resulting from the* BWT. *Then the* last-to-front *mapping* LF *is a permutation of* $[1, n]$, *defined by*

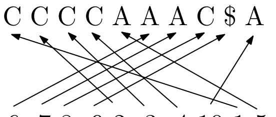
$$\text{LF}(i) = j \Longleftrightarrow A[j] = A[i] - 1 \ .$$

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

Example 4.

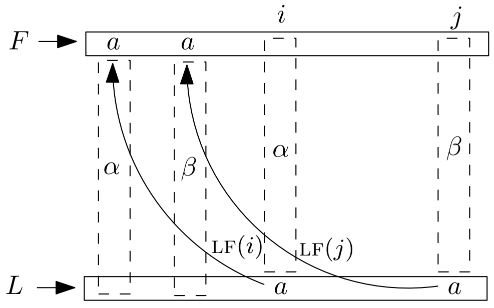
$$T = \text{CACAACCAC\$}$$

$$\begin{array}{ccccccccccl}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \\
\$ & A & A & A & A & C & C & C & C & C & \longrightarrow F \text{ (first)} \\
C & A & C & C & C & \$ & A & A & A & C & \\
A & C & \$ & A & C & C & A & C & C & A & \\
C & C & C & A & A & A & C & \$ & A & C & \\
A & A & A & C & C & C & C & C & A & \$ & \\
A & C & C & C & \$ & A & A & A & C & C & \\
C & \$ & A & C & C & A & C & C & C & A & \\
C & C & A & A & A & C & \$ & A & A & C & \\
A & A & C & \$ & C & C & C & A & C & A & \\
C & C & C & C & A & A & A & C & \$ & A & \longrightarrow T^{\text{BWT}} = L \text{ (last)}
\end{array}$$



$$\text{LF} = 6\ \ 7\ 8\ \ 9\ 2\ \ 3\ \ 4\ 10\ 1\ 5$$

Observation 3. *Equal characters preserve the same order in F and L. That is, if $L[i] = L[j]$ and $i < j$, then* LF(i) < LF(j). *To see why this is so, recall that the* BWT*-matrix is sorted lexicographically. Because both the* LF(i)*'th and the* 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* LF(i) < LF(j).



This observation allows us to compute the LF-mapping *without knowing the suffix array* of T. Using LF, we can recover $T = t_1t_2 \dots t_n$ *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 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.

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

$$\uparrow$$

T reversed

6.4 Compression

Storing T^{BWT} *plainly* needs the same space as storing the original text T. However, because equal characters are grouped together in T^{BWT}, we can compress T^{BWT} in a second stage.

We can directly exploit that T^{BWT} consists of many equal-letter runs. Each such run a^ℓ can be encoded as a pair (a, ℓ) with $a \in \Sigma, \ell \in [1, n]$. This is known as *run-length encoding.*

Example 6.

$$T^{\text{BWT}} = \overset{1\,2\,3\,4\,5\,6\,7\,8\,9\,10}{\text{CCCCAAAC\$A}}$$

$$\Rightarrow \text{RLE}(T^{\text{BWT}})=(\text{C},4),(\text{A},4),(\text{C},1),(\$,1),(\text{A},1)$$

A different possibility for compression is to proceed in two steps: first, we perform a move-to-front encoding of the BWT. Then, we review different methods for compressing the output of the move-to-front algorithm using a 0-order compressor. Both steps are explained in the following sections.

6.4.1 Move-to-front (MTF)

- Initialize a list Y containing each character in Σ in alphabetic order.
- In a left-to-right scan of T^{BWT}, $(i = 1, \ldots, n)$, compute a new array $R[1, n]$:
  - Write the position of character T_i^{BWT} in Y to $R[i]$.
  - Move character T_i^{BWT} to the front of Y.

MTF is easy to reverse.

Observation 4. *MTF produces "many small" numbers for equal characters that are "close together" in T^{BWT}. These can be compressed using an order-0 compressor, as explained next.*

6.4.2 0-Order Compression

We looked at Huffman-, unary-, Elias-γ and Elias-δ codes, but found that Huffman is definitely the best choice in this setting.

7 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 not too large, namely $\sigma = o(n/\log\sigma)$. This is a realistic assumption in practice (even if the alphabet consists of all *words* in a language, as we shall see in the chapter on information retrieval).

7.1 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!

Definition 14. *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* OCC $: \Sigma \times [1, n] \to [1, n]$ *such that* $\text{OCC}(a, i)$ *is the number of occurrences of a in L's length-i-prefix $L[1, i]$.*

Lemma 11. *With the definitions above,*

$$\text{LF}(i) = C(L[i]) + \text{OCC}(L[i], i) \ .$$

Proof: Follows immediately from observation 3 □

7.2 Backward Search

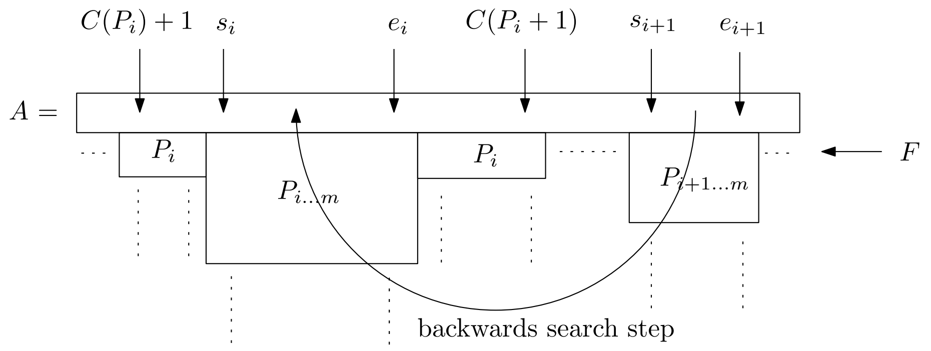
This section describes how the Burrows-Wheeler-Transformation can be used as an *index* on the text. We first focus our attention on the *counting problem* (p. 4); i.e., on finding the number of occurrences of a pattern $P_{1\ldots m}$ in $T_{1\ldots n}$.

We define two functions that are needed by the backwards-search algorithm.

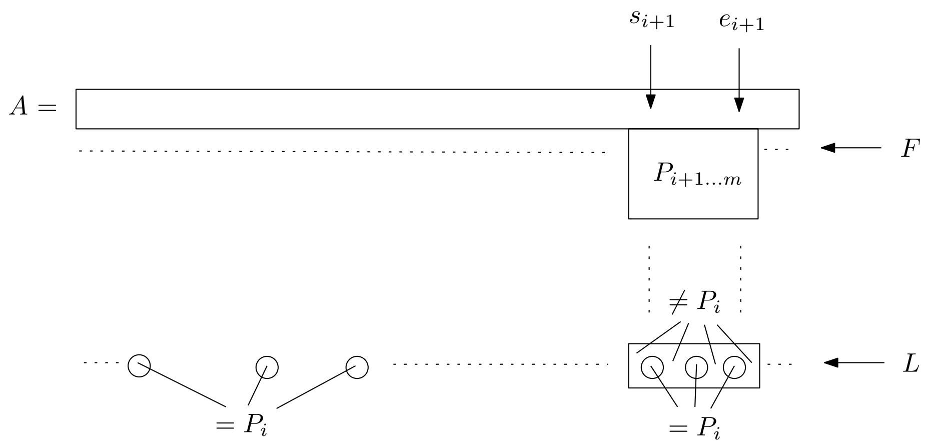
- $C(a)$ denotes the number of occurrences in T of characters lexicographically smaller than $a \in \Sigma$.
- $\mathsf{rank}_a(L, i)$ denotes the number of occurrences of a in $L[1, i]$.

The C-function can be stored as a plain array, using $\sigma\lceil \log n \rceil$ bits, which is $o(n)$ bits if $\sigma = o(n/\log \sigma)$. Solutions for rank will be given in Sect. 7.4.

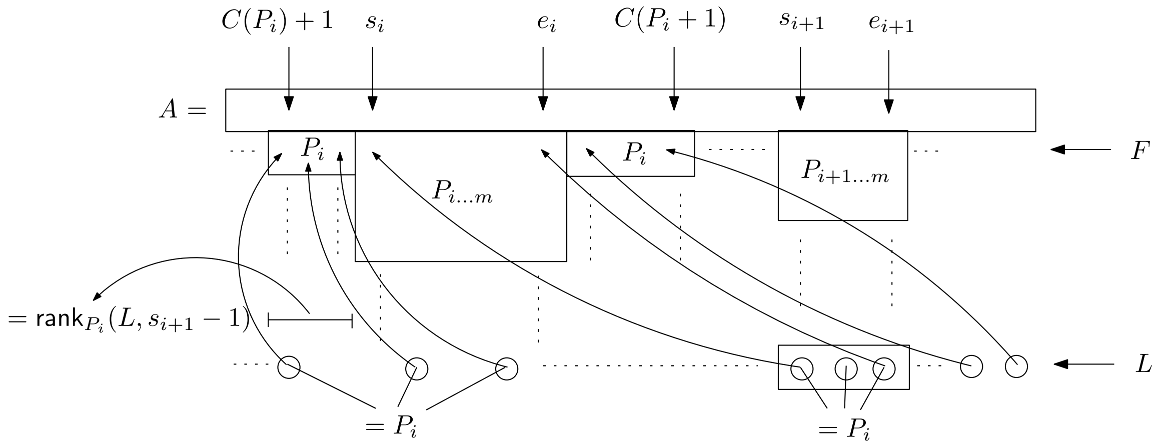
Our aim is identify the interval of P in A by searching P from right to left (= backwards). To this end, suppose we have already matched $P_{i+1\ldots m}$, and know that the suffixes starting with $P_{i+1\ldots m}$ form the interval $[s_{i+1}, e_{i+1}]$ in A. In a *backwards search step*, we wish to calculate the interval $[s_i, e_i]$ of $P_{i\ldots m}$. 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 $\mathsf{rank}_{P_i}(L, s_{i+1}-1)$. 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 rank-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:

Algorithm 6: function `backwards-search`$(P_{1\ldots m})$

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

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 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 $\textsc{occ}$-values, while using less space implies a higher query time.

Theorem 12. *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.*

7.3 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 a more *more practical implementation* between these two extremes, let us do the following: 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

$$\mathsf{rank}_c(L, i) = \mathsf{rank}_1(B_c, i) \ .$$

We shall see presently that a bit-vector B, together with additional information for constant-time rank-operations, can be stored in $n + o(n)$ bits. Then the total space for all σ indicator bit vectors is $\sigma n + o(\sigma n)$ bits. Note that for *reporting* queries, we still need the suffix array to output the values in $A[s, e]$ after the backwards search.

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

Theorem 13. *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 $n \log \sigma + \sigma \log n + \sigma n + o(\sigma n)$.* □

Example 7.

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

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

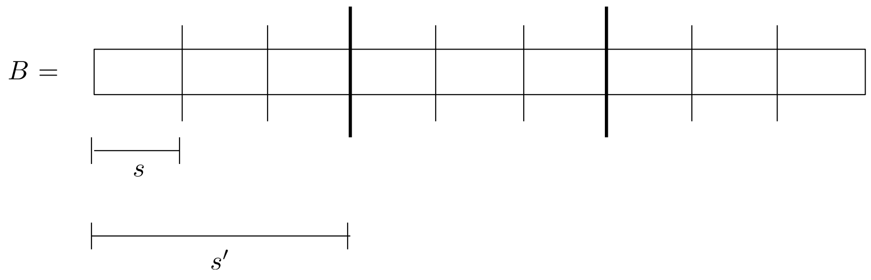
7.4 Compact Data Structures on Bit Vectors

We now show that a bit-vector B of length n can be augmented with a data structure of size $o(n)$ bits such that rank-queries can be answered in $O(1)$ time. First note that

$$\mathsf{rank}_0(B, i) = i - \mathsf{rank}_1(B, i) \ ,$$

so considering rank_1 will be enough.

We conceptually divide the bit-vector B into *blocks* of length $s = \lfloor \frac{\log n}{2} \rfloor$ and *super-blocks* of length $s' = s^2 = \Theta(\log^2 n)$.



The idea is to *decompose* a rank_1-query into 3 sub-queries that are aligned with the block- or super-block-boundaries. To this end, we store three types of arrays:

1. For all of the $\lfloor \frac{n}{s'} \rfloor$ super-blocks, $M'[i]$ stores the number of 1's from B's beginning up to the end of the i'th superblock. This table needs order of

   $$\underbrace{n/s'}_{\#\text{superblocks}} \times \underbrace{\log n}_{\text{value from } [1,n]} = O\left(\frac{n}{\log n}\right) = o(n)$$

   bits.

2. For all of the $\lfloor \frac{n}{s} \rfloor$ blocks, $M[i]$ stores the number of 1's from the beginning of the superblock in which block i is contained up to the end of the i'th block. This needs order of

   $$\underbrace{n/s}_{\#\text{blocks}} \times \underbrace{\log s'}_{\text{value from } [1,s']} = O\left(\frac{n \log \log n}{\log n}\right) = o(n)$$

   bits of space.

3. For all bit-vectors V of length s and all $1 \leq i \leq s$, $P[V][i]$ stores the number of 1-bits in $V[1,i]$. Because there are only $2^s = 2^{\frac{\log n}{2}}$ such vectors V, the space for table P is order of

$$\underbrace{2^{\frac{\log n}{2}}}_{\#\text{possible blocks}} \times \underbrace{s}_{\#\text{queries}} \times \underbrace{\log s}_{\text{value from }[1,s]} = O\left(\sqrt{n}\log n \log\log n\right) = o(n)$$

bits.

Example 8.

$s = 3 \quad s' = 9$

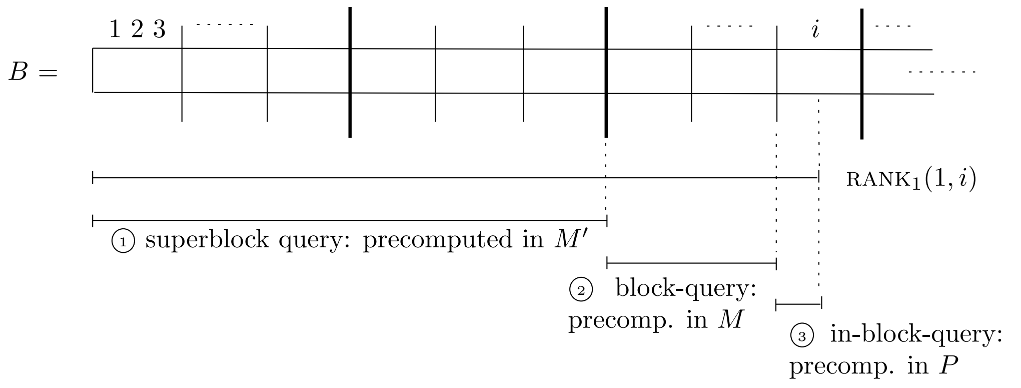
$$B = \Big|\,0\,1\,0\,\big|\,1\,1\,0\,\big|\,1\,1\,1\,\Big|\,0\,0\,0\,\big|\,1\,1\,1\,\big|\,0\,0\,1\,\Big|\,1\,0\,0\,\big|\,1\,1\,0\,\big|\,0\,1\,0\,\Big|$$

$M' = 6\ 10\ 14$
$M = 1\ 3\ 6\ 0\ 3\ 4\ 1\ 3\ 4$

P:

$V \setminus i$	1	2	3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

A query $\mathsf{rank}_1(B,i)$ is then decomposed into 3 sub-queries, as seen in the following picture:


B =
1 2 3
i
RANK₁(1, i)
① superblock query: precomputed in M′
② block-query: precomp. in M
③ in-block-query: precomp. in P

Thus, computing the block number as $q = \lfloor \frac{i-1}{s} \rfloor$, and the super-block number as $q' = \lfloor \frac{i-1}{s'} \rfloor$, we can answer

$$\mathsf{rank}_1(B,i) = M'[q'] + M[q] + P[\underbrace{B[qs+1,(q+1)s]}_{i\text{'s block}}]\ \underbrace{[i-qs]}_{\text{index in block}}$$

in constant time.

Example 9. *Continuing the example above, we answer* $\mathsf{rank}_1(B,17)$ *as follows: the block number is* $q = \lfloor \frac{17-1}{3} \rfloor = 5$, *and the super-block number is* $q' = \lfloor \frac{17-1}{9} \rfloor = 1$. *Further, i's block is* $B[5 \times 3 + 1, 6 \times 3] = B[16,18] = 001$, *and the index in that block is* $17 - 5 \times 3 = 2$. *Hence,* $\mathsf{rank}_1(B,17) = M'[1] + M[5] + P[001][2] = 6 + 3 + 0 = 9$.

This finishes the description of the data structure for $O(1)$ rank-queries. We summarize this section in the following theorem.

Theorem 14. *An n-bit vector B can be augmented with data structures of size $o(n)$ bits such that $\mathsf{rank}_b(B,i)$ can be answered in constant time ($b \in \{0,1\}$).*

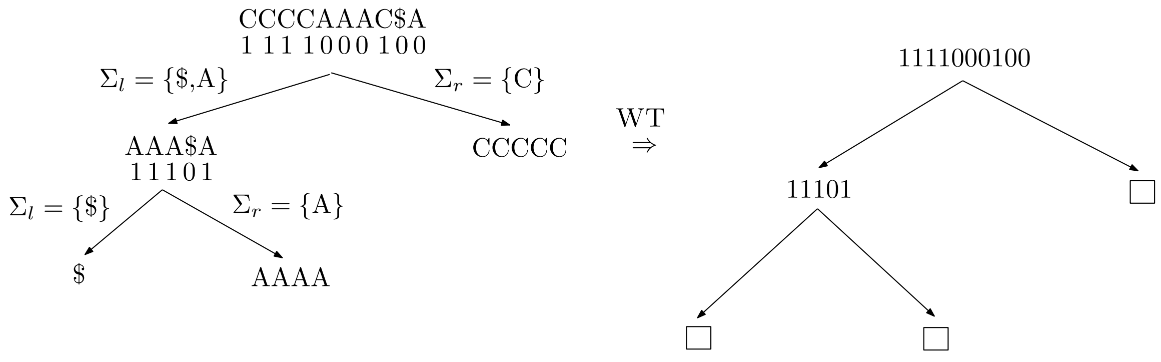
7.5 Wavelet Trees

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 10.

L=CCCCAAAC\$A $\quad\quad \Sigma$ ={\$,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 15. *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 ($b \in \{0,1\}$):

$$\mathsf{rank}_b(B_v, i) = \mathsf{rank}_b(B_d, \alpha_v + i - 1) - \mathsf{rank}_b(B_d, \alpha_v - 1) \ ,$$

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

NB: we noted in the lecture that (1) we should rather concatenate all bit-vectors into *one* large bit-vector of size $n \lg \sigma$ bits and build the rank-data structure onto this, and (2) that the above-mentioned α_v's/β_v's are not really necessary.

How does the wavelet tree help for implementing the OCC-function? Suppose we want to compute $\textsc{occ}(c, i)$, 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 $L_r[1, \mathsf{rank}_0(B_r, i)]$. 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 $R_r[1, \mathsf{rank}_1(B_r, i)]$ as the number of c's in $L[1, i]$. This leads to the following recursive procedure for computing $\textsc{occ}(c, i)$, to be invoked with WT-$\textsc{occ}(c, i, 1, \sigma, r)$, 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 7: function `WT-occ`$(c, i, \sigma_l, \sigma_r, v)$

```
1 if $\sigma_l = \sigma_r$ then
2     return $i$;
3 end
4 $\sigma_m = \lfloor \frac{\sigma_l + \sigma_r}{2} \rfloor$;
5 if $c \leq \Sigma[\sigma_m]$ then
6     return WT-occ$(c, \mathsf{rank}_0(B_v, i), \sigma_l, \sigma_m, l_v)$;
7 else
8     return WT-occ$(c, \mathsf{rank}_1(B_v, i), \sigma_m + 1, \sigma_r, r_v)$;
9 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 16. *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*

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

Note that we can upper bound the $O(\sigma \log n)$-term for C by $O(n)$ bits, since if $\sigma \log n \geq n$, instead of C we can always use a bit-vector B_C of length n that marks the bucket endings with a '1', such that $C[i] = \mathsf{rank}_1(B_c, i)$.

7.5.1 Further Applications of Wavelet Trees

We looked at range quantile queries (find the i'th smallest element in a given range $B[\ell, r]$), range next value queries (in a given range $B[\ell, r]$, find the smallest value $\geq i$, and 4-sided

2-dimensional range searching (given n static points on a grid $[1, n] \times [1, n]$, list all points in $[x_\ell, x_r] \times [y_b, y_t]$). All of these queries have applications in text indexing, as we saw in the lectures/exercises.

7.6 Sampling the Suffix Array

If we also want to solve the *reporting problem* (outputting all starting positions of P in T, see p. 4), 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 $A'[\mathsf{rank}_1(S, i)]$. 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 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 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.

8 Inverted Indexes

An inverted index is a *word*-based data structure for answering queries of the form "which documents contain pattern P?" (and more complicated ones, such as queries with multiple patterns), as we know them from the search engines that we use everyday on the internet.

More formally, let $\mathcal{S} = \{T_1, \ldots, T_k\}$ the set of documents to be indexed, each consisting of words over an alphabet Σ. Let n denote the total size of the documents ($n = \sum_{i=1}^{k} |T_i|$), and w the total number of words. (Texts can be tokenized into words by taking, for example, all alphanumeric characters between two consecutive non-alphanumeric characters.) A simple *inverted* index over $\mathcal{S}$ consists of the following two components:

Vocabulary V: The set *different* words occurring in $\mathcal{S}$. An empirical observation ("Heap's Law") is that $|V| = O(w^b)$, usually $4/10 < b < 6/10$. We assume a total order on V (e.g., lexicographic).

Postings Lists $L_1, \ldots, L_{|V|}$: For the i'th word S_i in the vocabulary V, L_i consists of all document numbers where S_i occurs as a word, listed in some total order on $\mathcal{S}$ (e.g., ranked by a static measure of document importance).

8.1 Queries

The above data structure—vocabulary plus postings lists—are already enough to answer basic one-pattern queries. For queries with more than one pattern ($P_1 P_2 \ldots P_q$), we first look at the possible semantics of those queries (for simplicity only for two patterns P_1 and P_2):

conjunctive: List all documents containing P_1 *and* P_2. Here, the postings lists for P_1 and P_2 have to be *intersected.*

disjunctive: List all documents containing P_1 *or* P_2. Here, the postings lists for P_1 and P_2 have to be *united*.

phrase query: List all documents containing the sequence P_1P_2 exactly in this order—those queries are often specified by quotes. We will see later (in the chapter on suffix arrays) how to handle those queries.

8.2 Representing the Vocabulary

Tries from Sect. 2 are an obvious possibility for representing V. Each leaf in the trie would then store a pointer to the corresponding inverted list.

Another possibility are hash tables. Assume we have a hash function h from $\Sigma^\star$ to $[0, m)$, for $m = C \cdot |V|$ denoting the size of the hash table with constant $C > 1$ (e.g., $C = 2$). Then we allocate a table M of size m, and at $M[h(S_i)]$ we store the inverted list L_i. The location $h(S_i)$ also stores the string S_i, in order to decide if the query word P really equals S_i. Collisions in M are handled in one of the usual ways, e.g., chaining or linear probing.

The question that remains to be answered is: "What is a good hash-function for (potentially unbounded-length) strings?". In actual programming languages, one often finds functions like $h(t_1t_2 \dots t_\ell) = (t_1 + K \cdot h(t_2 \dots t_\ell)) \mod m$ for some constant K. Those functions are easy to fool because they have some predetermined worst-case behavior. Better are so-called *randomized* hash functions like

$$h(t_1t_2 \dots t_\ell) = \left(\left(\sum_{i=1}^{\ell} a_i \cdot t_i \right) \mod p \right) \mod m$$

for a prime $p \geq m$ and *random* $a_i \in [1, p)$ ("multiplicative hashing"). Those functions have good worst-case guarantees, like $\mathsf{Prob}\left[h(x) = h(y)\right] = O(1/m)$ for $x \neq y$. Note that the necessary random numbers could be generated "on the fly" from a single random seed while computing the sum. However, at least for strings no longer than a certain threshold those random values should be precomputed and stored in RAM—then only for the longer ones the random numbers have to be recomputed.

8.3 List Intersection

As already said in Sect. 8.1, we need algorithms for intersecting two lists L_1 and L_2 (list union algorithms are similar). Assume w.l.o.g. that $|L_1| \leq |L_2|$. To compute $L_1 \cap L_2$, we can do the following:

1. Search every element from L_1 by a *binary search* in L_2. Time is $O(|L_1| \log |L_2|)$.

2. Walk simultaneously through both lists with one pointer each, increasing the pointer pointing to the smaller of the two values, and outputting numbers that occur in both lists. (This is similar to the *merging*-procedure in Merge-Sort.) Time is $O(|L_1| + |L_2|)$.

3. *Double Binary Search*: Take the median $L_1[\mu]$ of L_1 ($\mu = \lfloor |L_1|/2 \rfloor$), and do a binary search in L_2 to locate the position of μ in L_2, say at position λ. (If $L_1[\mu] = L_2[\lambda]$, then $L_1[\mu]$ is output at this point and the element is removed from both lists.) Then do two recursive calls of the same procedure, one for computing $L_1[1, \mu] \cap L_2[1, \lambda]$, and another one for computing $L_1[\mu + 1, |L_1|] \cap L_2[\lambda + 1, |L_2|]$. We analyzed in the lecture that this algorithm results in worst-case time $O(|L_1| \log(|L_2|/|L_1|))$. Note in particular that this is

never worse than possibilities (1) and (2) above—for if $|L_2| = K \cdot |L_1|$ with constant K, we have $O(|L_1| \log(|L_2|/|L_1|)) = O(|L_1| \log K) = O(|L_1|)$.

4. *Exponential Search*: This method is particularly useful if the lists are too long to be loaded entirely from disk into main memory (which is needed for method (3) above). Suppose we have already processed elements $L_1[1,k]$, and suppose the place for $L_1[k]$ in L_2 is x. Then to locate $L_1[k+1]$ in L_2, we compare it successively to $L_2[x]$, $L_2[x+1]$, $L_2[x+2]$, $L_2[x+4]$, ..., until the element $L_2[x+2^i]$ is larger than $L_1[k+1]$ (if it is "=", we are done). We then binary search $L_1[k+1]$ in $L_2[x+2^{i-1}, x+2^i]$. The whole procedure takes $O(\log d_k)$ if the distance between $L_1[k]$ and $L_1[k+1]$ in L_2 is d_k. This results in overall $O(\sum_{i=1}^{|L_1|} \log d_i)$ running time, which is maximized for $d_i = |L_2|/|L_1|$ for all i, and in this case the time is again $O(|L_1| \log(|L_2|/|L_1|))$.

8.4 Postings List Compression

Look at a particular list $L_i = [d_1, d_2, \ldots, d_{|L_i|}]$. Since the document ids listed in L_i are sorted, it can be beneficial to encode the *differences* between consecutive entries, in particular for dense lists. Formally, define $\delta_i = d_i - d_{i-1}$ with $d_0 = 0$. Since for dense lists we have that most δ_i's must be small, we now need to find codes that encode small numbers in few bits. Ideally, we would want to represent an integer $x \geq 1$ in $\lfloor \log_2 x \rfloor + 1$ bits (binary representation without leading 0's - denote this code by $(x)_2$). But this is problematic, since in a stream of such words we could not tell where the encodings of the δ_i's start and end.

8.4.1 Unary Code

To get the unary code for x, we write $(x-1)$ 1's, followed by a single 0. More formally, we let $(x)_1 = 1^{x-1} \circ 0$ denote the unary representation of x, where "$\circ$" denotes the concatenation of bit-strings. The length of this code is $|(x)_1| = x$ bits.

8.4.2 Elias Codes

We first introduce the γ-code. We encode x as

$$\begin{aligned} (x)_\gamma &= (|(x)_2|)_1 \circ (x)_2 \\ &= (\lfloor \log_2 x \rfloor + 1)_1 \circ (x)_2 \end{aligned}$$

where we omit the leading '1' in the $(x)_2$ (since it is redundant, once we know the length of $(x)_2$. The length of this code is $|(x)_\gamma| \approx 2 \log_2 x$ bits, since both the unary and the binary part take about $\log_2 x$ bits each.

Now the δ-code is similar, but the unary part is replaced by γ-coded numbers:

$$\begin{aligned} (x)_\delta &= (|(x)_2|)_\gamma \circ (x)_2 \\ &= (\lfloor \log_2 x \rfloor + 1)_\gamma \circ (x)_2 \end{aligned}$$

where again we omit the leading '1' in the $(x)_2$. The length of this code is $|(x)_\delta| \approx \log_2 x + \log_2 \log_2 x$ bits.

8.5 Ternary Code

We write $x-1$ in ternary and then substitute the trit 0 by the two bits 00, the trit 1 by the two trits 01, and the trit 2 by the bits 10. Finally, a pattern 11 is appended, and the resulting representation of x is denoted by $(x)_3$. The length of this code is $|(x)_3| = 2\lfloor \log_3(x-1)\rfloor + 2$ bits.

8.6 Fibonacci Code

Every integer x can be represented by the sum of different Fibonacci numbers, such that no two adjacent Fibonacci numbers appear in the sum. We can encode the Fibonacci numbers that contribute to this sum by 1-bits (higher Fibonacci-numbers should be written at the right end, in contrast to the usual 'most-significant-bit-left' rule). Since the pattern '11' does not appear in the resulting bit-sequence, we finally append a '1' to the bit-sequence to obtain the Fibonacci code $(x)_\phi$ for x.

8.7 Golomb Codes

The following paragraph is quoted more or less verbatim from:

- R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, second edition, 2011.

For some parameter b, let q and r be the quotient and remainder, respectively, of dividing $x-1$ by b: $q = \lfloor (x-1)/b \rfloor$ and $r = (x-1) - qb$. Then x is coded by concatenating the unary representation of $q+1$ and the binary representation of r, using either $\lfloor \log_2 b\rfloor$ or $\lceil \log_2 b\rceil$ bits for the latter, in the following way: If $r < 2^{\lfloor \log_2 b\rfloor -1}$, then r uses $\lfloor \log_2 b\rfloor$ bits, and the representation always starts with a 0-bit. Otherwise, it uses $\lceil \log_2 b\rceil$ bits, where the first bit is '1', and the remaining bits encode the value $r - 2^{\lfloor \log_2 b\rfloor -1}$ in $\lfloor \log_2 b\rfloor$ binary digits.

For example, for $b = 3$ there are three possible remainders, and those are coded as 0, 10, and 11, for $r = 0$, $r = 1$, and $r = 2$, respectively. For $b = 5$ there are five possible remainders r, 0 through 4, and these are assigned the codes 00, 01, 100, 101, and 110.

The Golomb code of x with parameter b is denoted by $(x)_{\mathrm{Gol}(b)}$. Plots showing the sizes of the different codes can be seen in Fig. 4.

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

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 17. *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.*

9.1 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 16. *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 18. *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 19. *For any pair of nodes v and w in T, $\textsc{lca}_T(v,w) = I[\textsc{rmq}_D(p_v, p_w)]$.*

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.

9.1.1 Reverse Direction (*)

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 17. *Let $A[1,n]$ 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 $p = \text{argmin}_{1\leq i\leq n} A[i]$.*
- *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 $v_1,\dots,v_k$ 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 20. *Let A and B be two arrays with equal Cartesian Trees. Then* $\textsc{rmq}_A(\ell,r) = \textsc{rmq}_B(\ell,r)$ *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 $\textsc{rmq}(1,1) = 1$ 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,

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

In total, we see that $\textsc{rmq}_A(\ell,r) = \textsc{rmq}_B(\ell,r)$ 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). □

9.2 $O(1)$-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 $\textsc{rmq}_D(l,r)$ can be decomposed into two overlapping sub-queries of equal length 2^h with $h = \lfloor \log_2(r-l+1) \rfloor$:

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

The final answer is then given by $\text{RMQ}_D(l,r) = \text{argmin}_{\mu \in \{m_1,m_2\}} D[\mu]$. 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

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

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

9.3 $O(1)$-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 $\text{RMQ}_D(l,r) = \text{argmin}_{\mu \in \{m_1,m_2,m_3\}} D[\mu]$, 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).

9.3.1 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. 9.2, 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: $W[i] = \text{argmin}_{(i-1)s < j \leq is} D[j]$. A query of the form $\text{RMQ}_D(is+1, (j-1)s)$ is then answered by $W[\text{RMQ}_{D'}(i+1, j-1)]$.

9.3.2 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. 9.3.1.

As a consequence of Lemma 20, 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 21. *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.

10 Document Retrieval

10.1 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.

10.2 The Straight-Forward Solution

Define a string

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

of length $n := \sum_{1\leq i\leq m}(|S_i|+1) = m+\sum_{1\leq i\leq m}|S_i|$. 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:

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

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?).

10.3 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 $|S_1| \geq \frac{n}{2}$.

10.4 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 $\texttt{report}(\ell, r)$, which is a procedure defined as follows.

The claimed $O(d)$ running time of the call to $\texttt{report}(\ell, r)$ 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 $\texttt{report}(i', j')$ for some $\ell \leq i' \leq j' < i$. Hence, we can safely stop the recursion at this point.

11 Lempel-Ziv Compression

11.1 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 18. *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: $\textsc{lcp}_T(i, j) = \max\{\ell \geq 0 \ : \ T_{i\ldots i+\ell-1} = T_{j\ldots j+\ell-1}\}$.*

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 $T_{i\ldots i+\textsc{lcp}(i,j)-1}$.

Note also that the LCP-array H from Sect. 3.1 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 22. *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, $T_{i\ldots i+\ell-1}$ is a common prefix of all suffixes $T^{A[k]}$ for $A^{-1}[i] \leq k \leq A^{-1}[j]$. Hence, $T_{i\ldots i+\ell-1}$ is a common prefix of T^i and T^j.

Now assume that $T_{i\ldots i+\ell}$ is also a common prefix of T^i and T^j. Then, by the lexicographic order of A, $T_{i\ldots i+\ell}$ is also a common prefix of $T^{A[m-1]}$ and $T^{A[m]}$. But $|T_{i\ldots i+\ell}| = \ell + 1$, 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. □

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 19. *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: $\textsc{lcs}_T(i, j) = \max\{k \geq 0 : T_{i-k+1\ldots i} = T_{j-k+1\ldots j}\}$.*

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

11.2 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 20. *The* longest-previous-substring-array $L[1, n]$ *is defined such that* $L[i] = \max\{\ell \geq 0 : \exists k < i$ *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 21. *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 23. *For all $2 \leq i \leq n$: $L[i] = \max\{\textsc{lcp}(i, j) : 1 \leq j < i\}$.* □

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 22. *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 24. *For all $1 \leq i \leq n$, $L[A[i]] = \max(\textsc{lcp}(A[PSV_A(i)], A[i]), \textsc{lcp}(A[i], A[NSV_A(i)]))$.*

Proof. Rewriting the claim of Lemma 23 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 22.
- Build L by applying Lemma 24 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.

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

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 $(b_1, e_1), \ldots, (b_k, e_k)$ such that $s_i = T_{b_i \ldots e_i}$. 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 8: $O(n)$-computation of the LZ-factorization

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

11.4 Without LCP-Array and RMQ

We saw in the exercises that the algorithm still runs in linear time if we do not use constant-time lcp-queries: Suppose that a prefix $T_{1,\ldots,i-1}$ is already factored into $s_1 s_2 \ldots s_{k-1}$. To find the next factor s_k, we compare the string $T[i,n]$ *naively* with $T[NSV_A(A^{-1}[i]), n]$ and $T[PSV_A(A^{-1}[i]), n]$ until a mismatch occurs—the longer match gives the length of the new factor s_k. Here it is important to note that not the entire L-array is computed.

11.5 A More Space Efficient Algorithm

The idea for an even more space efficient algorithm is to factor the string T by the usual pattern matching algorithm. Suppose that a prefix $T_{1,\ldots,i-1}$ is already factored into $s_1 s_2 \ldots s_{k-1}$. To find the next factor s_k, we start matching t_i in the text T itself, with the help of the suffix array A. Suppose the t_i-interval in A is $[\ell, r]$. Then t_i occurs before position i iff there is a value in $A[\ell, r]$ that is less than i, in particular iff the minimum in $A[\ell, r]$ is less than i. This can be efficiently checked by range minimum queries over $A[\ell, r]$. We then continue and find the $t_i t_{i+1}$-interval in A, and so on, until the minimum in the suffix array range equals i. Since a single search step in the suffix array takes $O(\log n)$ time, the whole algorithm takes $O(n \log n)$ time.

12 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 7, and show how all suffix tree operations can be simulated by computations on *suffix array intervals*. Space-efficient data structures that facilitate these computations will be handled in subsequent chapters.

12.1 Basic Concepts

The reader is encouraged to recall the definitions from Sect. 3.4, in particular Def. 8. From now on, we regard the suffix tree as an abstract data type that supports the following operations.

Definition 24. *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. 7.6 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 25. *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 $H[k] = \text{LCP}(T^{A[k]}, T^{A[k-1]}) \geq |\overline{v}| = d(v)$ 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 25 corresponds to an internal node.

Definition 25. *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 25 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. 24; 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. 22 in Sect. 11.2. However, this time we define them to work on the LCP-array:

Definition 26. *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 26. *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 24 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 $w_r = r - 1$. $\square$

12.2 Suffix Tree Operations

We now step through the operations from Def. 24 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:

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

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

- $\textsc{LeafLabel}(v)$: If $v_\ell \neq v_r$, return $\textsc{null}$. Otherwise, return $A[v_\ell]$ in $O(t_{\text{SA}})$ time.
- $\textsc{SDepth}(v)$: 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 25 we know that $d(v)$ is the minimum $\textsc{lcp}$-value in $H[v_\ell + 1, v_r]$. We hence return $H[\textsc{rmq}_H(v_\ell + 1, v_r)]$ in time $O(t_{\text{RMQ}} + t_{\text{LCP}})$.
- $\textsc{Parent}(v)$: 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 26, return $[PSV(p), NSV(p) - 1]$. Time is $O(t_{\text{LCP}} + t_{\text{PNSV}})$.
- $\textsc{FirstChild}(v)$: If v is a leaf, return $\textsc{null}$. Otherwise, locate the first $d(v)$-value in $H[v_\ell, v_r]$ by $p = \textsc{rmq}_H(v_\ell + 1, v_r)$. Here, we assume that $\textsc{rmq}$ 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 $O(t_{\text{RMQ}})$.
- $\textsc{NextSibling}(v)$: First, compute v's parent as $w = \textsc{Parent}(v)$. Now, if $v_r = w_r$, return $\textsc{null}$, 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 $p = \textsc{rmq}_H(v_r + 2, w_r)$. 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 $O(t_{\text{LCP}} + t_{\text{PNSV}} + t_{\text{RMQ}})$.
- $\textsc{lca}(v, w)$: 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) $\textsc{lca}$ 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 $v_r \le w_\ell$, we have $\hat{u}_r \le \grave{u}_\ell$, and hence there must be a $d(u)$-index in H between $\hat{u}_r$ and $\grave{u}_\ell$, which can be found by $p = \textsc{rmq}_H(v_r + 1, w_\ell)$. The endpoints of u's interval are again located by $u_\ell = PSV(p)$ and $u_r = NSV(p) - 1$. Time is $O(t_{\text{RMQ}} + t_{\text{PNSV}})$.
- $\textsc{EdgeLabel}(v, i)$: First, compute the string-depth of v by $d_1 = \textsc{SDepth}(v)$, and that of $u = \textsc{Parent}(v)$ by $d_2 = \textsc{SDepth}(u)$, in total time $O(t_{\text{RMQ}} + t_{\text{LCP}} + t_{\text{PNSV}})$. Now if $i > d_1 - d_2$, return $\textsc{null}$, 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 $T_{A[k] + d_2 \ldots A[k] + d_1 - 1}$ 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.

12.3 Compressed LCP-Arrays

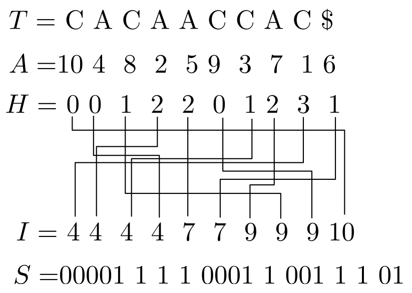
We now show how to reduce the space for the LCP-array H from $n \log n$ to $O(n)$ bits. To this end, we first note that the LCP-value can decrease by at most 1 when moving from suffix $A[i]-1$ to $A[i]$ in H (i. e., when enumerating the LCP-values in *text* order):

Lemma 27. *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 $I[1,n] = [H[A^{-1}[1]]+1, H[A^{-1}[2]]+2, H[A^{-1}[3]]+3, \ldots, H[A^{-1}[n]]+n]$ 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

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

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 28. *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.*

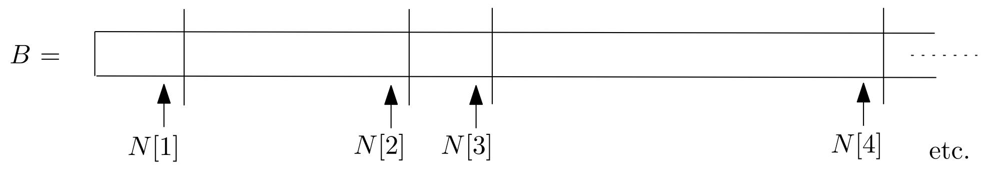
Note that with the sampled suffix array from Sect. 7.6, this means that we no more have constant-time access to H, as $t_{\text{SA}} = O(\log n)$ in this case.

12.4 Select in $o(n)$ Bits (*)

Definition 27. *Given a bit-vector $B[1, n]$, $select_1(B, i)$ returns the position of the i'th 1-bit in B, or $n + 1$ if B contains less than i 1's. Operation $select_0$ is defined similarly.*

Note that $\mathsf{rank}_1(B, select(B, i)) = i$. The converse $select(B, \mathsf{rank}(B, i))$ is only true if $B[i] = 1$. Note also that $select_0$ can*not* be computed easily from $select_1$ (as it was the case for rank), so $select_1$ and $select_0$ have to be considered separately.

Solving *select*-queries is only a little bit more complicated than solving rank-queries. We divide the *range* of arguments for $select_1$ into subranges of size $\kappa = \lfloor \log^2 n \rfloor$, and store in $N[i]$ the answer to $select_1(B, i\kappa)$. This table $N[1, \lceil \frac{n}{\kappa} \rceil]$ needs $O(\frac{n}{\kappa} \log n) = O(\frac{n}{\log n})$ bits, and divides B into *blocks* of different size, each containing κ 1's (apart from the last).



A block is called *long* if it spans more than $\kappa^2 = \Theta(\log^4 n)$ positions in B, and *short* otherwise. For the long blocks, we store the answers to all $select_1$-queries explicitly. Because there are at most $\frac{n}{\log^4 n}$ long blocks, this requires

$$O\left(\frac{n}{\kappa^2}\kappa \log n\right) = O(\underbrace{n/\log^4 n}_{\#\text{long blocks}} \times \underbrace{\log^2 n}_{\#\text{arguments}} \times \underbrace{\log n}_{\text{value from [1,n]}}) = O\left(\frac{n}{\log n}\right) = o(n) \text{ bits.}$$

Short blocks contain κ 1-bits and span at most κ^2 positions in B. We divide again their range of arguments into sub-ranges of size $\kappa' = \lfloor \log^2 \kappa \rfloor = \Theta(\log^2 \log n)$. In $N'[i]$, we store the answer to $select_1(B, i\kappa')$, relative to the beginning of the block where i occurs:

$$N'[i] = select_1(B, i\kappa') - N[\underbrace{\lfloor \frac{i\kappa' - 1}{\kappa} \rfloor}_{\text{block before } i}] .$$

Because the values in N' are in the range $[1, \kappa^2]$, table $N'[1, \lceil \frac{n}{\kappa'} \rceil]$ needs

$$O\left(\frac{n}{\kappa'} \log \kappa^2\right) = O\left(\frac{n}{\log^2 \log n} \log \log n\right) = o(n)$$

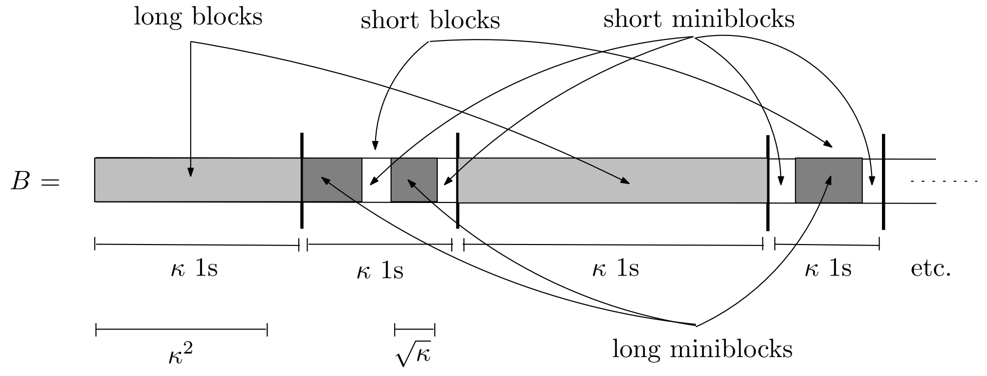
bits. Table N' divides the blocks into *miniblocks*, each containing κ' 1-bits.

Miniblocks are *long* if they span more than $\frac{\sqrt{\kappa}}{2} = \Theta(\log n)$ bits, and *short* otherwise. For long miniblocks, we store again the answers to all *select*-queries explicitly, relative to the beginning of the corresponding block. Because the miniblocks are contained in short blocks of length $\leq \kappa^2$, the answer to such a *select*-query takes $O(\log \kappa)$ bits of space. Thus, the total space for the long miniblocks is

$$O(\underbrace{n/\sqrt{\kappa}}_{\#\text{long miniblocks}} \times \underbrace{\kappa'}_{\#\text{arguments}} \times \log \kappa) = O\left(\frac{n \log^3 \log n}{\log n}\right) = o(n)$$

bits.

Finally, because short miniblocks are of length $\frac{\log n}{2}$, we can use a global lookup table (analogous to P in the solution for rank) to answer $select_1$-queries within short miniblocks.



Answering *select*-queries is done by following the description of the data structure.

The structures need to be duplicated for $select_0$. We summarize this section in the following theorem.

Theorem 29. *An n-bit vector B can be augmented with data structures of size $o(n)$ bits such that $\mathsf{rank}_b(B,i)$ and $select_b(B,i)$ can be answered in constant time ($b \in \{0,1\}$).*

13 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.

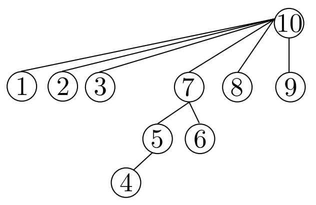
13.1 2-Dimensional Min-Heaps

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 28. *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 11.

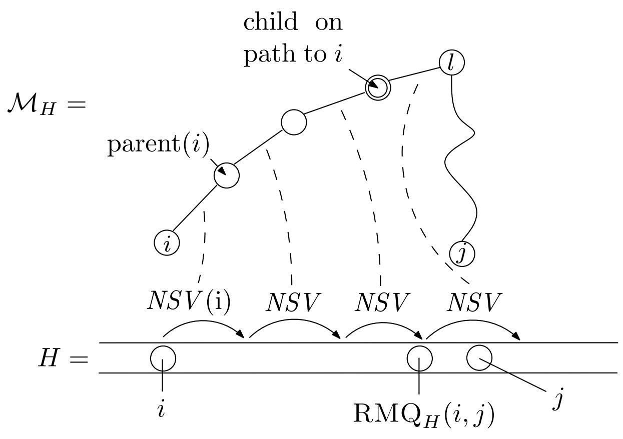


$$H = \begin{array}{cccccccccc} -1 & 0 & 0 & 3 & 1 & 2 & 0 & 1 & 1 & -\infty \\ \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 \end{array}$$

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 30. *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 12. *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, $\text{RMQ}_H(4, 6) = 5$.*

13.2 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 13. *Building on the 2d-Min-Heap from the Example 11, 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 $\mathsf{rank}_($(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).

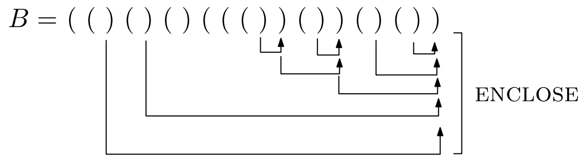
We also need the following operation.

Definition 29. *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 14.

$B = (\ (\ )\ (\ )\ (\ )\ (\ (\ (\ )\ )\ (\ )\ )\ (\ )\ (\ )\ )$



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

Theorem 31. *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

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

Example 15.

	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

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 32. *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.* □

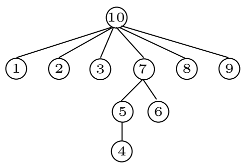
(The techniques are again similar to rank and *select*: *blocking* and *table-lookups.* Note in particular that $\frac{\log n}{2}$ excess-values $E[x], E[x+1], \ldots, E\left[x + \frac{\log n}{2} - 1\right]$ are encoded in a *single* computer-word $B\left[x, x + \frac{\log n}{2} - 1\right]$, and hence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

13.3 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 16.

$\mathcal{M}_H =$



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

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

$$j \leftarrow \mathsf{rank}_{)}(B, y) \ .$$

Example 17. *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* $j \leftarrow \mathsf{rank}_{)}(B, 20) = 10$*.*

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

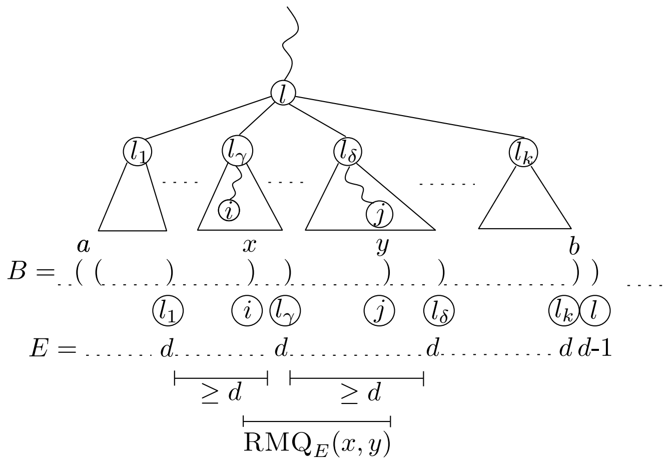
$$m \leftarrow \mathsf{rank}_{)}(B, z) \ .$$

This is the final answer.

Example 18. *We want to compute* $\text{RMQ}_H(4,9)$*. First, compute* $x \leftarrow select_{)}(B,4) = 11$ *and* $y \leftarrow enclose(B,9) = 19$*. The range minimum query yields* $z \leftarrow \text{RMQ}_E(11,19) = 15$*. Finally,* $m \leftarrow rank_{)}(B,15) = 7$ *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 $1 \le \gamma < \delta \le k$. By Lemma 30, 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 19.



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.

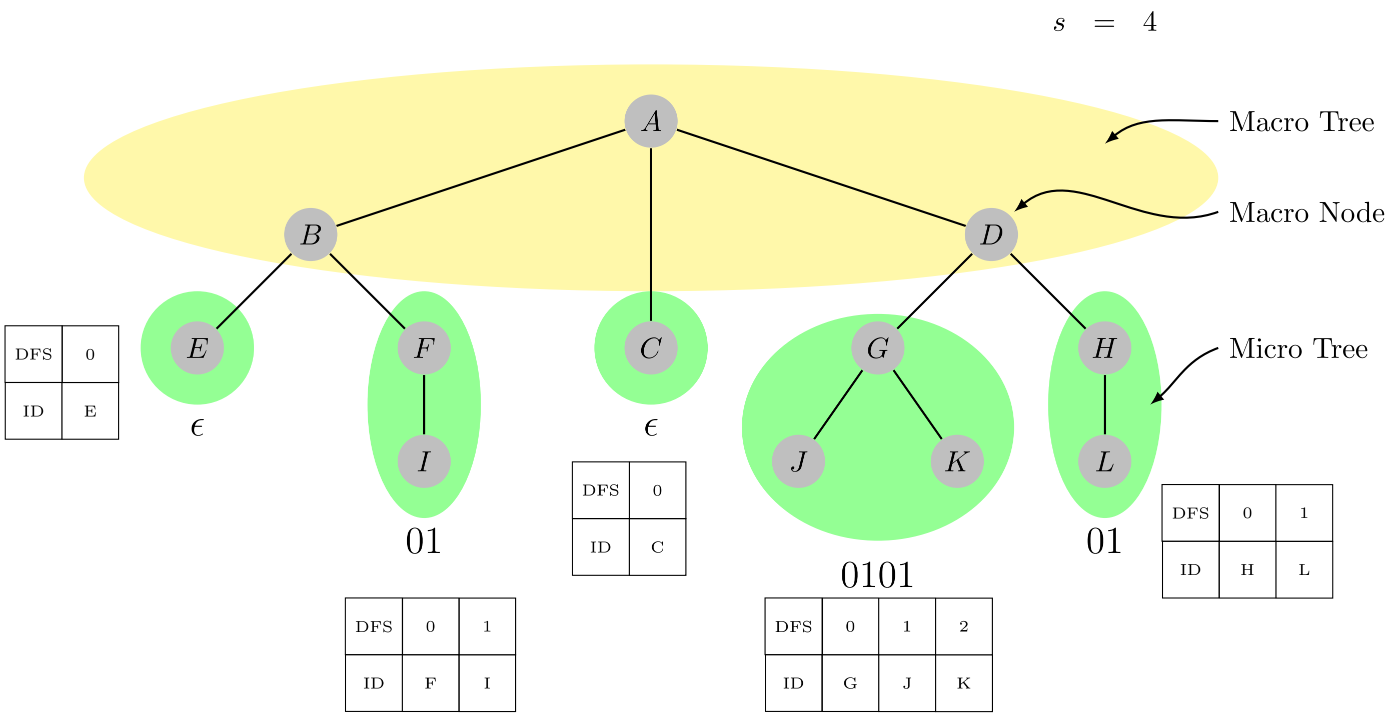
Because $B[a]$ is the opening parenthesis of node ℓ, we have $E[a] = d$. Further, because B is balanced, we have $E[c] \ge d$ for all $a < c < b$. But E assumes the values d at the positions of the closing parenthesis of nodes ℓ_β $(1 \le \beta \le k)$, in particular for ℓ_γ. Hence, the leftmost minimum 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:

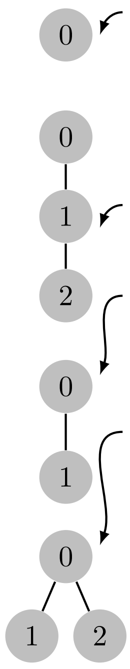
Theorem 33. *With a data structure of size* $2n + o(n)$ *bits, we can answer* RMQ*s and NSV-queries on an array of* n *ordered objects on* $O(1)$ *time.* □

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.)

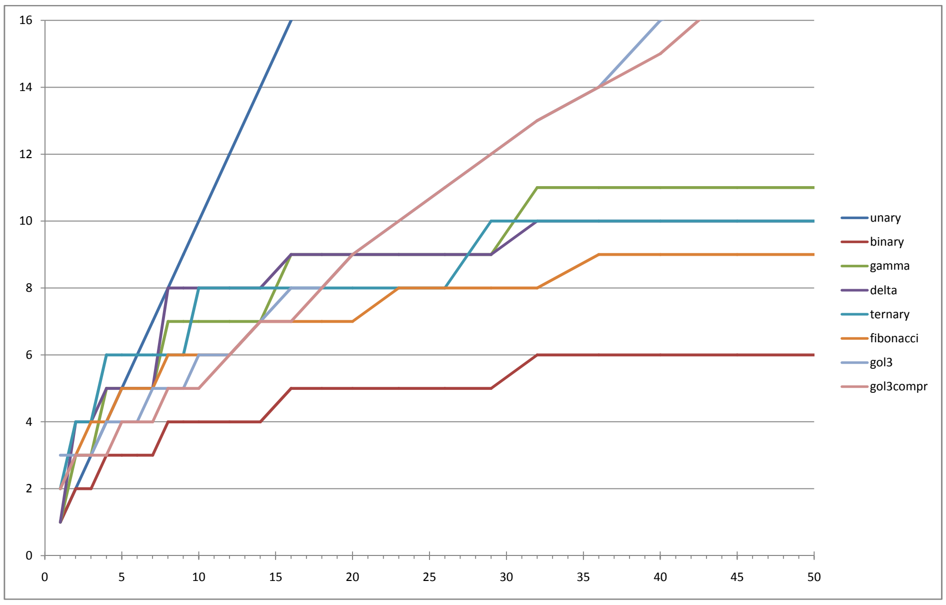
In the lecture we also discussed the possibility to just add another bit-vector of length n bits — however, this seems only to work if we represent the 2d-Min-Heap by DFUDS (instead of BPS). If we plug all these structures into the compressed suffix tree from Chapter 12 (which was indeed the reason for developing the solutions for RMQs and PNSVs), we get:

Theorem 34. *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 $O(t_{\mathrm{SA}})$ time (where t_{SA} denotes the time to retrieve an element from the suffix array).* □

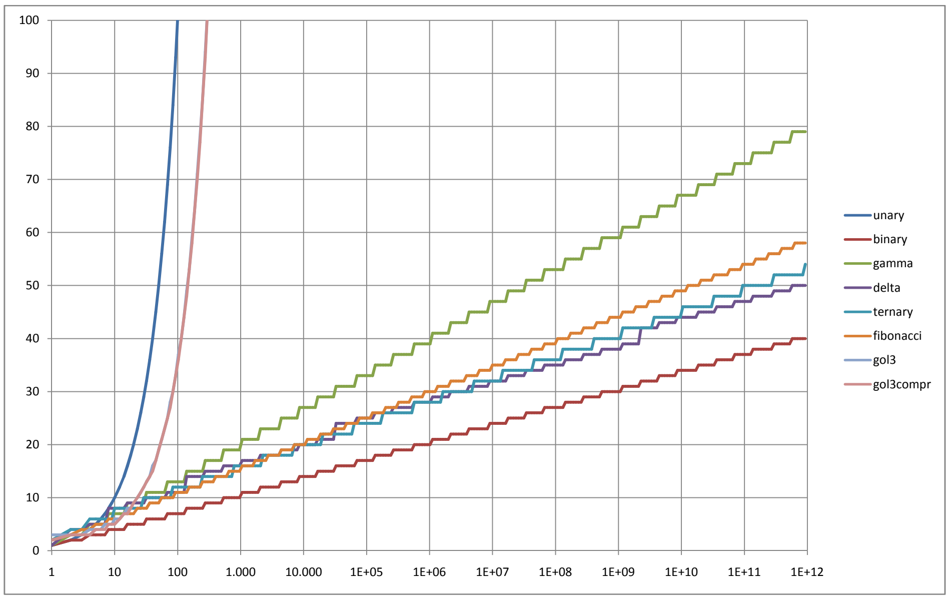




	$u =$	0	1	2	
bit pattern	$d =$	0 1 2	0 1 2	0 1 2	
000000		$0\bot\bot$	$\bot\bot\bot$	$\bot\bot\bot$	
000001		? ? ?	? ? ?	? ? ?	keine Micro Trees
000010		? ? ?	? ? ?	? ? ?	
⋮		?	?	?	
001100		$0\bot\bot$	$01\bot$	012	
⋮		?	?	?	keine Micro Trees
010000		$0\bot\bot$	$01\bot$	$\bot\bot\bot$	
010001		? ? ?	? ? ?	? ? ?	
010010		? ? ?	? ? ?	? ? ?	keine Micro Trees
010011		? ? ?	? ? ?	? ? ?	
010100		$0\bot\bot$	$01\bot$	$02\bot$	
⋮		?	?	?	
111111		? ? ?	? ? ?	? ? ?	



(a) Small numbers.



(b) Asymptotic Behavior.

Figure 4: Sizes of different codes for integers $x \geq 1$. ©Jens Quedenfeld 2014