Skriptum VL Text-Indexierung

Sommersemester 2010
Johannes Fischer (KIT)

1 Recommended Reading

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

2 Suffix Trees and Arrays

2.1 Suffix Trees

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

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

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

Definition 1. *A* compact Σ^+-tree *is a rooted tree $S = (V, E)$ with edge labels from Σ^+ that fulfills the following two constraints:*

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

Definition 2. *Let $S = (V, E)$ be a* compact Σ^+-tree.

- *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$(S) = \{\alpha \in \Sigma^* : S \text{ displays } \alpha\}$*
- *For $i \in \{1, 2, \dots, n\}$, $t_i t_{i+1} \dots t_n$ is called the i-*th suffix *of T and is denoted by $T_{i\dots n}$. In general, we use the notation $T_{i\dots j}$ as an abbreviation of $t_i t_{i+1} \dots t_j$.*

We are now ready to define suffix trees.

Definition 3. *Let factor(T) denote the set of all factors of T, factor(T) = $\{T_{i...j} : 1 \leq i \leq j \leq n\}$. The* suffix tree *of T is a compact Σ^+-tree S with words(S) = factor(T).*

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

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

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

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

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

2.2 Searching in Suffix Trees

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

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

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

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

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

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

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

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

2.3 Suffix- and LCP-Arrays

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

Definition 4. *The* suffix array A *of* T *is a permutation of* $\{1, 2, \ldots, n\}$ *such that* $A[i]$ *is the* i*-th smallest suffix in lexicographic order:* $T_{A[i-1]...n} < T_{A[i]...n}$ *for all* $1 < i \leq n$.

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

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

The second array H builds on the suffix array:

Definition 5. *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* $T_{A[i]...n}$ *and* $T_{A[i-1]...n}$.

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

Definition 6. *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:* $\forall i > 1 : H[i] = d(\text{LCA}(\overline{T_{A[i]...n}}, \overline{T_{A[i-1]...n}}))$.

2.4 Searching in Suffix Arrays

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

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

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

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

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

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

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

The idea of the algorithm is to insert the suffixes into S in the order of the suffix array: $T_{A[1]\ldots n}, T_{A[2]\ldots n}, \ldots, T_{A[n]\ldots n}$. 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 $T_{A[i+1]+H[i+1]\ldots n}$. Leaf x is labeled by $A[i+1]$. This gives us S_{i+1}.

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

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

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

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

2.6 Linear-Time Construction of Suffix Arrays

For an easier presentation, we assume in this section that T is index from 0 to $n-1$, $T = t_0 t_1 \dots t_{n-1}$.

Definition 7. *If $x \bmod y = z$, we write $x \equiv z (\bmod\ y)$ or simply $x \equiv z(y)$.*

Our general approach will be *recursive*: first construct the suffix array A_{12} for the suffixes $T_{i\dots n}$ with $i \not\equiv 0(3)$ by a recursive call to the suffix sorting routine. From this, derive the suffix array A_0 for the suffixes $T_{i\dots n}$ with $i \equiv 0(3)$. Then merge A_{12} and A_0 to obtain the final suffix array.

2.6.1 Creation of A_{12} by Recursion

In order to call the suffix sorting routine recursively, we need to construct a new text T' from whose suffix array A' we can derive A_{12}. To this end, we look at all *character triplets* $t_i t_{i+1} t_{i+2}$ with $i \not\equiv 0(3)$, and sort the resulting set of triplets $S = \{t_i t_{i+1} t_{i+2} : i \not\equiv 0(3)\}$ with a bucket-sort in $O(n)$ time. (To have all triplets well-defined, we pad T with sufficiently many \$'s at the end.)

We define T' as follows: its first half consists of the bucket-numbers of $t_i t_{i+1} t_{i+2}$ for increasing $i \equiv 1(3)$, and its second half consists of the same for the triplets with $i \equiv 2(3)$.

We now build the suffix array A' for T' by a recursive call (stop the recursion if $|T'| = O(1)$). This already gives us sorting of the suffixes starting at positions $i \not\equiv 0(3)$, because of the following:

- The suffixes starting at $i \equiv 2(3)$ in T have a one-to-one correspondence to the suffixes in T' and are hence in correct lexicographic order.
- The suffixes starting at $i \equiv 1(3)$ in T are longer than they should be (because of the bucket numbers of the triplets starting at 2(3)), but due to the \$'s in the middle of T' this "tail" does not influence the result.

Thus, all that remains to be done is to convert the indices in T' to indices in T, which is done as follows:

$$A_{12}[i] = \begin{cases} 1 + 3A'[i] & \text{if } A'[i] < \lceil \frac{|T'|}{2} \rceil \\ 2 + 3(A'[i] - \lceil \frac{|T'|}{2} \rceil) & \text{otherwise} \end{cases}$$

2.6.2 Creation of A_0

The following lemma follows immediately from the definition of A_{12}.

Lemma 5. *Let $i, j \equiv 0(3)$. Then $T_{i\dots n} < T_{j\dots n}$ iff $t_i < t_j$, or $t_i = t_j$ and $i+1$ appears before $j+1$ in A_{12}.*

This suggests the following strategy to construct A_0:

1. Initialize A_0 with all numbers $0 \leq i < n$ for which $i \equiv 0(3)$.
2. Bucket-sort A_0, where $A_0[i]$ has sort-key $t_{A_0[i]}$.
3. In a left-to-right scan of A_{12}: if $A_{12}[i] \equiv 1(3)$, move $A_{12}[i] - 1$ to the current beginning of its bucket.

2.6.3 Merging A_{12} with A_0

We scan A_0 and A_{12} simultaneously. The suffixes from A_0 and A_{12} can be compared among each other in $O(1)$ time by the following lemma, which follows again directly from the definition of A_{12}.

Lemma 6. *Let $i \equiv 0(3)$.*

1. *If $j \equiv 1(3)$, then $T_{i...n} < T_{j...n}$ iff $t_i < t_j$, or $t_i = t_j$ and $i+1$ appears before $j+1$ in A_{12}.*
2. *If $j \equiv 2(3)$, then $T_{i...n} < T_{j...n}$ iff $t_i < t_j$, or $t_i = t_j$ and $t_{i+1} < t_{j+1}$, or $t_i t_{i+1} = t_j t_{j+1}$ and $i+2$ appears before $j+2$ in A_{12}.*

One should note that in both of the above cases the values $i+1$ and $j+1$ (or $i+2$ and $j+2$, respectively) appear in A_{12} — this is why it is enough to compare at most 2 characters before one can derive the lexicographic order of $T_{i...n}$ and $T_{j...n}$ from A_{12}.

In order to check efficiently whether i appears before j in A_{12}, we need the *inverse suffix array* A_{12}^{-1} of A_{12}, defined by $A_{12}^{-1}[A_{12}[i]] = i$ for all i. With this, it is easy to see that i appears before j in A_{12} iff $A_{12}^{-1}[i] < A_{12}^{-1}[j]$.

The running time $\mathcal{T}(n)$ of the whole suffix sorting algorithm presented in this section is given by the recursion $\mathcal{T}(n) = \mathcal{T}(2n/3) + O(n)$, which solves to $\mathcal{T}(n) = O(n)$.

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

2.7 Linear-Time Construction of LCP-Arrays

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, A, and A^{-1}, the latter being the inverse suffix array.

We will construct H *in the order of the inverse suffix array* (i.e., 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 $T_{i...n}$ to $T_{i+1...n}$, we see that the latter equals the former, but with the first character t_i truncated. Let $h = H[i]$. Then the suffix $T_{j...n}$, $j = A[A^{-1}[i]-1]$, has a longest common prefix with $T_{i...n}$ of length h. So $T_{i+1...n}$ has a longest common prefix with $T_{j+1...n}$ of length $h-1$. But every suffix $T_{k...n}$ that is lexicographically between $T_{j+1...n}$ and $T_{i+1...n}$ must have a longest common prefix with $T_{j+1...n}$ that is at least $h-1$ characters long (for otherwise $T_{k...n}$ would not be in lexicographic order). We have thus proved the following:

Lemma 8. *For all $1 \leq i < n$: $H[A^{-1}[i+1]] \geq H[A^{-1}[i]] - 1$.*

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

Algorithm 2: Linear-Time Construction of the LCP-Array

```
1 h ← 0, H[1] ← 0
2 for i = 1, ..., n do
3 |   if A^{-1}[i] ≠ 1 then
4 |   |   while t_{i+h} = t_{A[A^{-1}[i]-1]+h} do h ← h + 1
5 |   |   H[A^{-1}[i]] ← h
6 |   |   h ← max{0, h − 1}
7 |   end
8 end
```

The linear running time follows because h is always less than n and decreased at most n times in line 6. Hence, the number of times where k is increased in line 4 is bounded by n, so there are at most $2n$ character comparisons in the whole algorithm. We have proved:

Theorem 9. *The LCP-array for a text of length n can be constructed in $O(n)$ time.*

3 Repeats

Definition 8. *Let $T = t_1 t_2 \dots t_n$ be a text of length n. A triple (i, h, ℓ) with $1 \le i < j \le n - \ell + 1$ is called* repeat *if $T_{i \dots i+\ell-1} = T_{j \dots j+\ell-1}$.*

We look at three different tasks:

1. Output all triples (i, j, ℓ) that are a repeat according to Def. 8.
2. Output all strings $\alpha \in \Sigma^\star$ such that there is a repeat (i, j, ℓ) with $T_{i \dots i+\ell-1} = \alpha (= T_{j \dots j+\ell-1})$.
3. Like (2), but instead of outputting α just output a pair (l, r) with $\alpha = T_{l \dots r}$.

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

3.1 Maximal Repeats

Definition 9. *A repeat (i, j, ℓ) is called*

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

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

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

Let

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

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

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

Definition 10. *Let S be T's suffix tree. A node v in S is called* left-diverse *if there are at least two leaves b_1 and b_2 below v such that $t_{\ell(b_1)-1} \neq t_{\ell(b_2)-1}$. [Recall that $\ell(\cdot)$ denotes the leaf label (=suffix number)]. Character $t_{\ell(v)-1}$ is called v's* left-character*.*

Lemma 10. *A repeat (i, j, ℓ) is maximal iff there is left-diverse node v in T's suffix tree S with $\overline{v} = T_{i\ldots i+\ell-1}$.*

Proof. It remains to care about left-maximality.

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

"⇐" analogous. □

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

In a depth-first search through S do:

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

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

3.2 Super-Maximal Repeats

The maximal repeats in Def. 9 can still contain other maximal repeats, as the example $T =$ `axybxxyyyaxyb` shows (both `xy` and `axyb` are maximal repeats, for example). This is prevented by the following definition:

Definition 11. *A maximal repeat (i, j, ℓ) is called* super-maximal *if there is no maximal repeat (i', j', ℓ') such that $T_{i\ldots i+\ell-1}$ is a proper subword of $T_{i'\ldots i'+\ell-1}$.*

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

3.3 Longest Common Substrings

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

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

4 Tandem Repeats and Related Repetitive Structures

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

4.1 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 Vorlesung. They ask for the position of the minimum element in a specified sub-array, formally defined as follows.

Definition 12. *Given an array $H[1, n]$ of n integers (or any other objects from a totally ordered universe) and two indices $1 \le i \le j \le n$, $\text{RMQ}_H(i, j)$ is defined as the position of the minimum in H's sub-array ranging from i to j, in symbols: $\text{RMQ}_H(i, j) = \text{argmin}_{i \le k \le j} H[k]$.*

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 at a later point that this naive approach can be dramatically improved, as the following proposition anticipates:

Proposition 11. *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.*

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

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

Note also that the LCP-array H from Sect. 2.3 holds the lengths of longest common prefixes of *lexicographically consecutive suffixes*: $H[i] = \text{LCP}(A[i], A[i-1])$. 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 12. *Let $i \neq j$ be two indices in T with $A^{-1}[i] < A^{-1}[j]$ (otherwise swap i and j). Then* $\text{LCP}(i,j) = H[\text{RMQ}_H(A^{-1}[i]+1, A^{-1}[j])]$.

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

Now assume that $T_{i\ldots i+\ell}$ is also a common prefix of $T_{i\ldots n}$ and $T_{j\ldots n}$. Then, by the lexicographic order of A, $T_{i\ldots i+\ell}$ is also a common prefix of $T_{A[m-1]\ldots n}$ and $T_{A[m]\ldots n}$. But $|T_{i\ldots i+\ell}| = \ell + 1$, contradicting the fact that $H[m] = \ell$ tells us that $T_{A[m-1]\ldots n}$ and $T_{A[m]\ldots n}$ 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 14. *For a text T of length n and two indices $1 \leq i, j \leq n$, $\text{LCS}_T(i,j)$ denotes the length of the* longest common suffix *of the prefixes ending at position i and j in T, in symbols:* $\text{LCS}_T(i,j) = \max\{k \geq 0 : T_{i-k+1\ldots i} = T_{j-k+1\ldots j}\}$.

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

4.3 Tandem Repeats and Runs

Definition 15. *A string $S \in \Sigma^*$ that can be written as $S = \omega^k$ for $\omega \in \Sigma^+$ and $k \geq 2$ is called a* tandem repeat.

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

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

Definition 16. *Given indices b and e with $1 \leq b \leq e \leq n$ and an integer $\ell \geq 1$, the triple (b, e, ℓ) is called a* run *in T if*

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

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

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

The following lemma, which is sometimes taken as the definition of runs, follows easily from Definition 16.

Lemma 13. *Let (b, e, ℓ) be a run in T. Then $t_{j-\ell} = t_j$ for all $b + \ell \leq j \leq e$.*

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

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

Proposition 14. *The number of runs in a text of length n is in $O(n)$.*

We now give a first hint at how we can use our previous knowledge to tackle the computation of all runs. From this point onwards, we will not require the period of a run be minimal (fourth point in Definition 16) — this condition can be easily incorporated into the whole algorithm.

Lemma 15. *For $\ell \leq \lfloor \frac{n}{2} \rfloor$ and an index j with $\ell < j \leq n - \ell + 1$, let $s = \text{LCS}(j-1, j-\ell-1)$ and $p = \text{LCP}(j, j-\ell)$. Then the following three statements are equivalent:*

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

Proof. (1)⇒(2): Let (b, e, ℓ) be the run. We decompose $T_{b...e}$ into three strings u, v, w with $u = T_{b...j-\ell-1}$, $v = T_{j-\ell...j-1}$, and $w = T_{j...e}$, so that $T_{b...e} = uvw$ (u and w are possibly empty). Due to Lemma 13, we have that $|u| = s$ and $|w| = p$. Because $|v| = \ell$ by construction and $|T_{b...e}| = e - b + 1$, we get

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

where the last inequality follows from the "$k \geq 2$" in Definition 16. Hence, $s + p \geq \ell$.
(2)⇒(3): Let $s + p \geq \ell$. First note that due to the definition of lcp/lcs, we get that

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

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

4.4 Algorithm

Lemma 15 gives rise to a first algorithm for computing all the runs in T: simply check for every possible period $\ell \leq \lfloor \frac{n}{2} \rfloor$ and every j if there is a run (b, e, ℓ) with $b+\ell \leq j \leq e+1$, using the lemma.

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

Algorithm 3: $O(n \log n)$-algorithms for locating all runs in a string

```
1 for ℓ = 1, ..., ⌊n/2⌋ do
2     j ← 2ℓ + 1
3     while j ≤ n + 1 do
4         s ← LCS(j − 1, j − ℓ − 1)
5         p ← LCP(j, j − ℓ)
6         if s + p ≥ ℓ and p < ℓ then output the run (j − ℓ − s, j + p − 1, ℓ)
7         j ← j + ℓ
8     end
9 end
```

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

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

Hence, we get:

Theorem 16. *We can find all runs in a string of length n in $O(n \log n)$ time.*

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

Proposition 17. *We can find all runs in a string of length n in $O(n)$ time.*

5 Lowest Common Ancestors and Range Minimum Queries

5.1 Reducing Lowest Common Ancestors to Range Minimum Queries

Recall the definition of range minimum queries (RMQs) from the chapter on tandem repeats: $\text{RMQ}_D(l,r) = \text{argmin}_{l \le k \le r} D[k]$ for an array $D[1,n]$ and two indices $1 \le l \le r \le n$. 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 17. *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 $|I| = 1 < 2n$.

$\le n \to n+1$: 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,

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

Here comes the desired connection between LCA and RMQ:

Lemma 19. *For any pair of nodes v and w in T,* $\text{LCA}_T(v,w) = I[\text{RMQ}_D(p_v, p_w)]$.

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

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

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

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

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.

5.2 $O(1)$-RMQs with $O(n \log n)$ Space

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

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

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

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.

5.3 $O(1)$-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).

5.3.1 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. 5.2, 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 $W[\text{RMQ}_{D'}(i + 1, j - 1)]$.

5.3.2 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. 5.3.1.

Definition 18. *Let $B_j[1, s]$ be a block of size s. The* Cartesian Tree $\mathcal{C}(B_j)$ *of B_j is a labelled binary tree, recursively defined as follows:*

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

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

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

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 B_j is not larger than $B_j[i + 1]$:

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

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 B_i and B_j be two blocks with equal Cartesian Trees. Then $\text{RMQ}_{B_i}(l,r) = \text{RMQ}_{B_j}(l,r)$ for all $1 \leq l \leq r \leq s$.*

Proof. By induction on s.

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

$\leq s \to s+1$: Let v be the root of $\mathcal{C}(B_i) = \mathcal{C}(B_j)$ with label μ. By the definition of the Cartesian Tree,

$$\operatorname*{argmin}_{1 \leq k \leq s} B_i[k] = \mu = \operatorname*{argmin}_{1 \leq k \leq s} B_j[k] \ . \tag{1}$$

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

$$\text{RMQ}_{B_i}(l,r) = \text{RMQ}_{B_j}(l,r) \forall 1 \leq l \leq r < \mu, \text{and } \text{RMQ}_{B_i}(l,r) = \text{RMQ}_{B_j}(l,r) \forall \mu < l \leq r \leq s. \tag{2}$$

In total, we see that $\text{RMQ}_{B_i}(l,r) = \text{RMQ}_{B_j}(l,r)$ for all $1 \leq l \leq r \leq s$, 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). □

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

Lemma 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:

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

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.

6 Lempel-Ziv Compression

6.1 Longest Previous Substring

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

Definition 19. *The* longest-previous-substring-array $L[1, n]$ *is defined such that* $L[i] = \max\{\ell \geq 0 : \exists k < i \text{ with } T_{i\ldots i+\ell-1} = T_{k\ldots k+\ell-1}\}$.

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

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

Definition 20. *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 22. *For all* $2 \leq i \leq n$*:* $L[i] = \max\{\text{LCP}(i, j) : 1 \leq j < i\}$. □

For convenience, from now on we assume that both A and H are padded with 0's at their beginning and end: $A[0] = H[0] = A[n+1] = H[n+1] = 0$. We further define $T_{0\ldots n}$ to be the empty string ϵ.

Definition 21. *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. The next lemma shows how PSVs/NSVs can be used to compute L efficiently:

Lemma 23. *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 22 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

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

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 $T_{A[j]\ldots n}$ and $T_{A[i]\ldots n}$ is also a prefix of $T_{A[PSV(i)]\ldots n}$. 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 12.
- Build L by applying Lemma 23 to all positions i.

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

6.2 Lempel-Ziv Factorization

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

Definition 22. *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 4: $O(n)$-computation of the LZ-factorization

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

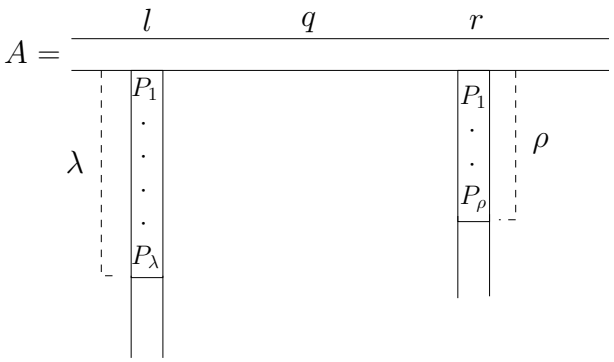
7 Faster Search with Suffix Trees and Arrays

7.1 Accelerated Search in Suffix Arrays

The simple binary search (Alg. 1) 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,

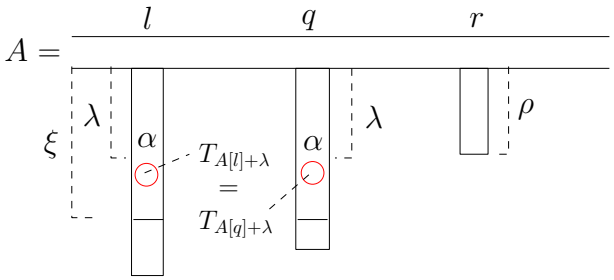
$$\lambda = \text{LCP}(P, T_{A[l]\ldots n}) \text{ and } \rho = \text{LCP}(P, T_{A[r]\ldots n}).$$

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. 1 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:



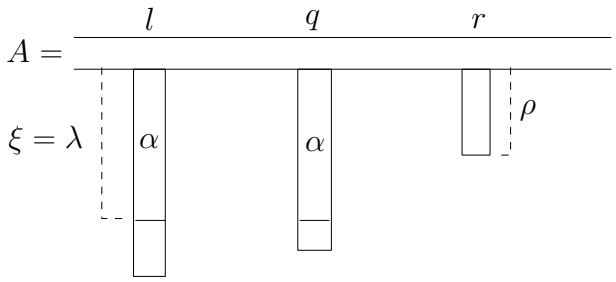
Without loss of generality, assume $\lambda \geq \rho$ (otherwise swap). We then look up $\xi = \text{LCP}(A[l], A[q])$ as the longest common prefix of the suffixes $T_{A[l]\ldots n}$ and $T_{A[q]\ldots n}$. We look at three different cases:

1. $\xi > \lambda$



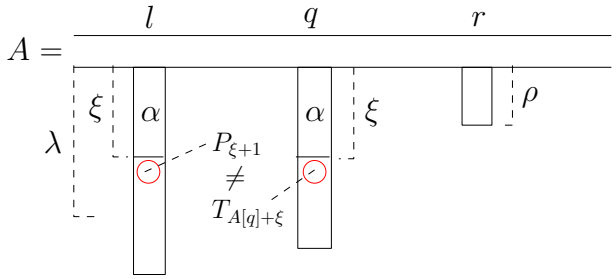
Because $P_{\lambda+1} >_{\text{lex}} T_{A[l]+\lambda} = T_{A[q]+\lambda}$, we know that $P >_{\text{lex}} T_{A[q]\ldots n}$, and can hence set $l \leftarrow q$, and continue the search *without any character comparison*. Note that ρ and in particular λ correctly remain unchanged.

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 $T_{A[q]\ldots n}$ lies lexicographically between $T_{A[l]\ldots n}$ and $T_{A[r]\ldots n}$. 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 24. *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).*

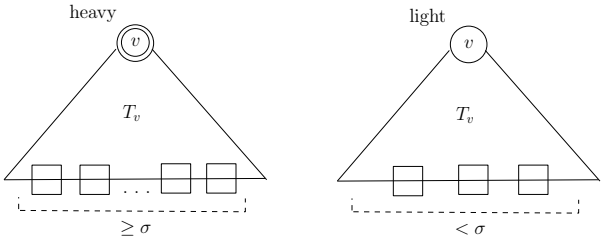
7.2 Suffix Trays

We now show how the $O(m \log \sigma)$-algorithm for suffix trees and the $O(m + \log n)$-algorithm can be combined to obtain a faster $O(m + \log \sigma)$ search-algorithm. The general idea is to start the search in

the suffix tree where some additional information has been stored to speed up the search, and then, at an "appropriate" point, continue the search in the suffix array with a sufficiently small interval. Note that the accelerated search-algorithm from Sect. 7.1, if executed on a sub-interval $I = A[x, y]$ of A instead of the complete array $A[1, n]$, runs in $O(m + \log |I|)$ time. This is $O(m + \log \sigma)$ for $|I| = \sigma^{O(1)}$.

We first classify the nodes in T's suffix tree S as follows:

1. Node v is called *heavy* if the numbers of leaves below v is at least σ.
2. Otherwise, node v is called *light*.



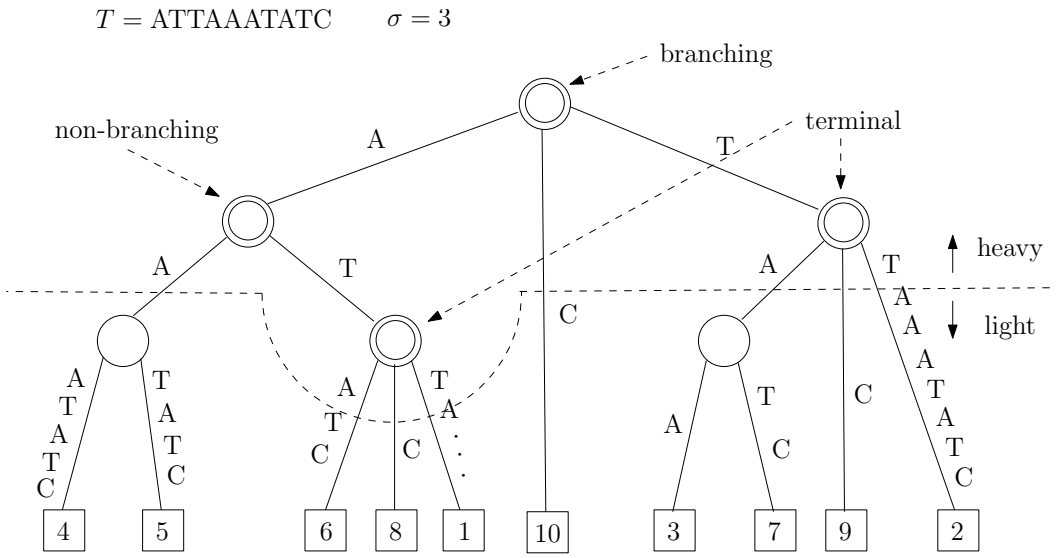
Note that all heavy nodes in S have only heavy ancestors by definition, and hence the heavy nodes form a connected subtree of S. Heavy nodes v are further classified into

(a) *branching*, if at least two of v's children are heavy,

(b) *non-branching*, if exactly one of v's children is heavy,

(c) *terminal*, if none of v's children are heavy.

Example 1.



Lemma 25. *The number of branching heavy nodes is* $O(\frac{n}{\sigma})$.

Proof: First count the *terminal* heavy nodes. By definition, every heavy node has $\geq \sigma$ leaves below itself. Note that *every* leaf in S must be below *exactly one* terminal heavy node. For the sake of contradiction, suppose there were more then $\frac{n}{\sigma}$ terminal heavy nodes. Then the tree S would contain $> \frac{n}{\sigma}\sigma = n$ leaves. Contradiction! Hence, the number of terminal heavy nodes is at most $\frac{n}{\sigma}$.

Now look at the subtree of S consisting of terminal and branching heavy nodes only. This is a tree with at most $\frac{n}{\sigma}$ leaves, and every internal node has at least two children. For such a tree we know that the number of internal nodes is bounded by the number of leaves. The claim follows. □

We now augment the suffix tree with additional information at every heavy node v:

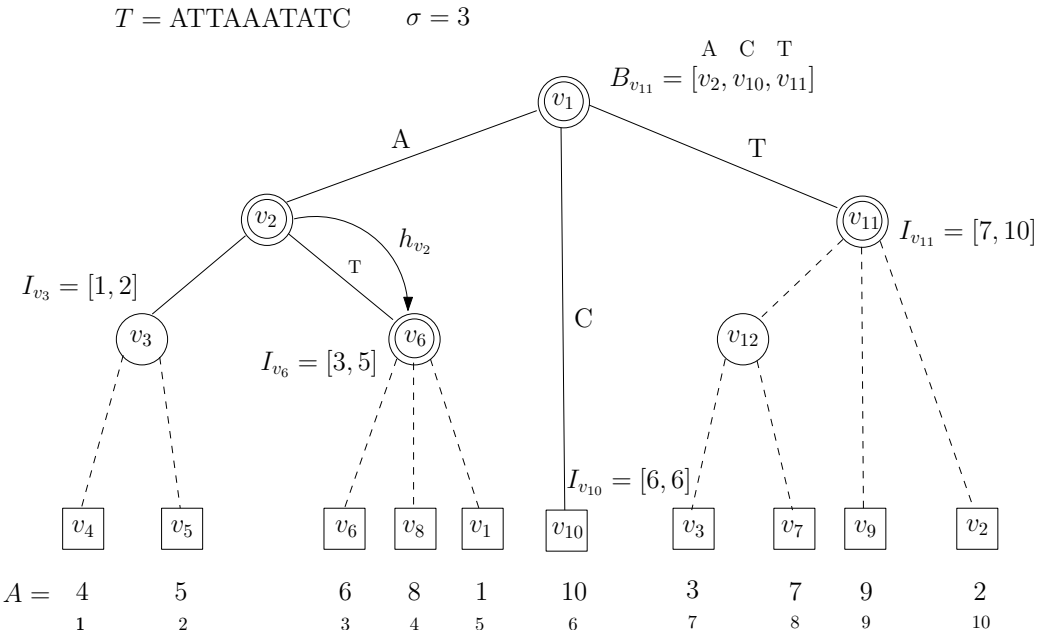
(a) At every branching heavy node v, we store an array B_v of size σ, where we store pointers to v's children according to the first character on the corresponding edge (like in the "bad" suffix tree with $O(n\sigma)$ space!).

(b) At every non-branching heavy node v, we store a pointer h_v to its single heavy child.

(c) All terminal heavy nodes, and all light children of branching or non-branching heavy nodes are called *interval nodes*. Every interval node v is augmented with its suffix-interval I_v, which contains the start- and end-position in the suffix array A of the leaf-labels below v. Furthermore, the suffix-intervals of adjacent light children of a non-branching heavy node are contracted into one single interval. Thus, non-branching heavy nodes v have at most 3 children (one heavy child h_v, and one interval node each to the left/right of h_v).

Everything in the tree below the interval nodes can be deleted. The resulting tree, together with the suffix array A, is called the *suffix tray*.

Example 2. *The suffix tray for* $T = \text{ATTAAATATC}$ *is shown in the following picture (the dotted part has been deleted from the tree):*



Lemma 26. *The size of the resulting data structure is* $O(n)$.

Proof: From Lemma 25 we know that there are only $O(\frac{n}{\sigma})$ heavy branching nodes. So the total space for heavy branching nodes is $O(\frac{n}{\sigma}\sigma) = O(n)$. All other information is constant at each node. The claim follows. □

Now we describe the *search* procedure. We start as with normal suffix trees and try reading P from the root downwards. There are three different cases to consider (assume the characters $P_{1\ldots i-1}$ have already been matched, and we have arrived at node $v = \overline{P_{1\ldots i-1}}$ of the suffix tray):

(a) At branching heavy nodes v, we can find the the correctly labeled edge (i. e., the edge whose label starts with P_i) in $O(1)$ time, by consulting $B_v[P_i]$.

(b) At non-branching heavy nodes v, we first check if the search continues at the heavy child, by comparing the next character P_i with the first character $a \in \Sigma$ on the incoming edge (v, h_v). If this is the case ($P_i = a$), we compare all characters on the edge (v, h_v) to the pattern and, if necessary, continue the search procedure at the heavy child h_v. Otherwise ($P_i \neq a$), there are two possibilities. If $P_i < a$, we continue at the (only) interval node left of h_v with case (c). If $P_i > a$, we do the same for the only interval node right of h_v with case (c).

(c) At interval nodes v, we switch to the suffix array search algorithm from section 1.8, using I_v as the start interval.

Lemma 27. *The length of the intervals stored at the interval nodes is $O(\sigma^2)$.*

Proof: The intervals of at most $\sigma - 1$ light nodes are contracted into a single interval, and each light node has at most $\sigma - 1$ leaves below itself. □

Theorem 28. *The suffix tray supports counting and reporting queries in $O(m + \log \sigma)$ and $O(m + \log \sigma + |O_P|)$ time, respectively.*

Proof: We either advance by one character in P with a constant amount of work, or we arrive at an interval node v, where we perform the accelerated binary search in $O(m + \log |I_v|) = O(m + \log \sigma^2) = O(m + 2 \log \sigma) = O(m + \log \sigma)$ time, by Lemma 27. □

8 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*. In this Chapter, we introduce the transformation and briefly review its merits for compression. The subsequent chapter on backwards search will then explain how it is used in the indexing scenario.

8.1 The Transformation

Definition 23. *Let $T_{1\ldots n}$ be a text of length n, where $T_n = \$$ is a unique character lexicographically smaller than all other characters in Σ. Then the i-th* cyclic shift *of T is $T_{i\ldots n}T_{1\ldots i-1}$. We denote it by $T^{(i)}$.*

Example 3.

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

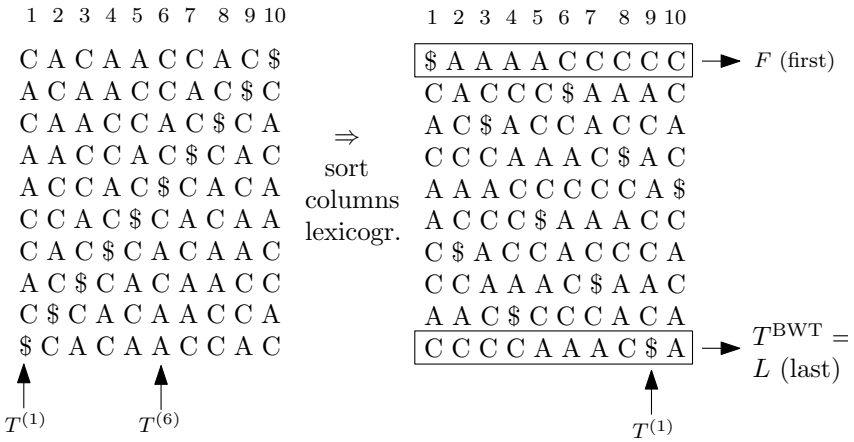
$$T^{(6)} = \text{CCAC\$CACAA}$$

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

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

Example 4.

$T =$ CACAACCAC\$



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.

8.2 Construction of the BWT

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

$$T_i^{\text{BWT}} = T_{A[i]-1} (= T_n^{(A[i])}) \ ,$$

where A denotes again T's suffix array, and T_0 is defined to be T_n (read T cyclically!). Because the suffix array can be constructed in linear time (Thm. 7), we get:

Theorem 29. *The BWT of a text length-n text over an integer alphabet can be constructed in $O(n)$ time.* □

Example 5.

$T = \text{CACAACCAC\$}$

	1	2	3	4	5	6	7	8	9	10	
A=	10	4	8	2	5	9	3	7	1	6	
	\$	A	A	A	A	C	C	C	C	C	→ F (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	→ T^{BWT} = L (last)

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

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

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

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

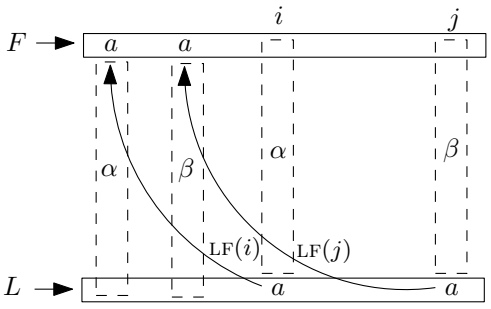
Example 6.

$T = \text{CACAACCAC\$}$

1	2	3	4	5	6	7	8	9	10	
\$	A	A	A	A	C	C	C	C	C	→ F (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	→ T^{BWT} = L (last)

LF = 6 7 8 9 2 3 4 10 1 5

Observation 4. *Equal characters preserve the same order in F and L. That is, if $L[i] = L[j]$ and $i < j$, then $\text{LF}(i) < \text{LF}(j)$. To see why this is so, recall that the* BWT*-matrix is sorted lexicographically. Because both the $\text{LF}(i)$'th and the $\text{LF}(j)$'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 $\text{LF}(i) < \text{LF}(j)$.*



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

Definition 25. *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 $\text{OCC} : \Sigma \times [1, n] \to [1, n]$ such that $\text{OCC}(a, i)$ is the number of occurrences of a in L's length-i-prefix $L[1, i]$.*

Lemma 30. *With the definitions above,*

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

Proof: Follows immediately from the observation above. □

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

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

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

Example 7.

$$C = \overset{\$\ \ A\ \ C}{0\ 1\ 5}$$

$$F = \$\ A\ A\ A\ A\ C\ C\ C\ C\ C$$

$$L = C\ C\ C\ C\ A\ A\ A\ C\ \$\ A$$

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

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

T reversed

8.4 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 review two different compression methods in this section.

8.4.1 Move-to-front (MTF) & Huffman Coding

- Initialize a list Y containing each character in Σ in alphabetic order.
- In a left-to-right scan of $T^{\mathrm{BWT}}, (i = 1, \ldots, n)$, 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.
- Encode the resulting string/array R with any kind of reversible compressor, e. g. Huffman, into a string R.

Example 8.

T^{BWT} =C C C C A A A C \$ A

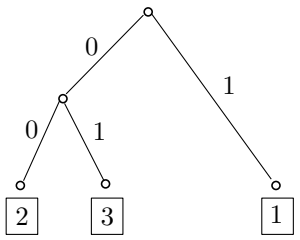
i	Y_{old}	T_i^{BWT}	$R[i]$	Y_{new}
1	\$AC	C	3	CA\$
2	CA\$	C	1	CA\$
3	CA\$	C	1	CA\$
4	CA\$	C	1	CA\$
5	CA\$	A	2	AC\$
6	AC\$	A	1	AC\$
7	AC\$	A	1	AC\$
8	AC\$	C	2	CA\$
9	CA\$	\$	3	\$CA
10	\$CA	A	3	A\$C

Observation 5. *MTF produces "many small" numbers for equal characters that are "close together" in* T^{BWT}*. These can be compressed using an order-0 compressor, e. g. Huffman, as in the next example.*

Example 9.

character	frequency
1	5
2	2
3	3

$\Rightarrow$



$R[i]$
3
1
1
1
2
1
1
2
3
3

$\Rightarrow$ $C = 011110011000101$

Both steps (Huffman & *MTF*) are easy to reverse.

8.4.2 Run-Length Encoding

We can also 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 $a \in \Sigma, \ell \in [1, n]$.

Example 10.

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

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

9 Backwards Search and *FM*-Indices

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

9.1 Recommended Reading

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

9.2 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.

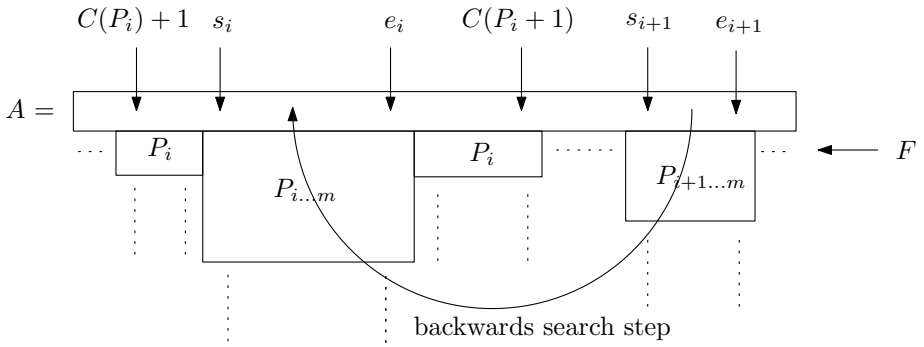
From now on, we measure the space of all data structures in *bits* instead of words, in order to be able to differentiate between the various text indexes. For example, an array of n numbers from the range $[1, n]$ occupies $n\lceil\log n\rceil$ bits, as each array cell stores a binary number consisting 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!

9.3 Backward Search

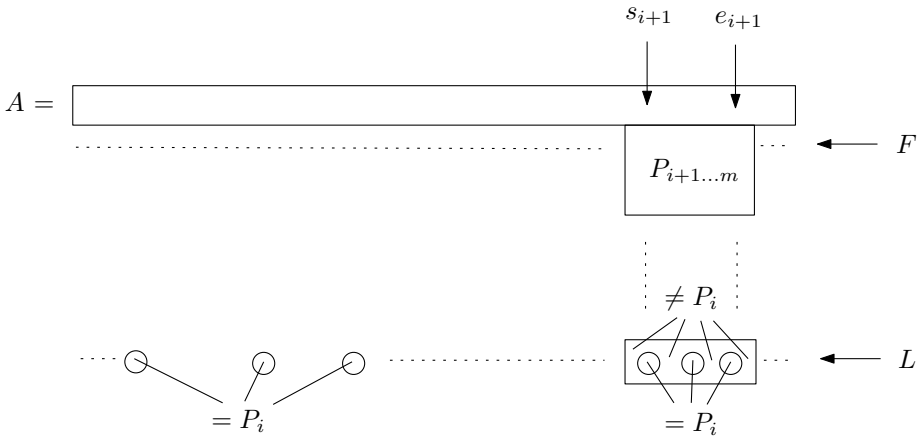
We first focus our attention on the *counting problem* (p. 2); i.e., on finding the number of occurrences of a pattern $P_{1\ldots m}$ in $T_{1\ldots n}$. Recall from Chapter 8 that

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

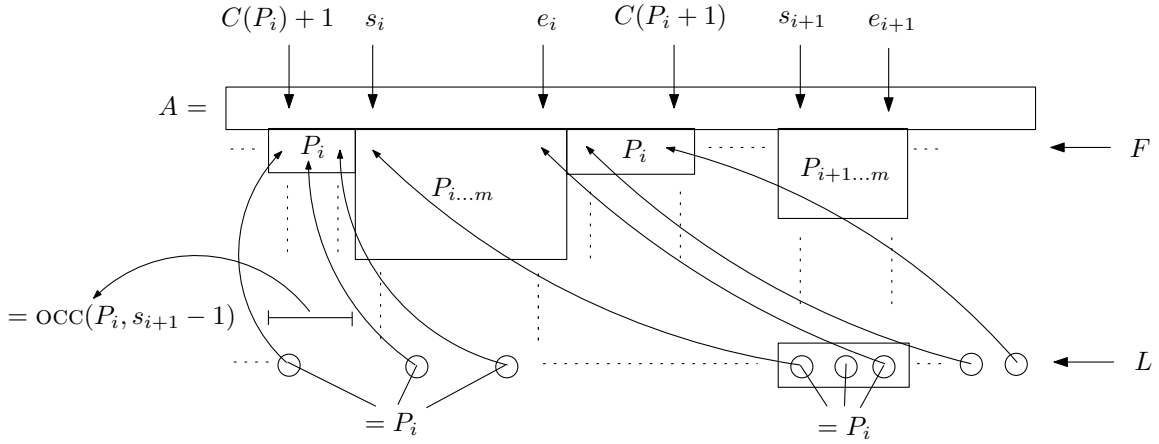
Our aim is identify the interval of P in A by searching P from right to left (= backwards). To this end, suppose we have already matched $P_{i+1\ldots m}$, and know that the suffixes starting with $P_{i+1\ldots m}$ form the interval $[s_{i+1}, e_{i+1}]$ in A. In a *backwards search step*, we wish to 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 $P_{i+1...m}$. 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 $\text{OCC}(P_i, s_{i+1} - 1)$. Likewise, if there are y occurrences of P_i within $L[s_{i+1}, e_{i+1}]$, then $e_i = s_i + y - 1$. Again, y can be computed from the OCC-function.



Algorithm 5: function `backwards-search`($P_{1\ldots m}$)

```
1 s ← 1; e ← n;
2 for i = m ... 1 do
3 |   s ← C(P_i) + occ(P_i, s − 1) + 1;
4 |   e ← C(P_i) + occ(P_i, e);
5 |   if s > e then
6 |   |   return "no match";
7 |   end
8 end
9 return [s, e];
```

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

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

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

Hence, T and A can be deleted once T^{BWT} has been computed. It remains to show how array C and 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 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 occ-values, while using less space implies a higher query time.

Theorem 31. *With backwards search, we can solve the counting problem in $O(m \cdot t_{\text{occ}})$ time, where t_{occ} denotes the time to answer an $\text{occ}(\cdot)$-query.*

9.4 First Ideas for Implementing Occ

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

1. Scan L every time an $\text{occ}(\cdot)$-query has to be answered. This occupies no space, but needs $O(n)$ time for answering a single $\text{occ}(\cdot)$-query, leading to a total query time of $O(mn)$ for backwards search.
2. Store all answers to $\text{occ}(c, i)$ in a two-dimensional table. This table occupies $O(n\sigma \log n)$ bits of space, but allows constant-time $\text{occ}(\cdot)$-queries. Total time for backwards search is *optimal* $O(m)$.

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

Definition 26. *Given a bit-vector $B[1, n]$, $\text{rank}_1(B, i)$ counts the number of 1's in B's prefix $B[1, i]$. Operation $\text{rank}_0(B, i)$ is defined similarly for 0-bits.*

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

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. This can be used as follows for implementing OCC: For each character $c \in \Sigma$, store an *indicator* bit vector $B_c[1, n]$ such that $B_c[i] = 1$ iff $L[i] = c$. Then

$$\text{OCC}(c, i) = \mathit{rank}_1(B_c, i) \ .$$

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

Theorem 32. *With backwards search and constant-time rank operations on bit-vectors, we can answer counting queries in optimal $O(m)$ time. The space (in bits) is $\sigma n + o(\sigma n) + \sigma \log n$.* □

Example 11.

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

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

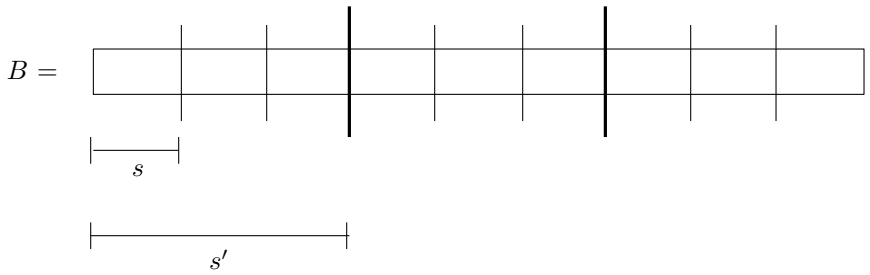
9.5 Compact Data Structures on Bit Vectors

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

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

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 12.

$s = 3 \quad s' = 9$

$B =$ | 0 1 0 | 1 1 0 | 1 1 1 | 0 0 0 | 1 1 1 | 0 0 1 | 1 0 0 | 1 1 0 | 0 1 0 |

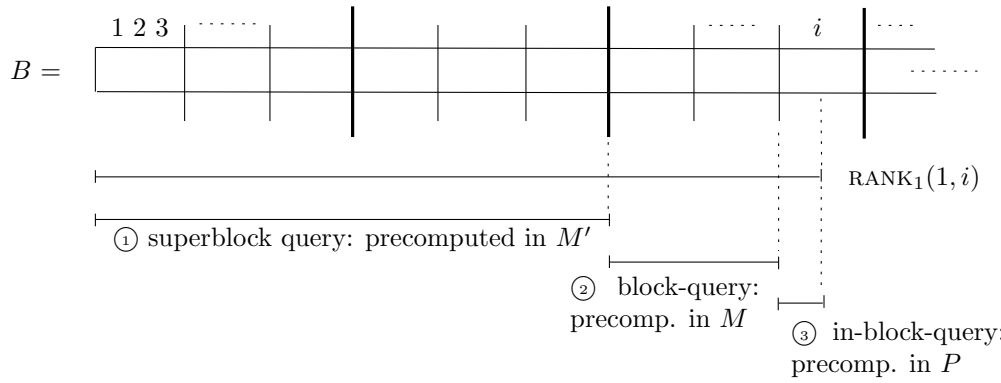
$M' = 6\ 10\ 14$

$M = 1\ 3\ 6\ 0\ 3\ 4\ 1\ 3\ 4$

P:

V \ 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 $rank_1(B,i)$ is then decomposed into 3 sub-queries, as seen in the following picture:



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

$$rank_1(B,i) = M'[q'] + M[q] + P[\underbrace{B[qs+1,(q+1)s]}_{i\text{'s block}}] \quad \underbrace{[i-qs]}_{\text{index in block}}$$

in constant time.

Example 13. *Continuing the example above, we answer* $rank_1(B,17)$ *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,* $rank_1(B,17) = M'[1] + M[5] + P[001][2] = 6 + 3 + 0 = 9$*.*

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

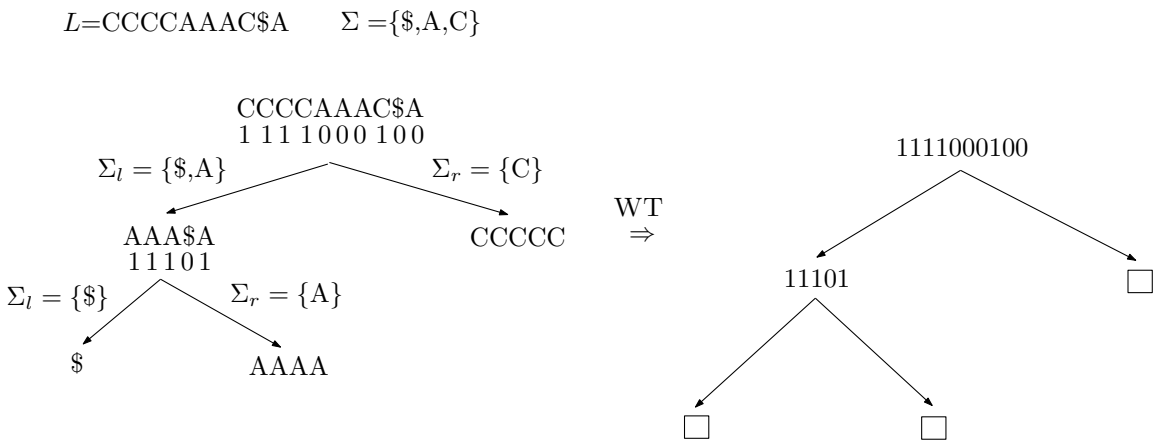
Theorem 33. *An* n*-bit vector* B *can be augmented with data structures of size* $o(n)$ *bits such that* $rank_b(B,i)$ *can be answered in constant time* ($b \in \{0,1\}$)*.*

9.6 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 14.



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 34. *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 (see Sect. 9.5). Hence, at any level, the space needed is $n + o(n)$ bits. Because the depth of the tree is $\lceil \log \sigma \rceil$ the claim on the space follows. In order to "know" the sub-interval of a particular node v in the concatenated bit-vector B_d at level d, we can store two indices α_v and β_v such that $B_d[\alpha_v, \beta_v]$ is the bit-vector B_v associated to node v. This accounts for additional $O(\sigma \log n)$ bits. Then a rank-query is answered as follows ($b \in \{0, 1\}$):

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

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

How does the wavelet tree help for implementing the OCC-function? Suppose we want to compute $\text{OCC}(c, i)$, i. e., the number of occurrences of $c \in \Sigma$ in $L[1, i]$. We start at the root r of the wavelet tree, and check if c belongs to the first or to the second half of the alphabet. In the first case, we know that the c's are "stored" in the *left* child of the root, namely L_r. Hence, the number of c's in $L[1, i]$ corresponds to the number of c's in $L_r[1, rank_0(B_r, i)]$. If, on the hand, c belongs to the second half of the alphabet, we know that the c's are "stored" in the subsequence R_r that corresponds to the *right* child of r, and hence compute the number of occurrences of c in $R_r[1, rank_1(B_r, i)]$ as the number of c's in $L[1, i]$. This leads to the following recursive procedure for computing $\text{OCC}(c, i)$, to be invoked with WT-$\text{OCC}(c, i, 1, \sigma, r)$, 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 6: function `WT-occ`$(c, i, \sigma_l, \sigma_r, v)$

```
1 if σ_l = σ_r then
2 |   return i;
3 end
4 σ_m = ⌊(σ_l + σ_r)/2⌋;
5 if c ≤ Σ[σ_m] then
6 |   return WT-occ(c, rank_0(B_v, i), σ_l, σ_m, l_v);
7 else
8 |   return WT-occ(c, rank_1(B_v, i), σ_m + 1, σ_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 35. *With backward-search and a wavelet-tree on T^{BWT}, we can answer counting queries in $O(m \log \sigma)$ time. The space (in bits) is*

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

10 Inside Google

10.1 Recommended Reading

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

10.2 The Task

You are given a collection $\mathcal{S} = \{s_1, \ldots, s_m\}$ of sequences $s_i \in \Sigma^*$ (web pages, protein or DNA-sequences, or the like). Your task is to build an index on $\mathcal{S}$ such that the following type of on-line queries can be answered *efficiently*:

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

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

Exercise: What has this to do with Google?

10.3 The Straight-Forward Solution

Define a string

$$S = s_1 \# s_2 \# \ldots \# s_m \#$$

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

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

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

10.4 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 $|s_1| \ge \frac{n}{2}$.

10.5 An Optimal Solution

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

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

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

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

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

Algorithm 7: Document Reporting

```
1 procedure report (i, j)
2     m ← RMQ_E(i, j)
3     if E[m] ≤ ℓ then
4         output D[m]
5         if m − 1 ≥ i then report(i, m − 1)
6         if m + 1 ≤ j then report(m + 1, j)
7 end
```

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

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