9 Suffix Trees and Suffix Arrays

This lecture is based on the following sources, which are all recommended reading:

- D. Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge UP, 1997.
- M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific, 2002.
- M. Crochemore, C. Hancart, T. Lecroq, *Algorithms on Strings*, Cambridge UP, 2007.
- J. Kärkkäinen *et al*, Simple Linear Work Suffix Array Construction, Journal of the ACM (JACM) Volume 53 Issue 6, November 2006 Pages 918-936.
- S. Kurtz, *Foundations of Sequence Analysis*, Universität Bielefeld, 2001.

9.1 Introduction

> **Problem** Assume we are given a long text T and many short queries $Q_1, \ldots, Q_k$. For each query sequence Q_i, find all its occurrences in T.

We would like to have a data-structure that allows us to solve this problem efficiently.

Important applications are in the comparison of genomes (in programs such as MUMmer that computes alignments between genomic sequences, based on "maximal unique matches") and in the analysis of repeats (in programs such as REPuter).

For example, the text T might be a genome and the queries might be short words such as transcription factor binding sites.

- Text $T =$
  tttttttgagacggagtctcgctctgtcgcccaggctggagtgcagt
  ggcgggatctcggctcactgcaagctccgcctcccgggttcacgcca
  tctcctgcctcagcctcccaagtagctgggactacaggcgcccgcca
  cggctaatttttttgtatttttagtagagacggggtttcaccgtttta
  cgggatggtctcgatctcctgacctcgtgatccgcccgcctcggcct
  cccaaagtgctgggattacaggcgt
- Query $Q =$ tttta
- Find (all) occurrences of the query Q in the text T

9.2 Basic definitions

Definition 9.2.1 (Prefix, suffix, factor) *We use Σ to denote an alphabet and Σ^* for the set of all strings over Σ. We use ϵ to denote the empty string and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.*

Let $T = t_1 t_2 \ldots t_n$ be a text.

- *A* prefix *of T is a substring of T beginning at the first position in T.*
- *A* suffix *of T is a substring $T_{i\ldots n} = t_i t_{i+1} \ldots t_n$ of T ending at the last position in T.*
- *A* factor *(or* infix*) of T is any substring $T_{i\ldots j} = t_i t_{i+1} \ldots t_j$ of T.*

In bioinformatics, the alphabet Σ is usually of size 4 or 20. In other applications, the alphabet can be much larger, for example, in the analysis of web-surfing patterns, Σ consists of the set of *all links* contained in a collection of web sites.

9.3 The role of suffixes

Consider the text `ababc`
It has the following suffixes: `ababc`, `babc`, `abc`, `bc` and `c`.

Queries are prefixes of suffixes: To determine whether a given query Q is contained in the text, we could simply check whether Q is the *prefix* of one of the suffixes.

9.4 Sharing prefixes

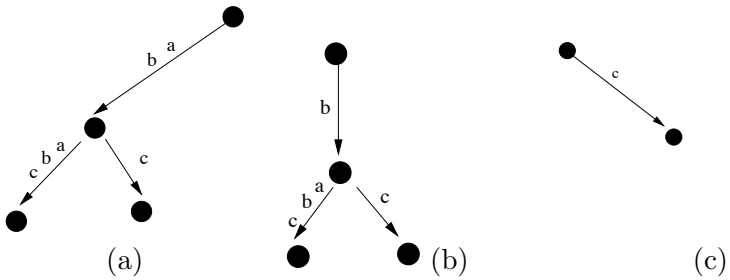
E.g., the query `ab` is the prefix of both `ababc` and `abc`.

To speed up the search for all suffixes that have the query as a prefix, we use a tree structure to *share common prefixes* between the suffixes.

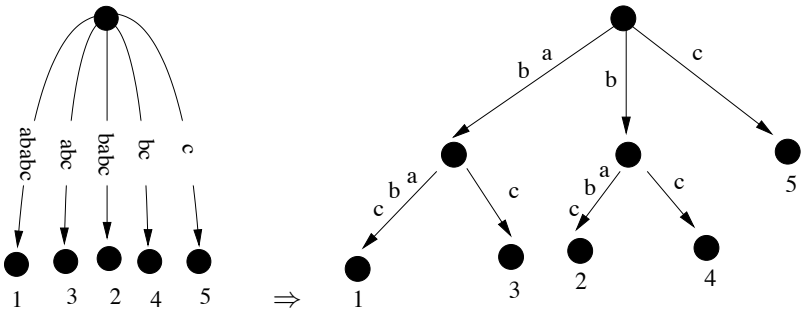
(a) The suffixes *ababc* and *abc* both share the prefix *ab*.

(b) The suffixes *babc* and *bc* both share the prefix *b*.

(c) The suffix *c* doesn't share a prefix.



Here is an example of how the idea of searching the prefix of every suffix can be optimized by sharing prefixes whenever possible:



A "suffix tree" for `ababc` is obtained by sharing prefixes whenever possible. The leaves are annotated by the positions of the corresponding suffixes in the text.

9.5 Σ^+-tree

Definition 9.5.1 (Σ^+-tree) *A* compact Σ^+-tree *is a rooted tree $S = (V, E)$ with edge labels from Σ^+ that fulfills the following two constraints:*

- *For each node $v \in V$, the labels of all outgoing edges each starts with a different letter.*

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

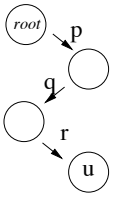
You can think of a compact Σ^+-tree as a tree that looks like a "trie" data structure, but with degree-one-paths contracted into a single edge.

A *leaf* is a node with no children and an edge leading to a leaf is called a *leaf edge*. A node with at least two children is called a *branching node*.

Definition 9.5.2 (Notations in Σ^+-trees) *Let $S = (V, E)$ be a* compact Σ^+-tree.

- *Then $\overline{v}$ denotes the concatenation of all path labels from the root of S to node v.*
- *Then $d(v)$ denotes the* string-depth $|\overline{v}|$ *of v.*
- *A string α* occurs *in S, or S* displays *α, iff there exists a node v and a (not necessarily non-empty) string β with $v = \overline{\alpha\beta}$.*
- *Then words(S) denotes the set of all all strings that occur in S.*
- *If $\overline{v} = \alpha$ for some node v and string α, then we also use $\overline{\alpha}$ to denote v.*

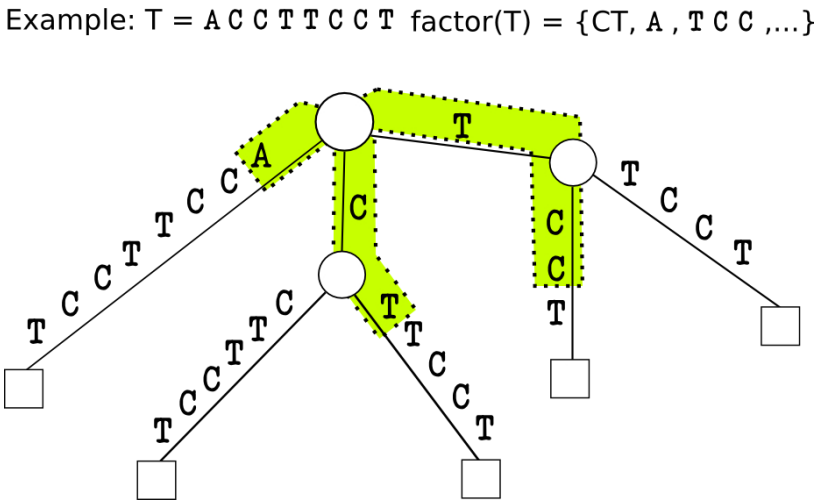
For example $u = \overline{pqr}$:



The root is called $\overline{\epsilon}$.

9.6 Suffix tree

Definition 9.6.1 (Suffix tree) *Let factor(T) denote the set of all factors of T. The* suffix tree *of T is a compact Σ^+-tree S with words(S) = factor(T).*



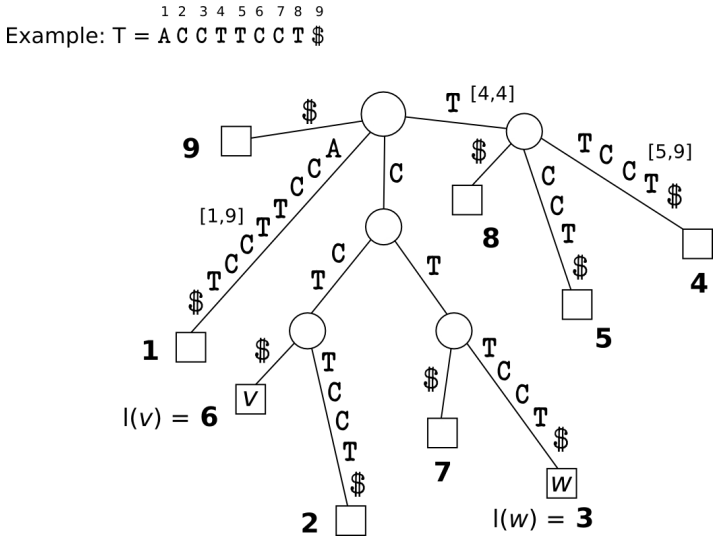
It is convenient to ensure that each suffix ends at a leaf of S. This can be accomplished by always assuming that the text T ends on a *sentinel* letter \$ $\notin \Sigma$ that is lexicographically smaller than all other letters.

Then there is a one-to-one between suffixes of T\$ and the leaves of S and we can *label* each leaf v by the start index $\ell(v) = i$ of the corresponding suffix $T_{i...n}$, i.e. we set $\ell(v) = i \iff \overline{v} = T_{i...n}$.

This explains the name *suffix tree*.

A suffix tree S built for T without a sentinel \$ is called an *implicit* suffix tree.

(Note that we did not define suffix trees in terms of *suffixes*, but rather in terms of *factors*.)



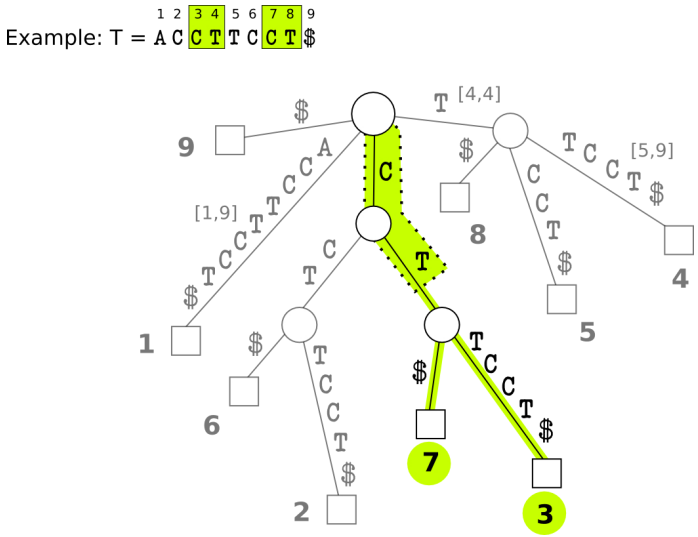
We can use S to *find occurrences* of a query $Q \in \Sigma^*$ as follows. Starting at the root, attempt to match the letters of Q against the labels of the edges of S, along a path into the tree.

If at some point the path cannot be extended, then Q does not occur in T.

Otherwise, Q occurs in T. In this case, the matching procedure ends at some location p in S, either at a node or in an edge. To identify all occurrences of Q in T, we traverse the subtree below p and report all leaf-labels.

- The decision query ("does Q occur in T?") takes $O(m)$ time, with m the length of Q.
- The reporting query ("report all occurrences of Q in T") takes $O(m+k)$ time, where k denotes the number of occurrences of Q in T.

The correctness of the two query algorithms follows from the fact that each factor of T is a *prefix of some suffix* of T.



For an alphabet of constant size, finding the appropriate outgoing edge can be done in constant time.

An important implementation detail is that the edge labels in a suffix tree can be represented by a

pair (i, j), $1 \leq i \leq j \leq n$ such that $T_{i...j}$ equals the corresponding edge label. Hence, each 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 branching nodes, we get the following result:

Theorem 9.6.2 (Linear space) *A suffix tree requires only $O(n)$ space.*

9.7 Suffix Array

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

Definition 9.7.1 (Suffix array) *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 $1 < i \leq n$.*

Let S be the associated suffix tree. If we do a traversal through S, visiting children in lexicographic order of the labels of the edges leading to them, then all leaves will be visited in the order given in the suffix array A.

This links the concepts of suffix array A and suffix tree S.

9.8 LCP Array

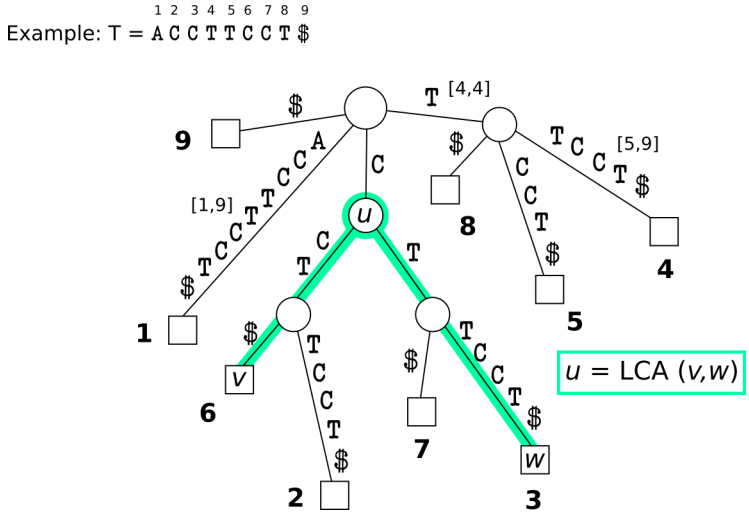
The LCP array is based on the suffix array:

Definition 9.8.1 (LCP array) *The* LCP array *of T is defined as follows:*

- *set $LCP[1] = 0$, and*
- *for all $i > 1$, let $LCP[i]$ equal the length of the longest common prefix (*lcp*) of $T_{A[i]...n}$ and $T_{A[i-1]...n}$.*

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

Definition 9.8.2 (Lowest common ancestor) *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, denoted by $LCA(v, w)$.*

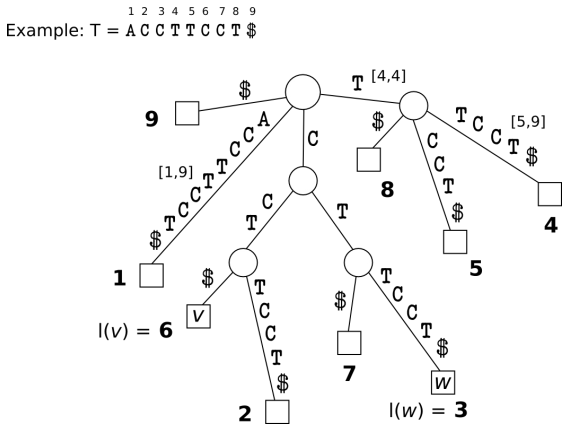


Lemma 9.8.3 (LCA and LCP) *The string-depth of the lowest common ancestor of the two leaves with labels $A[i]$ and $A[i-1]$, respectively, is given by the corresponding entry $LCP[i]$ of the LCP array.*

In other words, for all $i > 1$ we have:

$$LCP[i] = d(LCA(\overline{T_{A[i]\ldots n}}, \overline{T_{A[i-1]\ldots n}})).$$

Example:



i	$A[i]$	suffix	$LCP[i]$
0	9	$	0
1	1	ACCTTCCT$	0
2	6	CCT$	0
3	2	CCTTTCCT$	3
4	7	CT$	1
5	3	CTTCCT$	2
6	8	T$	0
7	5	TCCT$	1
8	4	TTCT$	1

9.9 Suffix Tree from Suffix- and LCP Array

Assume we are given T, A, and LCP, and we wish to construct S, the suffix tree of T. We will show how to do this in $O(n)$ time.

(Later, we will also see how to construct A and LCP 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 $0 \leq i \leq n$, i.e. 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 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 $d(v) \leq LCP[i+1]$.

If $d(v) = LCP[i+1]$, we simply insert a new leaf x to S_i as a child of v, and label (v,x) by $T_{A[i+1]+LCP[i+1]\ldots n}$. Leaf x is labeled by $A[i+1]$. This gives us S_{i+1}.

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

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

9.9.1 Correctness of Algorithm

The correctness of this algorithm follows from results stated above.

What is the execution time of this algorithm?

Although climbing up the rightmost path could take $O(n)$ time in a single step, the running time 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 9.9.1 (Suffix tree from arrays) *We can construct T's suffix tree in linear time from T's suffix- and LCP array.*

This suffix tree construction algorithm is good in terms of *memory locality*; there is a high correlation between how close two pieces of data are in the tree, and how close they are in physical memory.

Hence, an average query will produce only "few" cache misses and leads to a good performance in practice.

9.10 Linear-Time Construction of Suffix Arrays

For an easier presentation, we assume in this section that T is indexed from 0 to $n-1$, $T = t_0 t_1 \dots t_{n-1}$. Also, we will assume that the end of T is padded by a sufficient number of `$`'s. We use T_i to denote the suffix that starts at index i.

Example:

		0	1	2	3	4	5	6	7	8	9	10	11
T	=	y	a	b	b	a	d	a	b	b	a	d	o

We want to obtain the following in linear time:

$$A = (1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$$

We will describe the DC3 algorithm[1], which operates in three steps:

(0) Extract a *sample* that contain 2/3 of all suffixes.

(1) Recursively sort the extracted suffixes.

(2) Use these to sort the other 1/3 of suffixes.

(3) Merge the two sorted lists of suffixes to obtain the final suffix array.

Step 0: Extract sample. For $k = 0, 1, 2$ define

$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and T_C the set of *sample suffixes*.

Example: $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$ and $C = \{1, 2, 4, 5, 7, 8, 10, 11\}$.

Let $\max B_k$ denote the maximum element in B_k, in this example $\max B_1 = 10$ and $\max B_2 = 11$.

Step 1: Sort sample suffixes. For $k = 1, 2$ construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \dots [t_{\max B_k} t_{\max b_k + 1} t_{\max B_k + 2}],$$

[1] J. Kärkkäinen *et al* 2006

whose letters are triples $[t_i t_{i+1} t_{i+2}]$.

Note that the last letter of R_k is always unique because it ends on `$`.

Let $R = R_1 \oplus R_2$ be the concatenation of R_1 and R_2.

Then the (nonempty) suffixes of R correspond to the set T_C of sample suffixes:

$$[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \ldots \text{ corresponds to } T_i.$$

This correspondence is order-preserving, so by sorting the suffixes of R we get the order of the sample suffixes T_C.

Example: $R = [abb][ada][bba][do\$][bba][dab][bad][o\$\$]$.

To sort the suffixes of R, first radix sort the letters (which are triples) of R and then rename them with their ranks to obtain a new string R'.

Example:	$i =$	0	1	2	3	4	5	6	7
	$R =$	[abb]	[ada]	[bba]	[do$]	[bba]	[dab]	[bad]	[o$$]
	$R' =$	1	2	4	6	4	5	3	7

If all the characters are different, then the lexicographic ordering of the characters determines the order of the suffixes. Otherwise, we *recursively* call Algorithm DC3 to sort the suffixes of R'.

Example: $A_{R'} = (0, 1, 6, 4, 2, 5, 3, 7)$.

Now assign a rank to each sample suffix based on $A_{R'}$.

Example:	$i =$	0	1	2	3	4	5	6	7
	Index in $T =$	1	4	7	10	2	5	8	11
	$R' =$	1	2	4	6	4	5	3	7
	rank=	1	2	5	7	4	6	3	8

This sorts the set of suffixes in T_C.

Example: $T_1 < T_4 < T_8 < T_2 < T_7 < T_5 < T_{10} < T_{11}$.

For $i \in C$ let $rank(T_i)$ denote the rank of suffix T_i in the sample set T_C, obtained from the rank of the corresponding suffix in R'.

Additionally, define $rank(T_{n+1}) = rank(T_{n+2}) = 0$.

For $i \in B_0$, the value of $rank(T_i)$ is undefined ("$\perp$").

Example:	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	$rank(T_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Step 2: Sort nonsample suffixes. Represent each nonsample suffix $T_i \in T_{B_0}$ as the pair $(t_i, rank(T_{i+1}))$. Note that $rank(T_{i+1})$ is defined for all $i \in B_0$.

For all $i, j \in B_0$ we have:

$$T_i \leq T_j \Leftrightarrow (t_i, rank(T_{i+1})) \leq (t_j, rank(T_{j+1})).$$

Using this and applying radix sort to the set of all pairs $(t_i, rank(T_{i+1}))$, we sort all nonsample suffixes.

Example:
$T_{12} < T_6 < T_9 < T_3 < T_0$ because $(\$, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.

Step 3: Merge. The two sorted sets of suffixes are merged in the straightforward way, comparing any sample suffix $T_i \in T_C$ with a non-sample suffix $T_j \in T_{B_0}$ as follows:

if $i \in B_1$ then:

$$T_i \leq T_j \Leftrightarrow (t_i, rank(T_{i+1})) \leq (t_j, rank(T_{j+1})),$$

else ($i \in B_2$):

$$T_i \leq T_j \Leftrightarrow (t_i, t_{i+1}, rank(T_{i+2})) \leq (t_j, t_{j+1}, rank(T_{j+2})).$$

Example:
$T_1 < T_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$, and $T_3 < T_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

The time complexity is given by the following result:

Theorem 9.10.1 (Linear time) *The time complexity of Algorithm DC3 is $O(n)$.*

Proof Excluding the recursive call, everything can clearly be done in linear time. The recursion is on a string of length $\lfloor 2n/3 \rfloor$. Thus the time is given by the recurrence $T(n) = T(2n/3) + O(n)$, whose solution is $T(n) = O(n)$. □

9.10.1 Linear-Time Construction of LCP Arrays

It remains to be shown how the LCP array 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 that maps the i-th suffix to its rank in the lexicographical ordering of all suffixes.

We will construct the LCP in the original order of the suffixes, using A^{-1} to index the appropriate element of LCP. This is a good order in which to proceed because the following result states that in each step the value can only decrease by at most one:

Lemma 9.10.2 *For all $1 \leq i < n$: $LCP[A^{-1}[i+1]] \geq LCP[A^{-1}[i]] - 1$.*

Proof Let T_j be the lexicographic predecessor of T_i, i.e., $T_j < T_i$ and there are no other suffixes between them in the lexicographical order. Then T_j is direct predecessor of T_i in A and so

$$LCP[A^{-1}[i]] = lcp(T_i, T_j) =: h.$$

If $h = 0$, then the claim is trivially true.

If $h > 0$, then for some letter c we have $T_i = cT_{i+1}$ and $T_j = cT_{j+1}$. Then $T_{j+1} < T_{i+1}$ and $lcp(T_{i+1}, T_{j+1}) = h - 1$.

Let T_k be the lexicographical predecessor of T_{i+1}. Then either $T_{j+1} = T_k$ or $T_{j+1} < T_k < T_{i+1}$. In either case,

$$LCP[A^{-1}[i+1]] = lcp(T_{i+1}, T_k) \geq lcp(T_{i+1}, T_{j+1}) = h - 1,$$

based on the following argument:

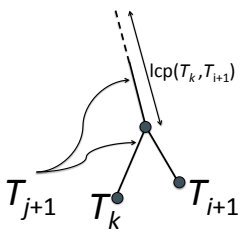
If $T_{j+1} = T_K$, then equality holds in the equation above.

Otherwise, T_{j+1} must come before T_k.

Because T_k is the direct predecessor of T_{i+1}, it follows that in the corresponding suffix tree we have that the leaf representing T_k is immediately followed by the leaf representing T_{k+1}, and both leaf edges have the same parent.

Because T_{j+1} is a predecessor of T_k, it follows that T_{j+1} attaches directly, or indirectly, to a node on the path from the root to T_k.

Thus the longest common prefix of T_{j+1} and T_{i+1} cannot exceed $lcp(T_k, T_{i+1})$, as seen here:



□

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

Algorithm 9.10.3 (LCP from suffix array)

begin
0 $LCP[0] = 0$
1 $h \leftarrow 0$ /* *length of lcp for previous suffix* */
2 **for** $i = 1, \ldots, n$ **do**
3 $k \leftarrow A^{-1}[i]$
4 $j \leftarrow A[k-1]$ /* T_j *is lexicographical predecessor of* T_i */
5 **while** $t_{i+h} = t_{j+h}$ **do**
6 $h \leftarrow h + 1$
7 $LCP[k] \leftarrow h$
8 $h \leftarrow \max\{0, h-1\}$
end

Theorem 9.10.4 (Linear time construction of LCP) *The LCP array for a text of length n can be constructed in $O(n)$ time.*

Proof This follows from the presented algorithm: The *for* loop (line 2) is executed n times. Note that h cannot grow larger than n, is decremented at most n times, and is incremented in each round of the *while* loop (line 5). It follows that line (6) is executed at most $2n$ times. □

9.11 Summary and outlook

The LCP array and suffix array A can be constructed in linear time for any given sequence T.

We can construct the suffix tree S from LCP and A in linear time.

In total, this provides a linear time construction of S from T.

Suffix array and LCP can be used together to replace a suffix tree, using less space (not shown here).