# Chapter 2 : String comparison

## *Basic notions*

Let $\Sigma$ be finite ordered alphabet. We suppose $|\Sigma|$ is constant. $\Sigma^*$ is the set of all strings over $\Sigma$. We will use symbols $S, S_1, S_2, \ldots$ for strings. $|S|$ denotes length of string $S$. We also use symbol $n$ to refer to length of string, if not specified otherwise. We write $S[i]$, where $0 \leqslant i < |S|$, to refer to i-th character of $S$. We define $S[-1] = ¢$ and $S[|S|] = \$$, where $¢, \$ \notin \Sigma$ are special symbols. $S[i..j]$, where $0 \leqslant i \leqslant j < |S|$ refers to substring of $S$, starting at position $i$ and ending at position $j$.

Each position $0 \leqslant i < |S|$ represents unique suffix $S[i..n-1]$ of $S$. We define left context $LC_S(i) = S[i-1]$ for each suffix $i$. Note that suffix 0 has special left context $¢$. When it is clear, which string we are refering to, we write just $LC(i)$.

## *Matches and repeats*

**Definition:** Let $S$ be string of length $n$. A triple $(p_1, p_2, l) \in N_n^3$ is called *repeat* iff

1. $p_1 + l - 1 < n \wedge p_2 + l - 1 < n$
2. $S[p_1 .. p_1 + l - 1] = S[p_2 .. p_2 + l - 1]$

A repeat $(p_1, p_2, l)$ is called *left maximal* if $LC(p_1) \neq LC(p_2) \vee LC(p_1) = ¢$.
A repeat $(p_1, p_2, l)$ is called *right maximal* if $S[p_1 + l] \neq S[p_2 + l] \vee S[p_1 + l] = \$$.
A repeat is called *maximal* if it is left and right maximal.

**Definition:** Let $S_1, S_2$ be strings, $m = min(|S_1|, |S_2|) + 1$. A triple $(p_1, p_2, l) \in N_{|S_1|} \times N_{|S_2|} \times N_m$ is called *match (of* $S_1$ *and* $S_2$*)* iff

1. $p_1 + l - 1 < |S_1| \wedge p_2 + l - 1 < |S_2|$
2. $S_1[p_1 .. p_1 + l - 1] = S_2[p_2 .. p_2 + l - 1]$

A match $(p_1, p_2, l)$ is called *left maximal* if $LC_{S_1}(p_1) \neq LC_{S_2}(p_2) \vee LC_{S_1}(p_1) = ¢$.
A match $(p_1, p_2, l)$ is called *right maximal* if $S_1[p_1 + l] \neq S_2[p_2 + l] \vee S_1[p_1 + l] = \$$.
A match is called *maximal* if it is left and right maximal.

# Chapter 3 : Enhanced suffix array and it's implementation

In Chapter 4, we'll describe algorithms for computing maximal matches and maximal repeats. These algorithms work with rather non-trivial data structure: *enhanced suffix array*. This data structure was originally introduced in [1]. This chapter contains only brief introduction and also deals with implementation issues.

Enhanced suffix array consists of regular suffix array enhanced with additional information: lcp-table. Rows of lcp-table that meet certain conditions can be grouped to intervals called *lcp-intervals*. These then constitute a virtual data structure *lcp-interval tree*. This structure is sufficient to replace suffix tree (text indexing data structure used for similar purposes) in almost every application, being more memory efficient, as shown in [2].

[TODO: better introduction, mention suffix tree and reference paper about it]

**Definition:** Let $S$ be string, $|S|=n$. An array sa of $n$ integers in range $0$ to $n$-$1$ is called a *suffix array* of string $S$ iff $sa[0], sa[1], \ldots, sa[n-1]$ is sequence of positions of suffixes of $S$ in ascending lexicographic order, i.e $\forall i, j: 0 \leqslant i < j < n \Rightarrow S[sa[i]..n-1] <_L S[sa[j]..n-1]$, where $<_L$ is lexicographic order on $\Sigma^*$.

Currently, there are three different ways how to compute suffix array directly (without first computing suffix tree) in linear time. For details see [4], [5] and [6]. In implementation of DiffEngine library, we use algorithm described in [5]. When DiffEngine computes the suffix array over $n$ bytes of data, peak memory usage in our implementation is $9.28n$ bytes. We suppose[1] that $n < 2^{31}$. Resulting suffix array can be therefore stored in $4n$ bytes, where most significant bit in each value can be used for special purposes during further computation.

**Definition:** Let $S$ be string, $|S|=n$, sa is suffix array of $S$. The *lcp-table* lcptab of string $S$ is an array of integers in range 0 to n-1. We define *lcptab[0] = 0* and *lcptab[i]* is length of longest common prefix of suffixes $S[sa[i-1]..n-1]$ and $S[sa[i]..n-1]$ for $0 < i < n$.

Lcp-table can be computed in linear time when suffix array is available (see [7]), or can be computed as by-product of linear time suffix array construction as shown in [4]. In DiffEngine library, we compute lcp-table separately from suffix array, using space saving implementation trick described in [3]. That way, we can compute it, with $4n$ peak extra memory usage (apart from memory used to store text ($n$) and suffix array ($4n$)). Amount of memory for storage varies and depends on input data. DiffEngine library uses variable coding scheme ranging from $n$ to $4n$. This will be described in section XXX.

Table 1 shows example of enhanced suffix array. Note that we also added a row with left context of each suffix.

| *i* | *suftab* | *lcptab* | *LC* | *S[suftab[i]..n-1]* |
|---|---|---|---|---|
| 0 | 2 | 0 | c | aaacatat |
| 1 | 3 | 2 | a | aacatat |
| 2 | 0 | 1 | ¢ | acaaacatat |
| 3 | 4 | 3 | a | acatat |
| 4 | 6 | 1 | c | atat |
| 5 | 8 | 2 | t | at |
| 6 | 1 | 0 | a | caaacatat |
| 7 | 5 | 2 | a | catat |
| 8 | 7 | 0 | a | tat |
| 9 | 9 | 1 | a | t |

*Table 1: Enhanced suffix array of the string S=acaaacatat*

1 This limit is practically much lower. Current version of DiffEngine works only on 32bit systems with 4GB RAM limit. Therefore with peak memory usage $9.28n$, internal memory algorithm can compute suffix array on cca 440MB of data. The number is even little lower as the factor doesn't consider a few megabytes of data structures whose size don't depend on $n$.

**Definition:** A triple $(l, i, j) \in N_n^3$ is an lcp-interval iff

1. $i < j$
2. $lcptab[i] < l$
3. $\forall k, i+1 \leqslant k \leqslant j : lcptab[k] \geqslant l$
4. $\exists k, i+1 \leqslant k \leqslant j : lcptab[k] = l$
5. $lcptab[j+1] < l$

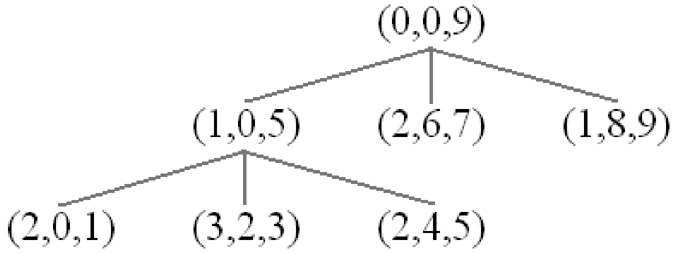


*Illustration 1: Lcp-interval tree for suffix array in Table 1*

**Definition:** A lcp-interval $(p, q, r)$ is said to be *embedded* in lcp-interval $(l, i, j)$ if $i \leqslant q < r \leqslant j \wedge p > l$. Lcp-interval $(l, i, j)$ is then called the interval *enclosing* $(p, q, r)$. If $(l, i, j)$ encloses $(p, q, r)$ and there is no interval embedded in $(l, i, j)$ that also encloses $(p, q, r)$, then is $(p, q, r)$called a *child interval* of $(l, i, j)$.

This parent-child relationship constitutes a conceptual (or virtual) tree which we call *lcp-interval tree*. Root of this tree is interval *(0,0,n-1).* See Illustration 1.

This tree is never really constructed i.e. no parent-child pointers are stored in memory. All lcp-intervals and their relationships can be computed during one sequential scan of lcp-table which would simulate bottom-up traversal of the tree.

lcp-interval $(l, i, j)$ also defines an interval of suffix array sa. We say that a position *p* belongs to (or is from) interval $(l, i, j)$ if $p = sa[k]$ where $i \leqslant k \leqslant j$.

Lcp-intervals represent very useful information from repeat point of view.
Consider following three facts about lcp-intervals:

Let $p_{1,} p_2$ be the positions from a lcp-interval $(l, i, j)$, $p_1 = sa[k_1]$, $p_2 = sa[k_2]$, $k_1 < k_2$

**Fact 1:** $(l, p_{1,} p_2)$ is repeat.

Proof: From property 3 of lcp-interval it follows that $\forall k, k_1 < k \leqslant k_2 : lcptab[k] \geqslant l$, i.e.
$\forall k, k_1 < k \leqslant k_2$ : *S[sa[k-1]..sa[k-1]+l-1] = S[sa[k]..sa[k]+l-1]*
$\Rightarrow S[sa[k_1]..sa[k_1]+l-1] = S[sa[k_2]..sa[k_2]+l-1]$

**Fact 2:** $(l, p_{1,} p_2)$ is left maximal iff $LC(p_1) \neq LC(p_2) \vee LC(p_1) = ¢$

**Fact 3:** $(l, p_{1,} p_2)$ is right maximal iff there is no other lcp-interval $(p, q, r)$ embedded in $(l, i, j)$ so that $p_{1,} p_2$ both belong to $(p, q, r)$.

Proof: $\Rightarrow$:
Let $(l, p_{1,} p_2)$ be right maximal and let $p_{1,} p_2$ belong to $(p, q, r)$ embedded in $(l, i, j)$. From fact 1, $(p, p_{1,} p_2)$ is also repeat, with *p>l*, which contradicts right maximality of $(l, p_{1,} p_2)$
$\Leftarrow$:
Let $(l, p_{1,} p_2)$ be not right maximal, i.e. $S[p_1+l] = S[p_2+l] \wedge S[p_1+l] \neq \$$
$\Rightarrow S[sa[k_1]..sa[k_1]+l] = S[sa[k_2]..sa[k_2]+l]$. We see that suffixes at $p_{1,} p_2$ have common prefix of length at least l+1. Since sa is sorted lexicographically, also suffixes at *sa[k]* for $\forall k, k_1 < k < k_2$, have the same prefix and therefore

$\forall k, k_1 < k \leqslant k_2 : lcptab[k] \geqslant l+1$. Let $p = min\{lcptab[k] | k_1 < k \leqslant k_2\}$ From property 2 of lcp-interval:$\exists q, i \leqslant q \leqslant k_1 : lcptab[q] < p$ and from property 5 $\exists r, k_2 \leqslant r \leqslant j : lcptab[r] < p$. If we take maximal such *q* and minimal such *r*, we have an lcp-interval $(p, q, r)$ embedded in $(l, i, j)$.

## *Implementation issues*

In worst case lcp-table takes *4n* bytes. In most cases lcp values rarely exceed $2^8$ or $2^{16}$ and therefore can be represented by 1- or 2- byte integer (`char` or `short`). When most of the values of LCP table are small, we can save space by using more economical coding and register exceptions for greater values.

| | |
|---|---|
| 0 | End of interval |
| 1 | m |
| 2 | m+1 |
| ... | |
| 254 | m+253 |
| 255 | Exception |

*Table 2: 1-byte LCP code*

Let *m* be minimal match/repeat length. Since we process only items i from *SA* and *LCP* tables where $LCP[i] \geqslant m$, we don't need to represent values smaller than *m*. Furthermore, we need one value to indicate end of root interval and one to indicate exception (value greater than m+254). LCP values will be coded to 1-byte representation according to Table 1. The 2-byte representation is coded similarly.

To determine which representation of LCP table to use, we determine following numbers:

$k_1$ - number of values greater than or equal m+254
$k_2$ - number of values greater than or equal m+65634
$k'_1$ - number of values greater than m+254
$k'_2$ - number of values greater than m+65634

Then, we use rules in Table 2 to decide coding. We start from line 1. If condition in line i holds we use given coding, else we assume conditions 1..i are false and move to line i+1.

| *i* | *condition* | *lcp value coding* | *exception coding* | *table size* |
|---|---|---|---|---|
| 1 | $k'_1 = 0$ | 1-byte | - | $n$ |
| 2 | $k'_2 = 0 \wedge n + 2k_1 < 2n$ | 1-byte | 2-byte | $n + 2k_1$ |
| 3 | $k'_2 = 0$ | 2-byte | - | $2n$ |
| 4 | $n + 4k_1 < 2n + 4k_2$ | 1-byte | 4-byte | $n + 4k_1$ |
| 5 | $2n + 4k_2 < 4n$ | 2-byte | 4-byte | $2n + 4k_2$ |
| 6 | true | 4-byte | - | $4n$ |

Table 3: coding of lcp value

# Bibliography


1: M.I. Abouelhoda, S. Kurtz and E. Ohlebusch, *The Enhanced Suffix Array and its Applications to Genome Analysis*, In Proceedings of the Second Workshop on Algorithms in Bioinformatics, pages 449-463. Lecture Notes in Computer Science 2452, Springer-Verlag, 2002

2: M.I. Abouelhoda, S. Kurtz and E. Ohlebusch, *Replacing Suffix Trees with Enhanced Suffix Arrays*, Journal of Discrete Algorithms, 2:53-86, 2004

3: Giovanni Manzini, *Two space saving tricks for linear time LCP computation*, Technical Report TR-INF-2004-02-03-UNIPMN, 2004

4: Juha Kärkkäinen and Peter Sanders, *Simple linear work suffix array construction*, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03). LNCS 2719, Springer, 2003, pp. 943-955, 2003

5: P. Ko and S. Aluru, *Space-efficient linear time construction of suffix arrays*, Combinatorial Pattern Matching, pp. 200-210, 2003

6: D. K. Kim, J. S. Sim, H. Park, K. Park, *Linear-time construction of suffix arrays*, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, pp. 186-199, 2003

7: T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, In Proc. 12th Symposium on Combinatorial Pattern Matching (CPM ’01), pages 181–192. Springer-Verlag LNCS n. 2089, 2001