Fast and Practical Algorithms for Computing All the Runs in a String*

Gang Chen[1], Simon J. Puglisi[2], and W.F. Smyth[1,2]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
smyth@mcmaster.ca
www.cas.mcmaster.ca/cas/research/algorithms.htm

[2] Department of Computing, Curtin University, GPO Box U1987
Perth WA 6845, Australia
{puglissj,smyth}@computing.edu.au

Abstract. A ***repetition*** in a string $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e$ of $\boldsymbol{x}$, maximum $e \geq 2$, where $\boldsymbol{u}$ is not itself a repetition in $\boldsymbol{w}$. A ***run*** in $\boldsymbol{x}$ is a substring $\boldsymbol{w} = \boldsymbol{u}^e\boldsymbol{u}^*$ of "maximal periodicity", where $\boldsymbol{u}^e$ is a repetition and $\boldsymbol{u}^*$ a maximum-length possibly empty proper prefix of $\boldsymbol{u}$. A run may encode as many as $|\boldsymbol{u}|$ repetitions. The maximum number of repetitions in any string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is well known to be $\Theta(n \log n)$. In 2000 Kolpakov & Kucherov showed that the maximum number of runs in $\boldsymbol{x}$ is $O(n)$; they also described a $\Theta(n)$-time algorithm, based on Farach's $\Theta(n)$-time suffix tree construction algorithm (STCA), $\Theta(n)$-time Lempel-Ziv factorization, and Main's $\Theta(n)$-time leftmost runs algorithm, to compute all the runs in $\boldsymbol{x}$. Recently Abouelhoda *et al.* proposed a $\Theta(n)$-time Lempel-Ziv factorization algorithm based on an "enhanced" suffix array — a suffix array together with other supporting data structures. In this paper we introduce a collection of fast space-efficient algorithms for computing all the runs in a string that appear in many circumstances to be superior to those previously proposed.

1 Introduction

Periodicity (repetition) in infinite strings was the first topic of stringology [30]; counting and computing the maximum-length adjacent repeating substrings (repetitions) in a finite string was, along with pattern-matching, one of the earliest computational problems on strings to be studied [17,19]. Given a nonempty string $\boldsymbol{u}$ and an integer $e \geq 2$, we call $\boldsymbol{u}^e$ a ***repetition***; if $\boldsymbol{u}$ itself is not a repetition, then $\boldsymbol{u}^e$ is a ***proper repetition***. Given a string $\boldsymbol{x}$, a ***repetition in*** $\boldsymbol{x}$ is a substring

$$\boldsymbol{x}[i..i+e|\boldsymbol{u}|-1] = \boldsymbol{u}^e,$$

where $\boldsymbol{u}^e$ is a proper repetition and neither $\boldsymbol{x}\big[i+e|\boldsymbol{u}|..i+(e+1)|\boldsymbol{u}|-1)\big]$ nor $\boldsymbol{x}[i-|\boldsymbol{u}|..i-1]$ equals $\boldsymbol{u}$. Following [29], we say the repetition has ***generator*** $\boldsymbol{u}$,

* The work of the first and third authors was supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

B. Ma and K. Zhang (Eds.): CPM 2007, LNCS 4580, pp. 307–315, 2007.

period $|\boldsymbol{u}|$, and ***exponent*** e; it can be specified by the integer triple $(i, |\boldsymbol{u}|, e)$. It is well known [17,3] that the maximum number of repetitions in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is $\Theta(n \log n)$, and that the number of repetitions in $\boldsymbol{x}$ can be computed in $\Theta(n \log n)$ time [3,2,20].

A string $\boldsymbol{u}$ is a ***run*** iff it is periodic of (minimum) period $p \leq |\boldsymbol{u}|/2$. Thus $\boldsymbol{x} = abaabaabaabaab = (aba)^4ab$ is a run of period $|aba| = 3$. A substring $\boldsymbol{u} = \boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ is a ***run in*** $\boldsymbol{x}$ iff it is a run of period p and neither $\boldsymbol{x}[i-1..j]$ nor $\boldsymbol{x}[i..j+1]$ is a run of period p (***nonextendible***). The run $\boldsymbol{u}$ has ***exponent*** $e = \lfloor |\boldsymbol{u}|/p \rfloor$ and possibly empty ***tail*** $\boldsymbol{t} = \boldsymbol{x}[i+ep..j]$ (proper prefix of $\boldsymbol{x}[i..i+p–1]$). Thus

$$\begin{array}{cccccccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ \boldsymbol{x} & = & b & a & a & a & b & a & a & b & a & a & b & a & b & a \end{array}$$

has a run $\boldsymbol{x}[3..12]$ of period $p = 3$ and exponent $e = 3$ with tail $\boldsymbol{t} = a$ of length $t = |\boldsymbol{t}| = 1$. It can also be specified by a triple $(i, j, p) = (3, 12, 3)$, and it includes the repetitions $(aab)^3$, $(aba)^3$ and $(baa)^2$ of period $p = 3$. In general, for $e = 2$ a run ***encodes*** $t+1$ repetitions; for $e > 2$, p repetitions. Clearly, computing all the runs in $\boldsymbol{x}$ specifies all the repetitions in $\boldsymbol{x}$.

Runs were introduced by Main [18], who showed how to compute the leftmost occurrence of every run in $\boldsymbol{x} = \boldsymbol{x}[1..n]$ by

(1) computing $\mathrm{ST}_{\boldsymbol{x}}$, the suffix tree of $\boldsymbol{x}$ [32];
(2) using $\mathrm{ST}_{\boldsymbol{x}}$ to compute $\mathrm{LZ}_{\boldsymbol{x}}$, the Lempel-Ziv factorization of $\boldsymbol{x}$ [16];
(3) using $\mathrm{LZ}_{\boldsymbol{x}}$ to compute leftmost runs.

Since steps (2) and (3) require only $\Theta(n)$ (linear) time, the use of Farach's linear-time STCA [5] enables the leftmost runs to be computed in linear time. In [14] Kolpakov & Kucherov proved that the maximum number of runs in any string of length n is $\Theta(n)$, and then showed how to compute all the runs in $\boldsymbol{x}$ from the leftmost ones in linear time. Thus in theory all runs, hence all repetitions, could be computed in linear time, though Farach's algorithm is not practical for large n.

In [1] Abouelhoda, Kurtz & Ohlebusch show how to compute $\mathrm{LZ}_{\boldsymbol{x}}$ from a suffix array $\mathrm{SA}_{\boldsymbol{x}}$, together with other linear structures, rather than from $\mathrm{ST}_{\boldsymbol{x}}$. Since there now exist practical linear-time suffix array construction algorithms (SACAs) [9,12], it thus becomes feasible to compute all the runs in $\boldsymbol{x}$ in $\Theta(n)$ time for large values of n.

In this paper we describe variants of a worst-case linear-time algorithm (CPS) that, given $\mathrm{SA}_{\boldsymbol{x}}$ and the corresponding longest common prefix array $\mathrm{LCP}_{\boldsymbol{x}}$, computes $\mathrm{LZ}_{\boldsymbol{x}}$ in guaranteed $\Theta(n)$ time and, according to our experiments, does so generally faster and generally with lower space requirements than either of the algorithms AKO [1] or KK-LZ (a suffix tree-based implementation of Ukkonen's algorithm [31] by Kolpakov & Kucherov specifically designed for alphabet size $\alpha \leq 4$ [13]). Ukkonen's algorithm constructs ST on-line and so permits LZ to be built from subtrees of ST; this gives it an advantage, at least in terms of space, over the fast and compact version of McCreight's STCA [25] due to Kurtz [15]. Note also [26] that the linear-time algorithms [9,12] for computing $\mathrm{SA}_{\boldsymbol{x}}$ are not, in practice, as fast as other algorithms [24,22] that have only supralinear worst-case time bounds. Thus in testing AKO and CPS we make use of the supralinear SACA [22] that is probably at present the fastest in practice.

In Section 2 we describe our new algorithms. Section 3 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 4 outlines future work.

2 Description of the Algorithms

Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet A of size α, we refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as ***suffix*** i. Then $\mathrm{SA}_{\boldsymbol{x}}$ is an array $1..n$ in which $\mathrm{SA}_{\boldsymbol{x}}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes of $\boldsymbol{x}$. Let $\mathrm{lcp}_{\boldsymbol{x}}(i_1, i_2)$ denote the ***longest common prefix*** of suffixes i_1 and i_2 of $\boldsymbol{x}$. Then $\mathrm{LCP}_{\boldsymbol{x}}$ is an array $1..n+1$ in which $\mathrm{LCP}_{\boldsymbol{x}}[1] = \mathrm{LCP}_{\boldsymbol{x}}[n+1] = -1$, while for $j \in 2..n$,

$$\mathrm{LCP}_{\boldsymbol{x}}[j] = \Big|\mathrm{lcp}_{\boldsymbol{x}}\big(\mathrm{SA}_{\boldsymbol{x}}[j-1], \mathrm{SA}_{\boldsymbol{x}}[j]\big)\Big|.$$

Given $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{LCP}_{\boldsymbol{x}}$ can be quickly computed in $\Theta(n)$ time [11,23]. When the context is clear, we write SA for $\mathrm{SA}_{\boldsymbol{x}}$, LCP for $\mathrm{LCP}_{\boldsymbol{x}}$. For example:

$$\begin{array}{rl} & \;1\;\;2\;\;3\;\;4\;\;5\;\;6\;\;7\;\;8\;\;9 \\ \boldsymbol{x} = & \;a\;\;b\;\;a\;\;a\;\;b\;\;a\;\;b\;\;a \\ \mathrm{SA}_{\boldsymbol{x}} = & \;8\;\;3\;\;6\;\;1\;\;4\;\;7\;\;2\;\;5 \\ \mathrm{LCP}_{\boldsymbol{x}} = & \text{-}1\;1\;\;1\;\;3\;\;3\;\;0\;\;2\;\;2\;\text{-}1 \end{array}$$

The ***LZ factorization*** $\mathrm{LZ}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ is a factorization $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$ such that each $\boldsymbol{w_j}$, $j \in 1..k$, is

(a) a letter that does *not* occur in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_{j-1}}$; or otherwise
(b) the longest substring that occurs at least twice in $\boldsymbol{w_1 w_2} \cdots \boldsymbol{w_j}$.

For our example string, $\boldsymbol{w_1} = a$, $\boldsymbol{w_2} = b$, $\boldsymbol{w_3} = a$, $\boldsymbol{w_4} = aba$, $\boldsymbol{w_5} = ba$. Typically, integer pairs (POS, LEN) specify the factorization, where POS gives a position in $\boldsymbol{x}$ and LEN the corresponding length at that position (by convention zero if the position contains a "new" letter). The example thus yields $(\mathrm{POS}, \mathrm{LEN}) = (1,0), (2,0), (3,1), (4,3), (7,2)$. Normally $\mathrm{LZ}_{\boldsymbol{x}}$ is computed by first computing POS and LEN as arrays $\mathrm{POS}[1..n]$ and $\mathrm{LEN}[1..n]$, where $\mathrm{POS}[i] = j < i$, $j > 0$, means that the longest match for a prefix of suffix i of $\boldsymbol{x}$ that occurs left of i in $\boldsymbol{x}$ is at position $j = \mathrm{POS}[i]$ and has length $\mathrm{LEN}[i]$; $\mathrm{POS}[i] = 0$ means that i is the leftmost occurrence of letter $\boldsymbol{x}[i]$ in $\boldsymbol{x}$. As mentioned above, $\mathrm{LZ}_{\boldsymbol{x}}$ can be quickly computed from $\mathrm{ST}_{\boldsymbol{x}}$ in $\Theta(n)$ time [33], also from $\mathrm{SA}_{\boldsymbol{x}}$ [1]. Our new algorithm is displayed in Figure 1.

The basic strategy of CPS is first to locate, in a left-to-right traversal of SA, a next position i_2 such that $\mathrm{LCP}[i_2] > \mathrm{LCP}[i_3]$ for some least $i_3 > i_2$; then second to backtrack (using stack S) from i_2, setting $\mathrm{POS}[p_2] \leftarrow p1$ or $\mathrm{POS}[p_1] \leftarrow p2$ according as $p_1 = \mathrm{SA}[i_1] < p_2 = \mathrm{SA}[i_2]$ or not. until the LCP

```
— Using SAx and LCPx, compute POS[1..n] and LEN[1..n].
i1 ← 1; i2 ← 2; i3 ← 3
while i3 ≤ n+1 do
  — Identify the next position i2 < i3 with LCP[i2] > LCP[i3].
    while LCP[i2] ≤ LCP[i3] do
        push(S, i1); i1 ← i2; i2 ← i3; i3 ← i3+1
  — Backtrack using the stack S to locate the first i1 < i2 such that
  — LCP[i1] < LCP[i2], at each step setting the larger position in POS
  — corresponding to equal LCP to point leftwards to the smaller one,
  — if it exists; if not, then POS[i] ← i.
    p2 ← SA[i2]; ℓ2 ← LCP[i2]
    assign(POS, LEN, p2)
    while LCP[i1] = ℓ2 do
        i1 ← pop(S)
        assign(POS, LEN, p2)
    SA[i1] ← p2
  — Reset pointers for the next stage.
    if i1 > 1 then
        i2 ← i1; i1 ← pop(S)
    else
        i2 ← i3; i3 ← i3+1

procedure assign(POS, LEN, p2)
p1 ← SA[i1]
if p1 < p2 then
    POS[p2] ← p1; LEN[p2] ← ℓ2; p2 ← p1
else
    POS[p1] ← p2; LEN[p1] ← ℓ2
```

Fig. 1. Algorithm CPS: computing LZ$\boldsymbol{x}$

value for the position i_1 popped from S falls below LCP$[i_2]$. This processing does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the *leftmost* occurrence in $\boldsymbol{x}$, the norm for LZ factorization; however, the Main and KK runs algorithms do not require this property for their correct functioning, they require only that each position in POS should point left. In other terminology, what is in fact computed by CPS is a ***quasi suffix array*** (QSA) [6]. We call the algorithm of Figure 1 CPSa.

CPSa maintains the invariant that $i_1 < i_2 < i_3$, terminating when i_3 is incremented beyond $n+1$. There are two main stages corresponding to two simple inner **while** loops. The first of these pushes all entries i_1 (actually, the previous value of i_2) onto S until LCP$[i_2] >$ LCP$[i_3]$. The second **while** loop assigns

$$\mathrm{POS}\big[\max\{p_1,p_2\}\big] \leftarrow \min\{p_1,p_2\}$$

(thus ensuring that POS always points left) corresponding to the current LCP value, until that value changes.

Now observe that none of the position pointers i_1, i_2, i_3 will ever point to any position i in SA such that POS$\big[$SA$[i]\big]$ has been previously set. It follows that the storage for SA and LCP can be dynamically reused to specify the location and contents of the array POS, thus saving $4n$ bytes of storage — neither the Main nor the KK algorithm requires SA/LCP. In Figure 1 this is easily accomplished by inserting $i_2 \leftarrow i_1$ at the beginning of the second inner **while** loop, then replacing

$$\mathrm{POS}[p_2] \leftarrow p_1 \ \text{ by } \ \mathrm{SA}[i_2] \leftarrow p_2;\ \mathrm{LCP}[i_2] \leftarrow p_1$$
$$\mathrm{POS}[p_1] \leftarrow p_2 \ \text{ by } \ \mathrm{SA}[i_2] \leftarrow p_1;\ \mathrm{LCP}[i_2] \leftarrow p_2$$

POS can then be computed by a straightforward in-place compactification of SA and LCP into SA (now redefined as POS). We call this second algorithm CPSb.

But more storage can be saved. Remove all reference to LEN from CPSb, so that it computes only POS and in particular allocates no storage for LEN. Then, after POS is computed, the space previously required for LCP becomes free and can be reallocated to LEN. Observe that only those positions in LEN that are required for the LZ-factorization need to be computed, so that the total computation time for LEN is $\Theta(n)$. In fact, without loss of efficiency, we can avoid computing LEN as an array and compute it only when required; given a sentinel value $\mathrm{POS}[n+1] = \$$, the simple function of Figure 2 computes LEN corresponding to POS$[i]$. We call the third version CPSc.

function LEN($\boldsymbol{x}$, POS, i)
$j \leftarrow \mathrm{POS}[i]$
if $j = i$ **then**
  LEN $\leftarrow 0$
else
  $\ell \leftarrow 1$
  while $\boldsymbol{x}[i+\ell] = \boldsymbol{x}[j+\ell]$ **do**
    $\ell \leftarrow \ell+1$
  LEN $\leftarrow \ell$

Fig. 2. Computing LEN corresponding to POS$[i]$

Since at least one position in POS is set at each stage of the main **while** loop, it follows that the execution time of CPS is linear in n. For CPSa space requirements total $17n$ bytes (for $\boldsymbol{x}$, SA, LCP, POS & LEN) plus $4s$ bytes for a stack of maximum size s. For $\boldsymbol{x} = a^n$, $s = n$, but in practical cases s will be close to the maximum height of $\mathrm{SA}_{\boldsymbol{x}}$ and so s is bounded by $O(\log_\alpha n)$ [10].

For CPSb and CPSc, the minimum space required is $13n$ and $9n$ bytes, respectively, plus stack. Observe that for CPSa and CPSb the original (and somewhat faster) method [11] for computing LCP can be used, since it requires $13n$ bytes of storage, not greater than the total space requirements of these two variants. For CPSc, however, to achieve $9n$ bytes of storage, the Manzini variant [23] for computing LCP must be used. In fact, as described below, we test two versions of CPSc, one that uses the original LCP calculation (and therefore requires no additional space for the stack), the other using the Manzini variant (CPSd).

We remark that all versions of Algorithm CPS can easily be modified (with the introduction of another stack) to compute the LZ factorization in its usual form.

3 Experimental Results

We implemented the three versions of CPS described above, with two variants of CPSc; we call them `cpsa`, `cpsb`, `cpsc` ($13n$-byte LCP calculation), and `cpsd` ($9n$-byte LCP calculation). We also implemented the other SA-based LZ-factorization algorithm, `ako` of [1]. The implementation `kk-lz` of Kolpakov and Kucherov's algorithm was obtained from [13]. All programs were written in C or C++. We are confident that all implementations tested are of high quality.

All experiments were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. All running times given are the average of four runs and do not include time spent reading input files. Times were recorded with the standard C `getrusage` function. Memory usage was recorded with the `memusage` command available with most Linux distributions.

Times for the `cps` implementations and `ako` include time required for SA and LCP array construction. The implementation of `kk-lz` is only suitable for strings on small alphabets ($|\Sigma| \leq 4$) so times are only given for some files. File `chr22` was originally on an alphabet of five symbols A,C,G,T,N but was reduced by one of replacing occurrences of N randomly by the other four symbols. The N's represent ambiguities in the sequencing process. Results are not given for `ako` and `kk-lz` on some files because the memory required exceeded the capacity of the test machine.

We conclude:

(1) If speed is the main criterion, KK-LZ remains the algorithm of choice for DNA strings of moderate size.
(2) For other strings encountered in practice, CPSb is consistently faster than AKO except for some strings on very large alphabets; it also uses substantially less space, especially on run-rich strings.
(3) Overall, and especially for strings on alphabets of size greater than 4, CPSd is probably preferable since it will be more robust for main-memory use on very large strings: its storage requirement is consistently low (about half that of AKO, including on DNA strings) and it is only 25–30% slower than CPSb (and generally faster than AKO).

4 Discussion

The algorithms presented here make use of full-size suffix arrays, but there have been many "succinct" or "compressed" suffix structures proposed [21,8,28] that make use of as little as n bytes. We wish to explore the use of such structures in

Table 1. Description of the data set used in experiments

String	Size (bytes)	Σ	# runs	Description
fib35	9227465	2	7049153	The 35th Fibonacci string (see [29])
fib36	14930352	2	11405771	The 36th Fibonacci string
fss9	2851443	2	2643406	The 9th run rich string of [7]
fss10	12078908	2	11197734	The 10th run rich string of [7]
rnd2	8388608	2	3451369	Random string, small alphabet
rnd21	8388608	21	717806	Random string, larger alphabet
ecoli	4638690	4	1135423	E.Coli Genome
chr22	34553758	4	8715331	Human Chromosome 22
bible	4047392	62	177284	King James Bible
howto	39422105	197	3148326	Linux Howto files
chr19	63811651	4	15949496	Human Chromosome 19

Table 2. Runtime in milliseconds for suffix array construction and LCP computation

String	`saca`	`lcp13n`	`lcp9n`
fib35	5530	2130	3090
fib36	10440	3510	5000
fss9	1490	660	960
fss10	8180	2810	4070
rnd2	2960	2360	3030
rnd21	2840	2620	3250
ecoli	1570	1340	1700
chr22	14330	12450	16190
bible	1140	1020	1270
howto	12080	11750	14490
chr19	28400	25730	31840

Table 3. Runtime in milliseconds (in parentheses peak memory usage in bytes per input symbol) for the LZ-factorization algorithms. Underlining indicates least time/space.

String	`cpsa`	`cpsb`	`cpsc`	`cpsd`	`ako`	`kk-lz`
fib35	9360 (19.5)	8560 (15.5)	9240 (13.0)	10200 (11.5)	12870 (26.9)	10060 (19.9)
fib36	16730 (19.5)	15420 (15.5)	16240 (13.0)	17730 (11.5)	23160 (26.9)	18680 (20.8)
fss9	2680 (19.1)	2430 (15.1)	2690 (13.0)	2990 (11.1)	3740 (25.4)	1270 (21.3)
fss10	13240 (19.1)	12170 (15.1)	13390 (13.0)	14650 (11.1)	17890 (25.4)	7850 (22.5)
rnd2	6950 (17.0)	6130 (13.0)	7010 (13.0)	7680 (9.0)	9920 (17.0)	9820 (11.8)
rnd21	7100 (17.0)	6270 (13.0)	7130 (13.0)	7760 (9.0)	7810 (17.0)	– (–)
ecoli	3800 (17.0)	3350 (13.0)	3830 (13.0)	4190 (9.0)	4740 (17.0)	1610 (11.0)
chr22	35240 (17.0)	30320 (13.0)	36480 (13.0)	40220 (9.0)	65360 (17.0)	18240 (11.1)
bible	2930 (17.0)	2540 (13.0)	2970 (13.0)	3220 (9.0)	3670 (17.0)	– (–)
howto	32150 (17.0)	27750 (13.0)	33760 (13.0)	36500 (9.0)	23830 (17.0)	– (–)
chr19	70030 (17.0)	61230 (13.0)	71910 (13.0)	78020 (9.0)	– (–)	40420 (11.1)

this context. More generally, we note that all algorithms that compute runs or repetitions need to compute all the information required for ***repeats*** — that is, not necessarily adjacent repeating substrings. Since runs generally occur sparsely in strings [14], it seems that they should somehow be computable with less heavy machinery. Recent results [7,27,4] may suggest more economical methods. In the shorter term, we are working on methods that compute the LCP as a byproduct of SA construction, also those that bypass LCP computation.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algs. 2, 53–86 (2004)
2. Apostolico, A., Preparata, F.P.: Optimal off-line detection of repetitions in a string. Theoret. comput. sci. 22, 297–315 (1983)
3. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inform. process. lett. 12(5), 244–250 (1981)
4. Fan, K., Puglisi, S.J., Smyth, W.F., Turpin, A.: A new periodicity lemma. SIAM J. Discrete Math. 20(3), 656–668 (2006)
5. Martin Farach, Optimal suffix tree construction with large alphabets Proc. 38^{th} FOCS pp. 137–143 (1997)
6. Franek, F., Holub, J., Smyth, W.F., Xiao, X.: Computing quasi suffix arrays. J. Automata, Languages & Combinatorics 8(4), 593–606 (2003)
7. Franek, F., Simpson, R. J., Smyth, W. F.: The maximum number of runs in a string. In: Miller, M., Park, K.(eds.) Proc. 14^{th} AWOCA, pp. 26–35 (2003)
8. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing & string matching. SIAM J. Computing 35(2), 378–407 (2005)
9. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. 30^{th} ICALP. pp. 943–955 (2003)
10. Karlin, S., Ghandour, G., Ost, F., Tavare, S., Korn, L.J.: New approaches for computer analysis of nucleic acid sequences. Proc. Natl. Acad. Sci. USA 80, 5660–5664 (1983)
11. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, Springer, Heidelberg (2001)
12. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, Springer, Heidelberg (2003)
13. Kolpakov, R., Kucherov, G.: `http://bioinfo.lifl.fr/mreps/`
14. Kolpakov, R., Kucherov, G.: On maximal repetitions in words. J. Discrete Algs. 1, 159–186 (2000)
15. Kurtz, S.: Reducing the space requirement of suffix trees. Software Practice & Experience 29(13), 1149–1171 (1999)
16. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Trans. Information Theory 22, 75–81 (1976)
17. Lentin, A., Schützenberger, M.P.: A combinatorial problem in the theory of free monoids, Combinatorial Mathematics & Its Applications. In: Bose, R.C., Dowling, T.A. (eds.) University of North Carolina Press, pp. 128–144 (1969)
18. Main, M.G.: Detecting leftmost maximal periodicities. Discrete Applied Maths 25, 145–153 (1989)

19. Main, M.G., Lorentz, R.J.: An O(n log n) Algorithm for Recognizing Repetition, Tech. Rep. CS-79–056, Computer Science Department, Washington State University (1979)
20. Main, M.G., Lorentz, R.J.: An O(nlog n) algorithm for finding all repetitions in a string. J. Algs. 5, 422–432 (1984)
21. Mäkinen, V., Navarro, G.: Compressed full-text indices, ACM Computing Surveys (to appear)
22. Maniscalco, M., Puglisi, S.J.: Faster lightweight suffix array construction. In: Ryan, J., Dafik (eds.) Proc. 17^{th} AWOCA pp. 16–29 (2006)
23. Manzini, G.: Two space-saving tricks for linear time LCP computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, Springer, Heidelberg (2004)
24. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica 40, 33–50 (2004)
25. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. Assoc. Comput. Mach. 32(2), 262–272 (1976)
26. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms, ACM Computing Surveys (to appear)
27. Rytter, W.: The number of runs in a string: improved analysis of the linear upper bound. In: Durand, B., Thomas, W. (eds.) Proc. 23^{rd} STACS. LNCS, vol. 2884, pp. 184–195. Springer, Heidelberg (2006)
28. Sadakane, K.: Space-efficient data structures for flexible text retrieval systems. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, Springer, Heidelberg (2002)
29. Smyth, B.: Computing Patterns in Strings, Pearson Addison-Wesley, p. 423 (2003)
30. Thue, A.: Über unendliche zeichenreihen. Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7, 1–22 (1906)
31. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
32. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th Annual IEEE Symp. Switching & Automata Theory, pp. 1–11 (1973)
33. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Information Theory 23, 337–343 (1977)