

# Fast & Practical Algorithms for Computing All the Runs in a String

*Gang Chen\**, *Simon J. Puglisi†* & *Bill Smyth\*†*

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

\* Algorithms Research Group, Department of Computing & Software, McMaster University

† Department of Computing, Curtin University

## Runs

A string  $u$  is a **run** iff it is periodic of (minimum) **period**  $p \leq |u|/2$ . Thus

$$x = abaabaabaabaab = (aba)^4 ab$$

is a run of period  $|aba| = 3$ . A substring  $u = x[i..j]$  of  $x$  is a **run in  $x$**  iff it is a run of period  $p$  and neither  $x[i-1..j]$  nor  $x[i..j+1]$  is a run of period  $p$  (**nonextendible**). The run  $u$  has **exponent**  $e = \lfloor |u|/p \rfloor$  and possibly empty **tail**  $t = x[i+ep..j]$  (proper prefix of  $x[i..i+p-1]$ ). Thus

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$x =$	$b$	$a$	$a$	$a$	$b$	$a$	$a$	$b$	$a$	$a$	$b$	$a$	$b$	$a$

has a run  $x[3..12]$  of period  $p = 3$  and exponent  $e = 3$  with tail  $t = a$  of length  $t = |t| = 1$ . 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. Computing all the runs in  $x$  specifies all the repetitions in  $x$ .

## Computing Runs

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

- (1) computing  $ST_x$ , the suffix tree of  $x$ ;
- (2) using  $ST_x$  to compute  $LZ_x$ , the Lempel-Ziv decomposition of  $x$ ;
- (3) using  $LZ_x$  to compute leftmost runs.

Kolpakov & Kucherov (KK) proved that the maximum number of runs in any string of length  $n$  is  $\Theta(n)$ , and showed how to compute all the runs in  $x$  from the leftmost ones in linear time.

Abouelhoda, Kurtz & Ohlebusch (AKO) show how to compute  $LZ_x$  from a suffix array  $SA_x$ , together with other linear structures, rather than from  $ST_x$ .

## $SA_x, LCP_x$

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

$$LCP_x[j] = \left| \text{lcp}_x(SA_x[j-1], SA_x[j]) \right|.$$

Given  $x$  and  $SA_x$ ,  $LCP_x$  can be quickly computed in  $\Theta(n)$  time (Manzini). For example:

	1	2	3	4	5	6	7	8	9
$x =$	$a$	$b$	$a$	$a$	$b$	$a$	$b$	$a$	
$SA_x =$	8	3	6	1	4	7	2	5	
$LCP_x =$	-1	1	1	3	3	0	2	2	-1

$$\underline{\text{LZ}_x = (\text{POS}, \text{LEN})}$$

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

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

Typically, integer pairs (POS, LEN) specify the decomposition, where POS gives a position in  $x$  and LEN the corresponding length at that position (by convention zero if the position contains a “new” letter).  $x = abaababa$  gives  $(\text{POS}, \text{LEN}) = (1, 0), (2, 0), (3, 1), (4, 3), (7, 2)$  for  $a/b/a/aba/ba$ .

## Generic Algorithm CPS

— From SA & LCP compute POS[1..n] & LEN[1..n].

$i_1 \leftarrow 1; i_2 \leftarrow 2; i_3 \leftarrow 3$

**while**  $i_3 \leq n+1$  **do**

— Find next  $i_2 < i_3$  with  $LCP[i_2] > LCP[i_3]$ .

**while**  $LCP[i_2] \leq LCP[i_3]$  **do**

    push( $S, i_1$ );  $i_1 \leftarrow i_2; i_2 \leftarrow i_3; i_3 \leftarrow i_3 + 1$

— Stack  $S$ : find first  $i_1 < i_2$  with  $LCP[i_1] < LCP[i_2]$ ,

— at each step resetting larger position in POS

— to point leftward to smaller position.

$p_2 \leftarrow SA[i_2]; \ell_2 \leftarrow LCP[i_2]$

  assign(POS, LEN,  $p_1, p_2$ )

**while**  $LCP[i_1] = \ell_2$  **do**

$i_1 \leftarrow \text{pop}(S)$

    assign(POS, LEN,  $p_1, p_2$ )

$SA[i_1] \leftarrow p_2$

— Reset pointers for next stage.

**if**  $i_1 > 1$  **then**

$i_2 \leftarrow i_1; i_1 \leftarrow \text{pop}(S)$

**else**

$i_2 \leftarrow i_3; i_3 \leftarrow i_3 + 1$

**procedure** assign(POS, LEN,  $p_1, p_2$ )

$p_1 \leftarrow SA[i_1]$

**if**  $p_1 < p_2$  **then**

  POS[ $p_2$ ]  $\leftarrow p_1$ ; LEN[ $p_2$ ]  $\leftarrow \ell_2$ ;  $p_2 \leftarrow p_1$

**else**

  POS[ $p_1$ ]  $\leftarrow p_2$ ; LEN[ $p_1$ ]  $\leftarrow \ell_2$

## Remarks

- \* The output of CPS (POS & LEN) is input to the Main/KK algorithms.
- \* CPS does not guarantee that, for equal LCP (LEN), each corresponding position in POS necessarily points to the *leftmost* occurrence in  $x$ , as normally required for LZ decomposition — but Main/KK do not require this property! Actually, CPSa computes a **quasi suffix array** (QSA).
- \* Since Main/KK do not require SA/LCP, this storage can be dynamically reused to compute POS – this is CPSb.
- \* Moreover, all reference to LEN can be removed from CPS. Then, after POS has been computed, LEN can be computed in the space previously used for LCP. Since only the positions in LEN needed for the LZ decomposition are required, LEN can be computed in  $\Theta(n)$  time. This is CPSc.
- \* All versions of CPS require  $\Theta(n)$  time and expected space  $8 \log_{\alpha} n$  for the stack  $S$ . Additional space: CPSa ( $17n$  bytes), CPSb ( $13n$  bytes), CPSc ( $9n$  bytes).
- \* All versions of CPS can easily be modified (with the introduction of another stack) to compute LZ in its usual form.

## Test Cases

String	Size (bytes)	$\Sigma$	# runs	Description
fibonacci35	9227465	2	7049153	Fibonacci string 35
fibonacci36	14930352	2	11405771	Fibonacci string 36
fss9	2851443	2	2643406	Run Rich 9
fss10	12078908	2	11197734	Run Rich 10
random2	8388608	2	3451369	Random, $ \Sigma  = 2$
random21	8388608	21	717806	Random, $ \Sigma  = 21$
ecoli	4638690	4	1135423	E.Coli Genome
chr22	34553758	4	8715331	Chromosome 22
bible	4047392	62	177284	King James
howto	39422105	197	3148326	Linux Howto files
chr19	63811651	4	15949496	Chromosome 19

## Test Results: ms(bytes/letter)

String	CPSb	CPSc	AKO	KK
fib35	<u>8560</u> (15.5)	10200(11.5)	12870(26.9)	10060(19.9)
fib36	<u>15420</u> (15.5)	17730(11.5)	23160(26.9)	18680(20.8)
fss9	<u>2430</u> (15.1)	2990(11.1)	3740(25.4)	<u>1270</u> (21.3)
fss10	12170(15.1)	14650(11.1)	17890(25.4)	<u>7850</u> (22.5)
random2	<u>6130</u> (13.0)	7680(9.0)	9920(17.0)	9820(11.8)
random21	<u>6270</u> (13.0)	7760(9.0)	7810(17.0)	-(-)
ecoli	<u>3350</u> (13.0)	4190(9.0)	4740(17.0)	<u>1610</u> (11.0)
chr22	30320(13.0)	40220(9.0)	65360(17.0)	<u>18240</u> (11.1)
bible	<u>2540</u> (13.0)	3220(9.0)	3670(17.0)	-(-)
howto	27750(13.0)	36500(9.0)	<u>23830</u> (17.0)	-(-)
chr19	61230(13.0)	78020(9.0)	-(-)	40420(11.1)

We conclude:

- (1) KK remains the algorithm of choice for DNA strings of moderate size (only handles  $|\Sigma| \leq 4$ ).
- (2) For all strings except those with very large alphabets, CPSb is consistently faster than AKO; it also uses substantially less space, especially on run-rich strings.
- (3) Overall, and especially for strings on alphabets of size greater than 4, CPSc 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.

## Future Work

- (1) Use of “succinct” or “compressed” suffix structures.
- (2) Compute the LCP as a byproduct of SA construction.
- (3) Construct LZ without computing LCP.
- (4) Runs are generally sparse in strings, but computing them requires all the heavy machinery needed to compute repeats. Perhaps it is possible to compute them directly, without recourse to the SA.