

Fast Optimal Algorithms for Computing All the Repeats in a String

Simon J. Puglisi¹, W. F. Smyth^{2,3} and Munina Yusufu²

¹School of Computer Science & Information Technology
RMIT University, Melbourne, Australia
email: sjp@cs.rmit.edu.au

²Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada
email: {smyth,yusufum}@mcmaster.ca

³Digital Ecosystems & Business Intelligence Institute
Curtin University, Perth, Western Australia

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

repeating substring/repeat/NE repeat

A **repeating substring** u in a string x is a substring of x that occurs more than once.

A **repeat** $(p; i_1, i_2, \dots, i_k)$, $k \geq 2$, is a set of repeating substrings of **period** (length) p that occur at positions i_1, i_2, \dots, i_k in x — **complete** if it includes all occurrences in x .

A repeat is **left-extensible** (LE) if

$$x[i_1 - 1] = x[i_2 - 1] = \dots = x[i_k - 1],$$

right-extensible (RE) if

$$x[i_1 + p] = x[i_2 + p] = \dots = x[i_k + p],$$

nonextendible (NE) if neither LE nor RE (both **NLE** and **NRE**).

supernonextendible

A repeat is **supernonextendible** (SNE) if it is NE and its repeating substring u is not a substring of any other repeating substring of x .

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

(3; 1, 4, 6, 9) (that is, aba) is NE;

(5; 1, 6, 9) (that is, $abaab$) is SNE.

Requiring that repeats be NE/SNE avoids redundant output.

Our Algorithms

Given a string \mathbf{x} and a positive integer p_{min} , we present two algorithms:

- PSY1 computes all the complete NE repeats of period $p \geq p_{min}$;
- PSY2 computes all the complete SNE repeats of period $p \geq p_{min}$.

Both of these algorithms execute in time linear in string length independent of alphabet size; both of them require computation of SA (suffix array) and LCP (longest common prefix array) for \mathbf{x} ; both of them output triples $(p; i, j)$, where $i..j$ is a range of positions in SA.

Applications

NE/SNE repeats are useful in various contexts:

- phrase selection in off-line data compression [AL00, LM00, TS02];
- duplicate text/document detection [BZ06];
- genome analysis and sequence alignment [B99, SK05, BIMSTTT07, ISY08].

NE pairs of repeats

Given a string $\mathbf{x} = \mathbf{x}[1..n]$ on an alphabet of size α , some algorithms compute NE *pairs* of repeats:

- Gusfield [G97] uses suffix trees, requires $O(\alpha n + q)$ time, where q is the number of outputs;
- Brodal *et al.* [BLPS00] use similar methods and introduce bounds on the “gaps” between repeating substrings;
- Abouelhoda *et al.* [AKO04] use suffix arrays and also require $O(\alpha n + q)$ time.

All of these algorithms require $O(n^2)$ time in the worst case (for $\alpha \in \Theta(n)$).

complete NE/SNE repeats

Two recent algorithms [FST03, NIBT07] use suffix arrays to compute all complete NE repeats in \mathbf{x} in $O(n)$ time and space independent of α . In practice PSY1 uses substantially less time and space than either of them.

For all complete SNE repeats:

- Gusfield [G97] uses suffix trees and $O(n \log \alpha)$ time;
- Abouelhoda *et al.* [AKO04] use suffix arrays and $O(n + \alpha^2)$ time.

PSY2 requires $O(n + r\alpha)$ time, where r is the number of SNRE repeats and $r\alpha < n$.

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms**
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 Conclusions

preprocessing-1

Both PSY1 (complete NE repeats) and PSY2 (complete SNE repeats) need to compute SA and LCP. In addition, PSY1 requires BWT [BW94], where for $SA[j] > 1$, $BWT[j] = \mathbf{x}[SA[j] - 1]$, while for j such that $SA[j] = 1$, $BWT[j] = \$$, a sentinel.

	1	2	3	4	5	6	7	8	9
$\mathbf{x} =$	a	b	a	a	b	a	b	a	$\$$
SA =	8	3	6	1	4	7	2	5	
LCP =	-1	1	1	3	3	0	2	2	-1
BWT =	b	b	b	$\$$	a	a	a	a	

SA and LCP are arrays of integers ($4n$ bytes), BWT of letters (n bytes).

preprocessing-2

All of these data structures can be computed in $O(n)$ time: for SA see [KA03, KS03], for LCP see [KLAAP01, M04].

However, for SA, the fastest (and most space-efficient: $\leq 6n$ bytes) algorithms are supralinear in the worst case [PST07, MF04, MP06].

In terms of time, SA construction is the main obstacle; in terms of space, LCP construction ($13n$ bytes for [KLAAP01], $9n$ bytes for the slightly slower Manzini variant) is the problem.

Instead of BWT, PSY2 requires an array $LAST = LAST[1..n]$ of byte (explained later).

PSY1-1

PSY1 uses only LCP and BWT ($5n$ bytes) for its execution. It performs a single left-to-right scan of LCP, looking for the left boundary lb of a range of high LCP value. If $lcp = LCP[lb+1] > LCP[lb]$, then a triple (lcp, lb, bwt) is pushed onto a stack LB, where $bwt = \$$ if $BWT[lb] \neq BWT[lb+1]$, otherwise equals $BWT[lb]$.

Thus the stack entry specifies the left boundary lb of a repeat of period lcp that is certainly NLE if $bwt = \$$.

Pops of LB occur at positions j where LCP decreases: $LCP[j+1] < top(LB).lcp$. Then j is the right boundary of the repeat. Furthermore, the repeat is NRE: if the same letter followed each occurrence of the repeating substring, the LCP value for all of them would be greater by at least one.

PSY1-2

So every pop identifies an NRE repeat — if in addition it is NLE ($bwt = \$$), we should output the NE repeat (lcp, lb, j) .

To ensure that the bwt value on the stack is correct, we update it according to $BWT[j+1]$ at each position j such that the LCP value does not decrease. This simple approach works because of a basic property of LCP arrays:

- Two ranges of repeats are either disjoint (empty common prefix) or else one range contains the other (common prefix over the longer range).

Thus the range currently popped is always the one of greater LCP (the one most recently placed on the stack).

PSY2-1

In PSY2 we look first for the left boundary i of a repeat whose minimum LCP value p is locally greatest; the right boundary is determined at the first subsequent position j for which $LCP[j+1] > LCP[j]$. The repeat $(p; i, j)$ is NRE. It is SNLE if and only if for every $h \in i..j$ every left extension $BWT[h]$ is distinct.

This condition can hold only if $j-i+1 \leq \alpha$.

To test efficiently for this condition we introduce an array $LAST = LAST[1..n]$ of byte, where for every $j \in 1..n$, $LAST[j]$ is the offset between the letter $BWT[j]$ and the rightmost prior occurrence of $BWT[j]$ in SA; if there is no such occurrence, or if the offset $\geq \alpha$, then $LAST[j] \leftarrow \alpha - 1$.

$LAST$ can be computed in $\Theta(n)$ time by a simple left-to-right scan of SA.

PSY2-2

Using LAST, the pseudocode for PSY2 is straightforward:

```
— Preprocessing: compute SA, LAST & LCP.  $j \leftarrow 0$ ;  $p \leftarrow -1$ ;  $q \leftarrow 0$   
while  $j < n$  do  
   $high \leftarrow 0$   
  repeat  
     $j \leftarrow j+1$ ;  $p \leftarrow q$ ;  $q \leftarrow LCP[j+1]$   
    if  $q > p$  then  $high \leftarrow q$ ;  $i \leftarrow j$   
  until  $p > q$   
  if  $high > 0$  and SNLE( $i, j, LAST$ ) then  
    output ( $p; i, j$ )  
  
function SNLE( $start, end, LAST$ )  
   $k \leftarrow end - start + 1$   
  if  $k > \alpha$  then return FALSE  
  else  
    for  $h \leftarrow start + 1$  to  $end$  do  
      if  $h - LAST[h] > start$  then return FALSE  
    return TRUE
```

Figure: Algorithm PSY2 with a simplified SNLE function using LAST

PSY1

Both PSY1 and its preprocessing require $\Theta(n)$ time independent of alphabet size.

Because PSY1's output is compact (ranges $(p; i, j)$ in the suffix array), it requires only $5n$ bytes of storage for its execution, plus storage for the stack LB (9-byte entries). Expected stack depth is $2 \log_{\alpha} n$ entries [KGOTK83], thus altogether an additional $18 \log_{\alpha} n$ bytes (for $\alpha = 2$, $n = 2^{20}$, 360 bytes).

Preprocessing for PSY1 can be as little as $9n$ bytes (Manzini's LCP calculation). If PSY1 were required to output positions in \mathbf{x} rather than SA, the processing could be handled as postprocessing: either input SA to overwrite LCP (much extra time but no extra storage), or store SA throughout (little extra time but overall $9n$ bytes of storage).

PSY2

Preprocessing for PSY2 (SA,LCP,LAST) has space and time requirements identical to those for PSY1 (SA,LCP,BWT). PSY2 itself uses $5n$ bytes with no stack and requires $O(n+r\alpha)$ time, $r\alpha < n$.

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments**
- 4 Conclusions

testbed

Tests were conducted using a 2.6GHz Opteron 885 processor (64-bit architecture) with 2GB main memory available, under Red Hat Linux 4.1.2–14. The compiler was `gcc` with the `-O3` option. The run times used were the minima over four runs, not including input/output.

For SA construction the linear time algorithm of Kärkkäinen/Sanders [KS03] was used; on most inputs, the worst-case supralinear algorithm of Maniscalco/Puglisi [MP06] is probably about five times faster, while using half the space ($5.2n$ bytes).

For LCP calculation, the algorithm of [KLAAP01] was used; Manzini's variant is probably 5–10% slower.

files

File Type	Name	No. Bytes	Description
highly periodic	fibonacci	9,227,465	Fibonacci
	fibonacci	14,930,352	Fibonacci
	fss9	2,851,443	run-rich [FSS03]
	fss10	12,078,908	run-rich [FSS03]
random	rand2	8,388,608	$\alpha = 2$
	rand21	8,388,608	$\alpha = 21$
DNA	ecoli	4,638,690	<i>escherichia coli</i> genome
	chr22	34,553,758	human chromosome 22
	chr19	63,811,651	human chromosome 19
Genbank protein database	prot-a	16,777,216	sample
	prot-b	33,554,432	sample
English	bible	4,047,392	King James version
	howto	39,422,105	Linux howto files
	mozilla	51,220,480	Mozilla source code

Table: Files used for testing.

results

File	SA	LCP	BWT	LAST	PSY1	[NIBT07]	PSY2
fibonacci35	0.898	0.169	0.025	0.031	0.012	0.448	0.009
fibonacci36	0.886	0.170	0.027	0.033	0.012	0.475	0.007
fss9	0.826	0.154	0.026	0.031	0.014	0.330	0.007
fss10	0.958	0.177	0.025	0.032	0.013	0.469	0.008
periodic AVG	0.892	0.168	0.026	0.032	0.013	0.430	0.008
rand2	0.947	0.188	0.026	0.031	0.017	0.215	0.012
rand21	1.135	0.199	0.025	0.031	0.012	0.122	0.012
random AVG	1.041	0.193	0.025	0.031	0.015	0.169	0.012
ecoli	1.413	0.175	0.025	0.031	0.015	0.155	0.011
chr22	1.635	0.285	0.035	0.040	0.016	0.278	0.012
chr19	1.873	0.333	0.044	0.053	0.016	0.242	0.012
DNA AVG	1.754	0.309	0.035	0.041	0.016	0.225	0.012
prot-a	1.778	0.222	0.027	0.032	0.013	0.211	0.012
prot-b	1.971	0.277	0.034	0.039	0.013	0.247	0.012
protein AVG	1.874	0.249	0.030	0.036	0.013	0.229	0.012
bible	1.417	0.151	0.024	0.030	0.015	0.168	0.012
howto	1.912	0.214	0.035	0.039	0.016	0.219	0.012
mozilla	1.815	0.187	0.032	0.036	0.013	0.139	0.011
English AVG	1.417	0.151	0.024	0.035	0.014	0.175	0.012
AVERAGE	1.390	0.207	0.029	0.035	0.014	0.266	0.011

Table: Microseconds per letter used by each run.

Outline

- 1 Introduction
 - Definitions
 - Applications
 - Previous Algorithms
- 2 The Algorithms
 - Preprocessing
 - PSY1
 - PSY2
 - Time & Space
- 3 Experiments
- 4 **Conclusions**





conclusions

We have described algorithms PSY1 and PSY2 that are more space- and time-efficient, both in theory and practice, than algorithms previously proposed. Moreover, both have very stable execution times, dependent primarily on string length rather than string structure or alphabet size.

We note that the output of PSY1 can be efficiently postprocessed to yield NE pairs of repeat, if required.





We would like to have

- an SA construction algorithm that is linear, lightweight and fast;
- a fast and lightweight LCP algorithm.

-  Mohamed Ibrahim Abouelhoda, Stefan Kurtz & Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algs.* 2 (2004) 53–86.
-  Alberto Apostolico & Stefano Lonardi, **Off-line compression by greedy textual substitution**, *Proc. IEEE* 88–11 (2000) 1733–1744.
-  A. Bakalis, C.S. Iliopoulos, C. Makris, S. Sioutas, E. Theodoridis, A. Tsakalidis & K. Tsihlias, **Locating maximal multirepeats in multiple strings under various constraints**, *The Computer Journal* 50–2 (2007) 178–185.
-  G. Benson, **Tandem repeats finder: a program to analyze DNA sequences**, *Nucleic Acids Res.* 27 (1999) 573–580.





-  Yaniv Berstein & Justin Zobel, **Accurate discovery of co-derivative documents via duplicate text detection**, *Information Systems* 31 (2006) 595–609.
-  Gerth S. Brodal, Rune B. Lyngso, Christian N. S. Pederesen & Jens Stoye, **Finding maximal pairs with bounded gap**, *J. Discrete Algs.* 1 (2000) 77–103.
-  Michael Burrows & David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Technical Report 124, Digital Equipment Corporation (1994).
-  Frantisek Franek, R. J. Simpson & W. F. Smyth, **The maximum number of runs in a string**, *Proc. 14th Australasian Workshop on Combinatorial Algs.*, Mirka Miller & Kunsoo Park (eds.) (2003) 36–45.

-  Frantisek Franek, W. F. Smyth & Yudong Tang, **Computing all repeats using suffix arrays**, *J. Automata, Languages & Combinatorics* 8–4 (2003) 579–591.
-  Dan Gusfield, *Algorithms on Strings, Trees & Sequences*, Cambridge University Press (1997) 534 pp.
-  Costas S. Iliopoulos, W. F. Smyth & Munina Yusufu, *Faster algorithms for computing maximal multirepeats in multiple sequences*, submitted for publication (2008).
-  S. Karlin, G. Ghandour, F. Ost, S. Tavare & L. J. Korn, **New approaches for computer analysis of nucleic acid sequences**, *Proc. Natl. Acad. Sci. USA* 80 (1983) 5660–5664.

-  Juha Kärkkäinen & Peter Sanders, **Simple linear work suffix array construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming* (2003) 943–955.
-  S. Karlin, G. Ghandour, F. Ost, S. Tavare & L. J. Korn, **New approaches for computer analysis of nucleic acid sequences**, *Proc. Natl. Acad. Sci. USA 80* (1983) 5660–5664.
-  T. Kasai, G. Lee, H. Arimura, S. Arikawa & K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181–192.
-  Pang Ko & Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc. 14th Annual Symp.*

Combinatorial Pattern Matching, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 200–210.

-  Jesper Larsson & Alistair Moffat, **Off-line dictionary-based compression**, *Proc. IEEE 88–11* (2000) 1722–1732.
-  Michael Maniscalco & Simon J. Puglisi, **Faster lightweight suffix array construction**, *Proc. 17th Australasian Workshop on Combinatorial Algs.*, J. Ryan & Dafik (eds.) (2006) 16–29.
-  Giovanni Manzini, **Two space-saving tricks for linear time LCP computation**, *Proc. 9th Scandinavian Workshop on Alg. Theory*, LNCS 3111, T. Hagerup & J. Katajainen (eds.), Springer-Verlag (2004) 372–383.

-  Giovanni Manzini & Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica* 40 (2004) 33–50.
-  Kazuyuki Narisawa, Shunsuke Inenaga, Hideo Bannai & Masayuki Takeda, **Efficient computation of substring equivalence classes with suffix arrays**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching* (2007) 340–351.
-  Simon J. Puglisi, W. F. Smyth & Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys* 39–2 (2007) 1–31.
-  Sung W. Shin & Sam M. Kim, **A new algorithm for detecting low-complexity regions in protein sequences**, *Bioinformatics* 21-2 (2005) 160–170.

 Andrew Turpin & W. F. Smyth, **An approach to phrase selection for offline data compression**, *Proc. 25th Australasian Computer Science Conference*, Michael Oudshoorn (ed.) (2000) 267–273.