Fast Optimal Algorithms for Computing All the Repeats in a String

Simon J. Puglisi\textsuperscript{1}, W. F. Smyth\textsuperscript{2,3} and Munina Yusufu\textsuperscript{2}

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

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

\textsuperscript{3}Digital Ecosystems & Business Intelligence Institute
Curtin University, Perth, Western Australia

The Prague Stringology Conference 2008
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
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
A repeating substring $u$ in a string $x$ is a substring of $x$ that occurs more than once.

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

A repeat is left-extendible (LE) if

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

right-extendible (RE) if

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

nonextendible (NE) if neither LE nor RE (both NLE and NRE).
A repeat is **supernonextendible** (SNE) if it is NE and its repeating substring $u$ is not a substring of any other repeating substring of $x$.

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

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

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 $x$; both of them output triples $(p; i, j)$, where $i..j$ is a range of positions in SA.
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].
Given a string $x = x[1..n]$ on an alphabet of size $\alpha$, 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 $\alpha \in \Theta(n)$).
Two recent algorithms [FST03, NIBT07] use suffix arrays to compute all complete NE repeats in $x$ in $O(n)$ time and space independent of $\alpha$. 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 $r\alpha < n$. 
1 Introduction
   - Definitions
   - Applications
   - Previous Algorithms

2 The Algorithms
   - Preprocessing
   - PSY1
   - PSY2
   - Time & Space

3 Experiments

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

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{x} & = & a & b & a & a & b & a & b & a & \$
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{SA} & = & 8 & 3 & 6 & 1 & 4 & 7 & 2 & 5 \\
\text{LCP} & = & -1 & 1 & 1 & 3 & 3 & 0 & 2 & 2 & -1 \\
\text{BWT} & = & b & b & b & \$ & a & a & a & a & a
\end{array}
\]

SA and LCP are arrays of integers ($4n$ bytes), BWT of letters ($n$ bytes).
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 (13$n$ bytes for [KLAAP01], 9$n$ bytes for the slightly slower Manzini variant) is the problem.

Instead of BWT, PSY2 requires an array $\text{LAST} = \text{LAST}[1..n]$ 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.
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).
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 $\text{LCP}[j + 1] > \text{LCP}[j]$. The repeat $(p; i, j)$ is NRE. It is SNLE if and only if for every $h \in i..j$ every left extension $\text{BWT}[h]$ is distinct. This condition can hold only if $j - i + 1 \leq \alpha$.

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

$\text{LAST}$ can be computed in $\Theta(n)$ time by a simple left-to-right scan of $\text{SA}$.
Using LAST, the pseudocode for PSY2 is straightforward:

---

**Preprocessing:** compute \( SA, \) \( \text{LAST} \) \& \( \text{LCP} \). \( j \leftarrow 0; \) \( p \leftarrow -1; \) \( q \leftarrow 0 \)

while \( j < n \) do

\( high \leftarrow 0 \)

repeat

\( j \leftarrow j+1; \) \( p \leftarrow q; \) \( q \leftarrow \text{LCP}[j+1] \)

if \( q > p \) then \( high \leftarrow q; \) \( i \leftarrow j \)

until \( p > q \)

if \( high > 0 \) and \( \text{SNLE}(i, j, \text{LAST}) \) then

output \((p; i, j)\)

---

**Function:** \( \text{SNLE}(\text{start}, \text{end}, \text{LAST}) \)

\( k \leftarrow \text{end} - \text{start} + 1 \)

if \( k > \alpha \) then return \( \text{FALSE} \)

else

for \( h \leftarrow \text{start} + 1 \) to \( \text{end} \) do

if \( h - \text{LAST}[h] > \text{start} \) then return \( \text{FALSE} \)

return \( \text{TRUE} \)

---

**Figure:** Algorithm PSY2 with a simplified SNLE function using LAST
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 $x$ 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).
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, $r\alpha < n$. 
Introduction
- Definitions
- Applications
- Previous Algorithms

The Algorithms
- Preprocessing
- PSY1
- PSY2
- Time & Space

Experiments

Conclusions
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 \texttt{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.
### Table: Files used for testing.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Name</th>
<th>No. Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>highly periodic</td>
<td>fibo35</td>
<td>9,227,465</td>
<td>Fibonacci</td>
</tr>
<tr>
<td></td>
<td>fibo36</td>
<td>14,930,352</td>
<td>Fibonacci</td>
</tr>
<tr>
<td></td>
<td>fss9</td>
<td>2,851,443</td>
<td>run-rich [FSS03]</td>
</tr>
<tr>
<td></td>
<td>fss10</td>
<td>12,078,908</td>
<td>run-rich [FSS03]</td>
</tr>
<tr>
<td>random</td>
<td>rand2</td>
<td>8,388,608</td>
<td>$\alpha = 2$</td>
</tr>
<tr>
<td></td>
<td>rand21</td>
<td>8,388,608</td>
<td>$\alpha = 21$</td>
</tr>
<tr>
<td>DNA</td>
<td>ecoli</td>
<td>4,638,690</td>
<td><em>Escherichia coli</em> genome</td>
</tr>
<tr>
<td></td>
<td>chr22</td>
<td>34,553,758</td>
<td>human chromosome 22</td>
</tr>
<tr>
<td></td>
<td>chr19</td>
<td>63,811,651</td>
<td>human chromosome 19</td>
</tr>
<tr>
<td>Genbank protein database</td>
<td>prot-a</td>
<td>16,777,216</td>
<td>sample</td>
</tr>
<tr>
<td></td>
<td>prot-b</td>
<td>33,554,432</td>
<td>sample</td>
</tr>
<tr>
<td>English</td>
<td>bible</td>
<td>4,047,392</td>
<td>King James version</td>
</tr>
<tr>
<td></td>
<td>howto</td>
<td>39,422,105</td>
<td>Linux howto files</td>
</tr>
<tr>
<td></td>
<td>mozilla</td>
<td>51,220,480</td>
<td>Mozilla source code</td>
</tr>
</tbody>
</table>
### Table: Microseconds per letter used by each run.

<table>
<thead>
<tr>
<th>File</th>
<th>SA</th>
<th>LCP</th>
<th>BWT</th>
<th>LAST</th>
<th>PSY1</th>
<th>[NIBT07]</th>
<th>PSY2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibo35</td>
<td>0.898</td>
<td>0.169</td>
<td>0.025</td>
<td>0.031</td>
<td>0.012</td>
<td>0.448</td>
<td>0.009</td>
</tr>
<tr>
<td>fibo36</td>
<td>0.886</td>
<td>0.170</td>
<td>0.027</td>
<td>0.033</td>
<td>0.012</td>
<td>0.475</td>
<td>0.007</td>
</tr>
<tr>
<td>fss9</td>
<td>0.826</td>
<td>0.154</td>
<td>0.026</td>
<td>0.031</td>
<td>0.014</td>
<td>0.330</td>
<td>0.007</td>
</tr>
<tr>
<td>fss10</td>
<td>0.958</td>
<td>0.177</td>
<td>0.025</td>
<td>0.032</td>
<td>0.013</td>
<td>0.469</td>
<td>0.008</td>
</tr>
<tr>
<td>periodic AVG</td>
<td>0.892</td>
<td>0.168</td>
<td>0.026</td>
<td>0.032</td>
<td>0.013</td>
<td>0.430</td>
<td>0.008</td>
</tr>
<tr>
<td>rand2</td>
<td>0.947</td>
<td>0.188</td>
<td>0.026</td>
<td>0.031</td>
<td>0.017</td>
<td>0.215</td>
<td>0.012</td>
</tr>
<tr>
<td>rand21</td>
<td>1.135</td>
<td>0.199</td>
<td>0.025</td>
<td>0.031</td>
<td>0.012</td>
<td>0.122</td>
<td>0.012</td>
</tr>
<tr>
<td>random AVG</td>
<td>1.041</td>
<td>0.193</td>
<td>0.025</td>
<td>0.031</td>
<td>0.015</td>
<td>0.169</td>
<td>0.012</td>
</tr>
<tr>
<td>ecoli</td>
<td>1.413</td>
<td>0.175</td>
<td>0.025</td>
<td>0.031</td>
<td>0.015</td>
<td>0.155</td>
<td>0.011</td>
</tr>
<tr>
<td>chr22</td>
<td>1.635</td>
<td>0.285</td>
<td>0.035</td>
<td>0.040</td>
<td>0.016</td>
<td>0.278</td>
<td>0.012</td>
</tr>
<tr>
<td>chr19</td>
<td>1.873</td>
<td>0.333</td>
<td>0.044</td>
<td>0.053</td>
<td>0.016</td>
<td>0.242</td>
<td>0.012</td>
</tr>
<tr>
<td>DNA AVG</td>
<td>1.754</td>
<td>0.309</td>
<td>0.035</td>
<td>0.041</td>
<td>0.016</td>
<td>0.225</td>
<td>0.012</td>
</tr>
<tr>
<td>prot-a</td>
<td>1.778</td>
<td>0.222</td>
<td>0.027</td>
<td>0.032</td>
<td>0.013</td>
<td>0.211</td>
<td>0.012</td>
</tr>
<tr>
<td>prot-b</td>
<td>1.971</td>
<td>0.277</td>
<td>0.034</td>
<td>0.039</td>
<td>0.013</td>
<td>0.247</td>
<td>0.012</td>
</tr>
<tr>
<td>protein AVG</td>
<td>1.874</td>
<td>0.249</td>
<td>0.030</td>
<td>0.036</td>
<td>0.013</td>
<td>0.229</td>
<td>0.012</td>
</tr>
<tr>
<td>bible</td>
<td>1.417</td>
<td>0.151</td>
<td>0.024</td>
<td>0.030</td>
<td>0.015</td>
<td>0.168</td>
<td>0.012</td>
</tr>
<tr>
<td>howto</td>
<td>1.912</td>
<td>0.214</td>
<td>0.035</td>
<td>0.039</td>
<td>0.016</td>
<td>0.219</td>
<td>0.012</td>
</tr>
<tr>
<td>mozilla</td>
<td>1.815</td>
<td>0.187</td>
<td>0.032</td>
<td>0.036</td>
<td>0.013</td>
<td>0.139</td>
<td>0.011</td>
</tr>
<tr>
<td>English AVG</td>
<td>1.417</td>
<td>0.151</td>
<td>0.024</td>
<td>0.035</td>
<td>0.014</td>
<td>0.175</td>
<td>0.012</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>1.390</td>
<td>0.207</td>
<td>0.029</td>
<td>0.035</td>
<td>0.014</td>
<td>0.266</td>
<td>0.011</td>
</tr>
</tbody>
</table>
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.


