On Maximal Suffixes and Constant-Space Linear-Time Versions of KMP Algorithm

Wojciech Rytter *

Abstract

Constant-space linear-time string-matching algorithms are usually very sophisticated. Most of them consist of two phases: (very technical) *preprocessing phase* and *searching phase*. An exception is *one-phase* Crochemore's algorithm [1]. It is an on-line version of Knuth-Morris-Pratt algorithm (KMP) with "*on-the-fly*" computation of pattern shifts (as *approximate* periods). In this paper we explore further Crochemore's approach, and construct alternative algorithms which are differently structured. In Crochemore's algorithm the approximate-period function is restarted *from inside*, which means that several internal variables of this function are changing globally, also Crochemore's algorithm strongly depends on the concrete implementation of approximate-periods computation. We present a simple modification of KMP algorithm which works in $O(1)$-space, $O(n)$-time for **any** function which computes periods or approximate periods in $O(1)$-space, and linear time. The approximate-period function can be treated as a *black box*. We identify class of patterns, *self-maximal* words, which are especially well suited for Crochemore-style string matching. A new $O(1)$ space string-matching algorithm, *MaxSuffix-Matching*, is proposed in the paper, which gives yet another example of applicability of maximal suffixes.

Keywords: String-matching; Linear time; Constant space; Maximal suffix

1 Introduction

The two very classical algorithms in "*stringology*" are Knuth-Morris-Pratt (KMP) algorithms and algorithm computing lexicographically maximal suffix *MaxSuf*(w) of the word w. In [1] Crochemore has shown how to *marry* KMP with *MaxSuf* to achieve *single-phase* constant-space linear-time string-matching on-line algorithm. "*On-line*" means here that only the symbols of the longest prefix of P occurring in T are read. We pursue Crochemore's approach. Assume that the pattern and the text are given as *read-only* tables P and T, where $|P| \le |T| = n$.

*Instytut Informatyki, Uniwersytet Warszawski, Poland and Department of Computer Science, Liverpool University, Chadwick Building, Peach Street, Liverpool L69 7ZF, U.K. Email:rytter@mimuw.edu.pl

We count space as the number of additional integer registers (each in the range $[0 \dots n]$) used in the algorithm. The *string-matching problem* consists in finding all occurrences of P in T. The algorithms solving this problem with linear cost and (simultaneously) constant space are the most interesting and usually the most sophisticated. The first constant-space linear time algorithm was given by Galil and Seiferas in [5]. Later Crochemore and Perrin in [2] have shown how to achieve $2n$ comparisons algorithm preserving small amount of memory. Alternative algorithms were presented in [6] and [7]. The KMP algorithm is $O(1)$-space algorithm except the table for shifts (or, equivalently, periods). The natural approach to constant-space algorithms is to get rid of these tables. We show how it can be done in different ways by employing Crochemore's approach [1]:

linear-space table $\Rightarrow$ *function computable in* $O(1)$*-space and linear time.*

The above approach lacks details and is not a full receipt, some unsophisticated *algorithmic engineering* related to technicalities of KMP algorithm is needed. In this paper we do not pursue the issue of the exact number of symbol comparisons, which is usually not a dominating part in the whole complexity. We will be satisfied with linear time, as long as the algorithm is reasonably simple. From a teaching point of view it is desirable to have a linear time $O(1)$ space algorithm which is easily understandable.

2 KMP algorithm with exact and approximate shifts

KMP algorithm aligns P with T starting at some position i and finds the longest partial match: $P[1 \dots j] = T[i+1 \dots i+j]$. Then KMP makes a *shift*, determined by the size j of the partial match. The shifts are precomputed and stored in an array, and they are exactly the periods of $P[1 \dots j]$ for $j \geq 1$. Consequently we use periods in place of shifts. If $j = 0$ then the shift is forced to be 1. Let $period(x)$ be the size of the shortest period of a word x.
Denote $Period(j) = period(P[1 \dots j])$ for $j \geq 1$ and $Period(0) = 0$.

algorithm $KMP1$;
 $i := 0; j := 0$;
 while $i \leq n - m$ **do**
  begin
   while $j < m$ and $P[j+1] = T[i+j+1]$ **do** $j = j+1$;
   {**MATCH**:} **if** $j = m$ **then report** match at i;
    $i := i + \max\{1,\ Period(j)\}$; $j := j - Period(j)$;
  end ;

It is much easier to compute in $O(1)$ space an *approximate period*. Define:

$$ApprPeriod(j) \;=\; \begin{cases} Period(j), & \text{if } Period(j) \le \frac{j}{2} \\ nil, & \text{if } Period(j) > \frac{j}{2} \end{cases}$$

We say that a text x is *periodic* iff $period(x) \le |x|/2$. Hence the function *ApprPeriod* gives the value of periods for periodic texts only. The function *ApprPeriod* is computed in very simple *MaxSuf-and-Period* algorithm presented in the last section for completeness. We substitute the function of exact period by *ApprPeriod* and modify the KMP1 algorithm as follows.

ALGORITHM *KMP2*;
i:=0; j:=0;
while $i \le n - m$ **do**
begin
while $j < m$ and $P[j+1] = T[i+j+1]$ **do** $j = j + 1$;
MATCH: **if** $j = m$ **then report** match at i;
$period := ApprPeriod(j)$;
if $period = nil$ **then** **begin** $i := i + \lceil \frac{j+1}{2} \rceil$; $j := 0$ **end**
else {**Periodic-Case**:}
begin $i := i + \max\{1,\ period\}$; $j = j - period$ **end**
end;

The following fact is well known and used in many constant space string-matching algorithms.

Theorem 1 *Algorithms KMP1 and KMP2 work in linear time if the values of ApprPeriod and Period are available in constant time.*

3 $O(1)$-space version of KMP without preprocessing for very special patterns: *SpecialCase-KMP*

There is one *very special* class of patterns for which exact period $O(1)$-space computation is trivial: **self-maximal patterns**. These are the patterns P such that $MaxSuf(P) = P$. *Maximal suffixes* play important role in the computation of periods for three reasons:

(1) if P is periodic then $period(MaxSuf(P)) \;=\; period(P)$,

(2) if P is *self-maximal* then each of its prefixes is.

(3) if P is *self-maximal* then $period(P)$ can be trivially computed by the following function:

```
function Naive-Period(j);
 period := 1;
  for i := 2 to j do
   if P[i] ≠ P[i − period]  then period := i;
 return (period);
```

Example. The function *Naive-Period* usually gives incorrect output for non-self-maximal words, for example consider the string: $P = (aba)^6ab = abaabaabaabaabaabaa$.

The consecutive values of *period* computed by the function for consecutive positions are:

a	b	a	a	b	a	a	b	a	a	b	a	a	b	a	a	b	a	a
1	2	2	4	5	5	7	8	8	10	11	11	13	14	14	16	17	17	19

Hence *Naive-Period*(19)) = 19, for $P = (aba)^6a$, while $Period(19) = 3$.

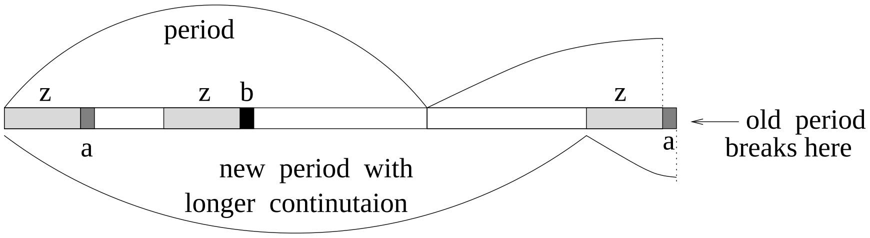

Figure 1: Assume in the algorithm *Naive-Period* $P[j - Period(j-1)] \neq P[j]$. Then $a = P[j]$, $b = P[j - period]$, and $a < b$, where $period = Period(j-1)$. If $Period(j) < j$ then, due to two periods, zb is a subword of $P[1 \ldots j]$ and za is a prefix of P. Then zb is a proper subword of $P[1 \ldots j-1]$ which is lexicographically greater than P. This contradicts self-optimality of P. Hence $Period(j) = j$.

The proof of the following lemma can be easily extracted from the correctness of maximal suffix algorithm, see [4]. A very informal justification is given in Figure 1.

Lemma 1 *Assume P is a self-maximal string. If $P[j - P(j-1)] \neq P[j]$ then $Period(j) = j$. The function Naive-Period computes correctly the exact period of P.*

We can embed the computation of *Naive-Period(j)* directly into KMP algorithm using Crochemore's approach. $Period(j) = period$ is computed here "*on-the-fly*" in especially simple way.

```
ALGORITHM SpecialCase-KMP;
 i := 0 ; j := 0;  period := 1;
 while i ≤ n − m do
   begin
     while j < m and P[j + 1] = T[i + j + 1] do
         begin
         j = j + 1;  if j > period and P[j] ≠ P[j − period]
         then period := j end;
     {MATCH:} if j = m then report match at i;
      i := i + period;
       if j ≥ 2 · period then j := j − period;
      else  begin j := 0;  period := 1 end;
   end ;
```

4 A simple two-phase version of KMP: *MaxSuffix-Matching*

Assume we know the decomposition $P = u \cdot v$, where $v = MaxSuf(P)$. Then the algorithm SpecialCase-KMP can be used to report each occurrence of v. We convert this algorithm into an $O(1)$-space algorithm for general patterns by testing in a *naive* way *some* occurrences of u to the left of v. If v starts at i then uv can start at $i - |u|$. We do not need to do it for each occurrence of v due to the following fact. Assume that 0 is a *special* default occurrence of v.

Lemma 2 [Key-Lemma] *Assume i is an actual occurrence of v in T and the previous occurrence is $prev$. If $i - prev < |u|$ then there is no occurrence of uv at position $i - |u|$.*

Proof: The maximal suffix v of P can start only in one place in P. □

```
ALGORITHM Informal-MaxSuffix-Matching ;
 Let P = uv, where v = MaxSuf(P);
  Search for v with algorithm SpecialCase-KMP;
  for each occurrence i of v in T do
      Let prev be the previous occurrence of v
      if  i − prev > |u| then
        if u occurs to the left of v then report match;
        { occurrence of u is tested in a naive way}
```

Example. Consider the decomposition of the 7-th Fibonacci word and history of the algorithm *MaxSuffix-matching* on an example text, see Figure 2.

$$P \;=\; Fib_7 \;=\; abaababaabaababaababa \;=\; \overbrace{abaababaabaa}^{u} \mid \overbrace{babaababa}^{v}$$

$$T \;=\; abaabaa\,\overbrace{babaababa}^{v}\,baa\,\overbrace{b\underline{abaababa}}^{v}\,\underline{abaa}\,\overbrace{babaababa}^{v}\,ba$$

Figure 2: The decomposition of the 7-th Fibonacci word and history of *MaxSuffix-Matching* on an example text T. Only for the third occurrence of v we check if there is an occurrence of u (underlined segment) to the left of this occurrence of v, because gaps between previous occurrences of v are too small. The first occurrence is too close to the beginning of T.

We embed the tests for u into the algorithm *SpecialCase-KMP* and obtain less informal description of the algorithm. The tests $u \;=\; T[i-|u|+1\ldots i]$ are being done in a *naive* way. If $v = MaxSuf(P)$ is known then it is a particularly simple $O(1)$-space matching algorithm.

```
ALGORITHM MaxSuffix-Matching;
 i := 0 ; j := 0; period := 1; prev := 0;
 while i ≤ n − |v| do
   begin
    while j < |v| and v[j + 1] = T[i + j + 1] do
       begin
       j = j + 1;  if j > period and v[j] ≠ v[j − period]
       then period := j end;
   {MATCH OF v:} if j = |v| then  begin
         if i − prev > |u| and u  =  T[i − |u| + 1 . . . i]
         then report match at i − |u|;  prev := i;
         end
       i := i + period;  if j ≥ 2 · period then j := j − period
         else  begin j := 0;  period := 1 end;
     end ;
```

Theorem 2 *The algorithm MaxSuffix-Matching works in linear time and constant space. It makes at most n extra symbol comparisons with respect to the total costs of searching for v.*

Proof: Additional *naive* tests for u are related to disjoint segments of T, due to Lemma 2, and altogether take at most n extra symbol comparisons. □

5 Simple $O(1)$-space versions of KMP without preprocessing

Let us treat $Period$ and $ApprPeriod$ as functions computable in linear time with $O(1)$ space. This converts algorithms KMP1 and KMP2 into constant-space algorithms. Are the algorithms KMP1 and KMP2 still linear time algorithms ? The answer is "NO". The counterexamples are given, for example, by highly periodic texts: $P = a^n$, $T = a^{n-1}ba^n$.

To deal with periodic cases we add *memory* to remember the last value of *period* and the range of its applicability. We use the following obvious fact:

Same-Period-Property:

$$Period(R) = period \;\Rightarrow\; \forall\ (i \in [2 \cdot period \ldots R])\ Period(i) = period$$

The value of j is *remembered* in the additional register R. Whenever we need later $Period(j)$ for a new value of j which is still in the range $[2 \cdot period \ldots R]$ we do not need to recompute it due to *Same-Period-Property.* In KMP1 we simply insert:

if $j \notin [2 \cdot period \ldots R]$ **then**
  begin $period := Period(j)$; $R := j$ **end**;

In KMP2 we have to change $Period$ to $ApprPeriod$ and take care additionally of situation $period = nil$. Observe that our modifications of KMP1 and KMP2 look rather cosmetic. The only new variables are *period* and R and the tables of periods are treated as functions. This is rather naive approach, though effective, and to reduce constant coefficients the algorithms should be tuned up further, resulting possibly in great loss of simplicity. Surprisingly such a *rough* approach guarantees linear time, assuming that computation of periods each time takes linear time. The modified algorithms KMP1 and KMP2 are named *ConstantSpace-KMP1* and *Constant Space-KMP2.* For technical reasons assume $Period(0) = ApprPeriod(0) = 0$.

We obtain now two very simple algorithms whose description and correctness are trivial but analysis is not.

We analyse both algorithms together. The complexity of the algorithms differs from KMP1 and KMP2 by the cost of computing the periods, and this costs can be *charged* to the lengths j of *partial matches* for which the period functions are called. A *partial match* of P is an alignment of P with T over a segment $[i+1 \ldots i+j]$ in T such that: $P[1 \ldots j] = T[i+1 \ldots i+j]$, and $(P[j+1] \neq T[i+j+1]$ or $j = m)$. We identify in this proof partial match with the segment of T equal to $[i+1 \ldots i+j]$ as well as with the text $T[i+1 \ldots i+j]$.

Define partial match as an **expensive match** if for this partial match we compute one of the functions *Period* or *ApprPeriod.*

```
ALGORITHM Constant-Space-KMP1;
 i:=0; j:=0; period := n; R := 0;
 while i ≤ n − m do
  begin
   while j < m and P[j + 1] = T[i + j + 1] do   j = j + 1;
    {MATCH:} if j = m then report  match at i;
    if j ∉ [2 · period . . . R] then
          begin period := Period(j); R := j end;
    i := i + max{1,period}; j = j − period
  end end;
```

```
ALGORITHM Constant-Space-KMP2;
 i:=0; j:=0; period := n; R := 0;
 while i ≤ n − m do
  begin
   while j < m and P[j + 1] = T[i + j + 1] do  j = j + 1;
    MATCH: if j = m then  report  match at i;
    if period = nil or j ∉ [2 · period . . . R] then
          begin period := ApprPeriod(j); R := j end;
    if period > nil then  begin i := i + ⌈(j+1)/2⌉; j := 0 end
    else  begin i := i + max{1,period}; j = j − period end
  end;
```

We say that position $j+1$ breaks periodicity if $\mathit{Period}(j) < \mathit{Period}(j+1)$, in other words $P[j+1-\mathit{Period}(j)] \neq P[j+1]$.

Lemma 3

Assume position $j+1$ breaks periodicity. Then for any $r > j$ $\mathit{Period}(r) \geq j - \mathit{Period}(j)$.

Proof: (By contradiction) Assume $\mathit{Period}(r) \leq j - \mathit{Period}(j)$, and $\mathit{period} = \mathit{Period}(j)$, $\mathit{period}' = \mathit{Period}(r)$. Then $j+1-\mathit{period}-\mathit{period}' \geq j+1-\mathit{period}-(j-\mathit{period}) \geq 1$, so $P[j+1-\mathit{period}'] = P[j+1]$ and $P[j+1-\mathit{period}-\mathit{period}'] = P[j+1-\mathit{period}]$. We have that $j+1-\mathit{period}' < j$, so the period of j applies and $P[j+1-\mathit{period}'] = P[j+1-\mathit{period}']$, see Figure 3. It would imply $P[j+1] = P[j+1-\mathit{period}]$, a contradition. $\square$

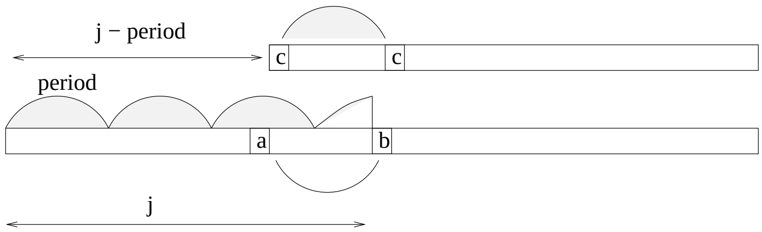

Figure 3: If $j+1$ breaks periodicity ($a \neq b$) then next period should be large.

If an *expensive match* x starts at i and next one starts at j then we say that $j-i$ is the shift of x and write $EShift(x) = j - i$. If y is the expensive match following x then write $NextEShift(x) = EShift(y)$. If x is the last match then $EShift(x) = NextEShift(x) = |x|$.

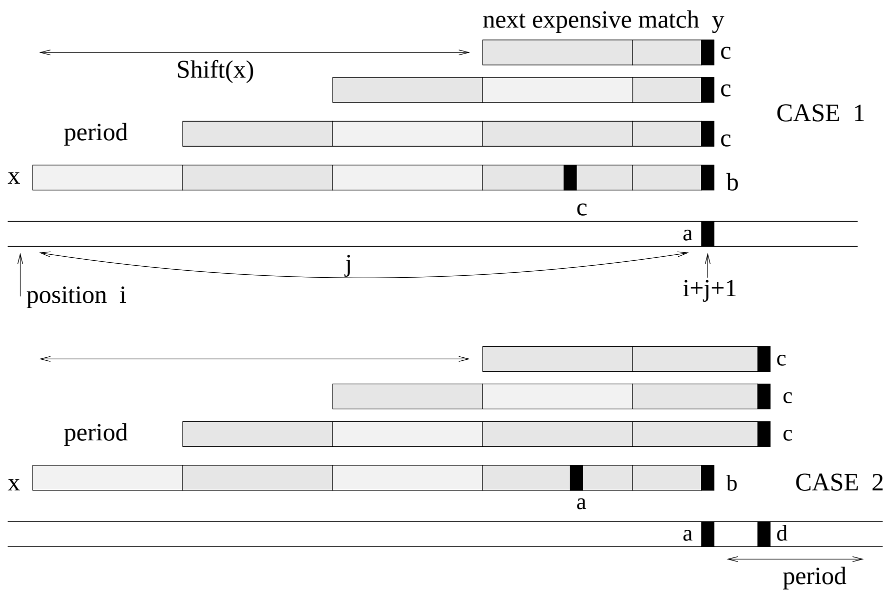

Figure 4: Cases 1 and 2.

Lemma 4 *If x is an expensive match then $EShift(x) < |x|/3 \Rightarrow NextEShift(x) \geq |x|/3$.*

Proof: We can assume that $Period(j) = period \leq |x|/3$, otherwise the first shift would be at least $|x|/3$. Consider first three cases when $|x| = j < m$. x corresponds in T to interval $[i, i+j+1]$, see Figure 4. We have: $P[1,j] = T[i, i+j]$, $P[j+1] \neq T[i+j+1]$.

Case 1 (Figure 4) $|x| < m$, $T[i+j+1] \neq P[j+1-period]$. The pattern is shifted several times by distance *period* until the partial match shrinks to the size less than $2 \cdot period$. All partial matches until this moment are not expensive since the same period occured at least twice. Since $period \leq |x|/3$ we have $EShift(x) \geq |x| - 2 \cdot period \geq |x|/3$.

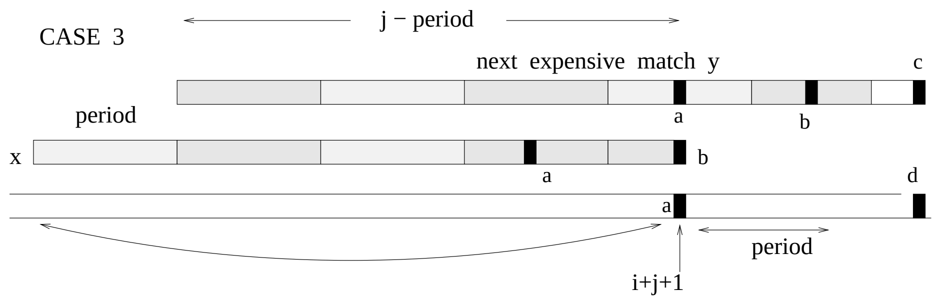

Figure 5: Case 3.

Case 2 (Figure 4) $|x| < m$, $T[i+j+1] = P[j+1-period]$, and $T[i+j+r] \neq P[i+j+r]$ for $r \leq period$. Next partial match (which is not necessarily an expensive one) is at most $|x|$.

After the first shift the situation is the same as in Case 1. All partial matches are non-expensive, till the moment when the match shrinks and the shift becomes large enough.

Case 3 (Figure 5) $|x| < m$, $T[i+j+1] = P[j+1-period]$, $|y| \geq j+1$, where y is next partial match. $|y| > |x|$ and y is necessarily expensive since its length exceeds R.

Now position $j+1$ breaks periodicity and Lemma 3 applies. The new period is at least $j - period$, and possibly approximate period does not give correct value. In this case $NextEShift(x) \geq (j - period)/2 \geq |x|/3$.

Case 4 $|x| = m$. The algorithm makes several shifts of the whole pattern over itself (without necessity to recompute the period) until a mismatch is hit. Several shifts without recomputing the period are done until the current partial match shrinks to the word no longer than $2 \cdot Period(m)$. The same argument as in Case 1 applies. □

Theorem 3 *Assume $Period(j)$ and $ApprPeriod(j)$ can be computed in $O(j)$ time with $O(1)$ space. Then the algorithms* Constant-Space-KMP1 *and* Constant-Space-KMP2 *report each occurrence of the pattern in linear time with constant space. Both algorithms are on-line: they only read the symbols of the longest prefix of P occurring in the text.*

Proof: We only need to prove that the total cost of all calls to *Period* or *ApprPeriod* is linear, due to Theorem 1. Due to Lemma 4 expensive matches are amortized by their shifts or the next shits. Hence the cost is $O(n)$ since total size of all shifts is linear. □

Corollary 1 *Both algorithms can compute the exact period of the pattern within the same complexity, by searching P in $T = P$ and starting from the second position.*

Corollary 2 *The algorithm Constant-Space-KMP1 with function $Period$ replaced by Naive-Period solves string-matching problem for self-maximal patterns in $O(1)$-space and linear time.*

6 Maximal suffixes and approximate periods

For completeness we include small space computation of approximate periods and maximal suffixes. Our presentation differs from previously known ones, see [8] and [4] , its main component is our (very simple) function *Naive-Period*. We convert first the function *Naive-Period* into a computation of the length of the longest self-maximal prefix of a given text x together with its shortest period.

```
function Longest-SelfMax-Prefix(x);
  period :=1;
   for i := 2 to |x| do
      if  x[i] < x[i − period]  then  period := i
      else if  x[i] > x[i − period]  then
            return (i − 1, period)
  return (|x|, period) {there was no return earlier};
```

We use the computation of *Longest-SelfMax-Prefix* as a key component in the maximal suffix computation for the whole text.

```
function MaxSuf-and-Period(P); {|P| = m}
 j := 1;
  repeat
     (k, period) := Longest-SelfMax-Prefix( P[j ... m] );
     if k = m − j + 1 then return  (P[j ... m], period)
     else  j := j + k − (k  mod period)
```

Technically the last function returns only the starting position of the maximal suffix, to economize space. We use the function *MaxSuf-and-Period* to compute approximate periods.

```
function ApprPeriod(x);
  (v, period) := MaxSuf-and-Period(x);
  if |v| ≥ |x|/2 and period is a period of x
  { test naively if positions in [1 ... |x| − |v|] obey periodicity}
 then return  period else return  nil;
```

The algorithms in this section are a different presentation of well known algorithms for maximal suffixes and the proof of the following result can be found for example in [4].

Lemma 5 *MaxSuffix-and-Period algorithm runs in $O(|x|)$ time with $O(1)$ space. It makes less than 2 $|x|$ letter comparisons. If x is periodic then the exact period of x is returned.*

References

[1] M. Crochemore, String-matching on ordered alphabets. *Theoret. Comput. Sci.*, 92, p. 33–47, 1992.

[2] M. Crochemore and D. Perrin, Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3), p. 651–675, 1991.

[3] M. Crochemore and W. Rytter, Cubes, squares and time space efficient string matching, Algorithmica 13,5 (1995) 405-425

[4] M. Crochemore and W. Rytter, Text algorithms, *Oxford University Press*, New York, 1994

[5] Z. Galil and J. Seiferas, Time-space-optimal string matching. *J. Comput. System Sci.*, 26, p. 280–294, 1983.

[6] L. Gąsieniec, W. Plandowski and W. Rytter, The zooming method: a recursive approach to time-space efficient string-matching. *Theoretical Computer Science* 1995

[7] L. Gąsieniec, W. Plandowski and W. Rytter, String matching in small time and space, Combinatorial Pattern Matching 1995, Lecture Notes in Computer Science 1995

[8] J-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983): 363-381.

[9] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings. *SIAM J. Comput.*, 6, p. 322–350, 1977.

[10] M. Lothaire, Combinatorics on Words. Addison-Wesley, Reading, MA., U.S.A., 1983.