# McMaster University

## Advanced Optimization Laboratory

**Title:**

Crochemore repetition algorithm revisited – computing runs

**Authors:**

Frantisek Franek and Mei Jiang

**AdvOL-Report No. 2009/1**

April 2009, Hamilton, Ontario, Canada

# Crochemore repetition algorithm revisited – computing runs

Frantisek Franek and Mei Jiang

April 22, 2009

**Abstract**

Crochemore repetition algorithm introduced in 1981 was the first $O(n \log n)$ algorithm for computing repetitions. Since then, several linear-time worst-case algorithms for computing runs have been introduced. They all follow a similar strategy – first compute the suffix tree or array, then use the suffix tree or array to compute the Lempel-Ziv factorization, then using the Lempel-Ziv factorization compute all the runs.

It is conceivable that in practice an extension of Crochemore repetition algorithm may outperform the linear-time algorithms, or at least for certain classes of strings. The nature of Crochemore algorithm lends itself naturally to parallelization, while the linear-time algorithms are not easily conducive to parallelization.

For all these reasons it is interesting to explore ways to extend the original Crochemore repetition algorithm to compute runs. We present three variants of extending the repetition algorithm to compute runs – two with a worsen complexity of $O(n \log^2 n)$, and one with the same complexity as the original algorithm. The three variants are tested for speed of performance and their memory requirements are analyzed.

**Keywords:** repetition, run, string, periodicity, suffix tree, suffix array

## 1 Introduction

An important structural characteristic of a string over an alphabet is its periodicity. Repetitions (tandem repeats) have always been in the focus of the research into periodicities. The notion of runs that captures maximal repetitions which themselves are not repetitions was introduced by Main ([M89]) as a more succinct notion in comparison with repetitions. It was shown by Crochemore in 1981 that there could be $O(n \log n)$ repetitions in a string of length $n$ and an $O(n \log n)$-time worst-case algorithm was presented ([C81]), while Kolpakov and Kucherov proved in 2000 that the number of runs was $O(n)$ ([KK00]).

Since then, several linear-time worst-case algorithm have been introduced, all based on linear algorithm for computing suffix trees or suffix arrays. Main ([M89]) showed how to compute the leftmost occurrences of runs from the Lempel-Ziv factorization in linear time, Weiner ([W73]) showed how to compute Lempel-Ziv factorization from a suffix tree in linear time. Finally, in 1997 Farach ([F97]) demonstrated a linear construction of suffix tree. In 2000, Kolpakov and Kucherov ([KK00]) showed how to compute all the runs from the leftmost occurrences in linear time. Suffix trees are complicated data structures and Farach construction was not practical to implement for sufficiently large $n$, so such a linear algorithm for computing runs was more of a theoretical result than a practical algorithm.

In 1993, Manber and Myers ([MM93]) introduced suffix arrays as a simpler data structure than suffix trees, but with many similar capabilities. Since then, many researchers showed how to use suffix arrays for most of the tasks suffix trees were used without worsening the time complexity. In

2004, Abouelhoda et. al. showed how to compute in linear time the Lempel-Ziv factorization from the extended suffix array. In 2003, several linear time algorithms for computing suffix arrays were introduced (e.g. [KS03, KA03]). This paved the way for practical linear-time algorithms to compute runs. Currently, there are several implementations (e.g. Johannes Fischer's, UnivarsitätTübingen, or Kucherov's, CNRS Lille) and the latest, CPS, is described and analyzed in [CPS07].

Though suffix arrays are much simpler data structures than suffix trees, these linear time algorithms for computing runs are rather involved and complex. In comparison, Crochemore algorithm is simple and mathematically elegant. It is thus natural to compare their performances. The strategy of Crochemore algorithm relies on repeated refinements of classes of equivalence, a process that can be easily parallelized, as each refinement of a class is independent of the other classes and their refinements, and so can be performed simultaneously by different processors. The linear algorithms for computing runs are on the other hand not very conducive to parallelization (the major reason is that all linear suffix array constructions rely on recursion). For these reasons we decided to extend the original Crochemore algorithm based on the most memory efficient implementation by Franek et. al. ([FSX03]).

In this report we discuss and analyze three possible extensions of FSX03 for computing runs and their performance testing: two variants with time-complexity of $O(n \log^2 n)$ and one variant with time-complexity of $O(n \log n)$.

## 2 Basic notions

**Repeat** is a collection of repeating substrings of a given string. **Repetition**, or **tandem repeat**, consists of two or more adjacent identical substrings. It is natural to code repetitions as a triple $(\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{e})$, where $\boldsymbol{s}$ is the ***start*** or ***starting position*** of the repetition, $\boldsymbol{p}$ is its ***period***, i.e. the length of the repeating substring, and $\boldsymbol{e}$ is its ***exponent*** (or ***power***) indicating how many times the repeating substring is repeated. The repeating substring is referred to as the ***generator*** of the repetition. More precisely:

**Definition 2.1.** *$(s, p, e)$ is a repetition in a string $\boldsymbol{x}[0..n{-}1]$ if $\boldsymbol{x}[s..(s{+}p{-}1)] = \boldsymbol{x}[(s{+}p)..(s{+}2p{-}1)] = \cdots = \boldsymbol{x}[(s{+}(e{-}1)p)..(s{+}ep{-}1)]$.*

A repetition $(s, p, e)$ is **maximal** if it cannot be extended to the left nor to the right, i.e. $(s, p, e)$ is a repetition in $\boldsymbol{x}$ and $\boldsymbol{x}[(s{-}p{+}1)..(s{-}1)] \neq \boldsymbol{x}[s..(s{+}p{-}1)]$ and $\boldsymbol{x}[(s{+}(e{-}1)p)..(s{+}ep{-}1)] \neq \boldsymbol{x}[(s{+}ep)..(s{+}(e{+}1)p{-}1)]$.
In order to make the coding of repetitions more space efficient, the repetitions with generators that are themselves repetitions are not listed; for instance, *aaaa* should be reported as (0,1,4) just once, there is no need to report (1,2,2) as it is subsumed in (0,1,4).
Thus we require that generator of a repetition to be **irreducible**, i.e. not a repetition.

Consider a string *abababa*, there are maximal repetitions (0,2,3) and (1,2,3). But in fact, it can be viewed as a fractional repetition $(0, 2, 3 + \frac{1}{2})$. This is an idea of a run coded into a quadruple $(\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{e}, \boldsymbol{t})$, where $\boldsymbol{s}$, $\boldsymbol{p}$, and $\boldsymbol{e}$ are the same as for repetitions, while $\boldsymbol{t}$ is the ***tail*** indicating the length of the last incomplete repeat. For instance, for the above string we can only report one run (0,2,3,1) and it characterizes all the repetitions implicitly. The notion of runs is thus more succinct and more space efficient in comparison with the notion of repetitions. More precisely:

**Definition 2.2.** *$\boldsymbol{x}[s..(s{+}ep{+}t)]$ is a* **run** *in a string $\boldsymbol{x}[0..n{-}1]$ if $\boldsymbol{x}[s..(s{+}p{-}1)] = \boldsymbol{x}[(s{+}p)..(s{+}2p{-}1)] = \cdots = \boldsymbol{x}[(s{+}(e{-}1)p)..(s{+}ep{-}1)]$ and $\boldsymbol{x}[(s{+}(e{-}1)p)..(s{+}(e{-}1)p{+}t)] = \boldsymbol{x}[(s{+}ep)..(s{+}ep{+}t)]$, where $0 \le \boldsymbol{s} < n$ is the* ***start*** *or the* ***starting position*** *of the run, $1 \le \boldsymbol{p} < n$ is the* ***period*** *of the* *run,* $\boldsymbol{e} \geq 2$ *is the* ***exponent*** *(or* ***power****) of the run, and* $0 \leq \boldsymbol{t} < \boldsymbol{p}$ *is the* ***tail*** *of the run. Moreover, it is required that either* $\boldsymbol{s} = 0$ *or that* $\boldsymbol{x}[s-1] \neq \boldsymbol{x}[s+2p-1]$ *(in simple terms it means that it is a leftmost repetition) and that* $\boldsymbol{x}[s+(ep)+t+1] \neq \boldsymbol{x}[s+(e+1)p+t+1]$ *(in simple terms it means that the tail cannot be extended to the right). It is also required, that the* ***generator*** *be irreducible.*

## 3 A brief description of Crochemore algorithm

Let $\boldsymbol{x}[0..n-1]$ be a string. We define an equivalence $\approx_p$ on positions $\{0, \cdots, n-1\}$ by $i \approx_p j$ if $\boldsymbol{x}[i..i+p-1] = \boldsymbol{x}[j..j+p-1]$. In Fig.1, the classes of $\approx_p$, $p = 1..8$ are illustrated. For technical reasons, a sentinel symbol \$ is used to denote the end of the input string; it is consider to be the lexicographically smallest character. If $i, i+p$ are in the same class of $\approx_p$ (as illustrated by 5, 8 in the class $\{0, 3, 5, 8, 11\}$ on level 3, or 0,5 in class $\{0, 5, 8\}$ on level 5, in Fig.1) then there is a tandem repeat of period $p$ (thus $\boldsymbol{x}[5..7] = \boldsymbol{x}[8..10]$ =**aba** and $\boldsymbol{x}[0..4] = \boldsymbol{x}[5..9]$ =**abaab**).

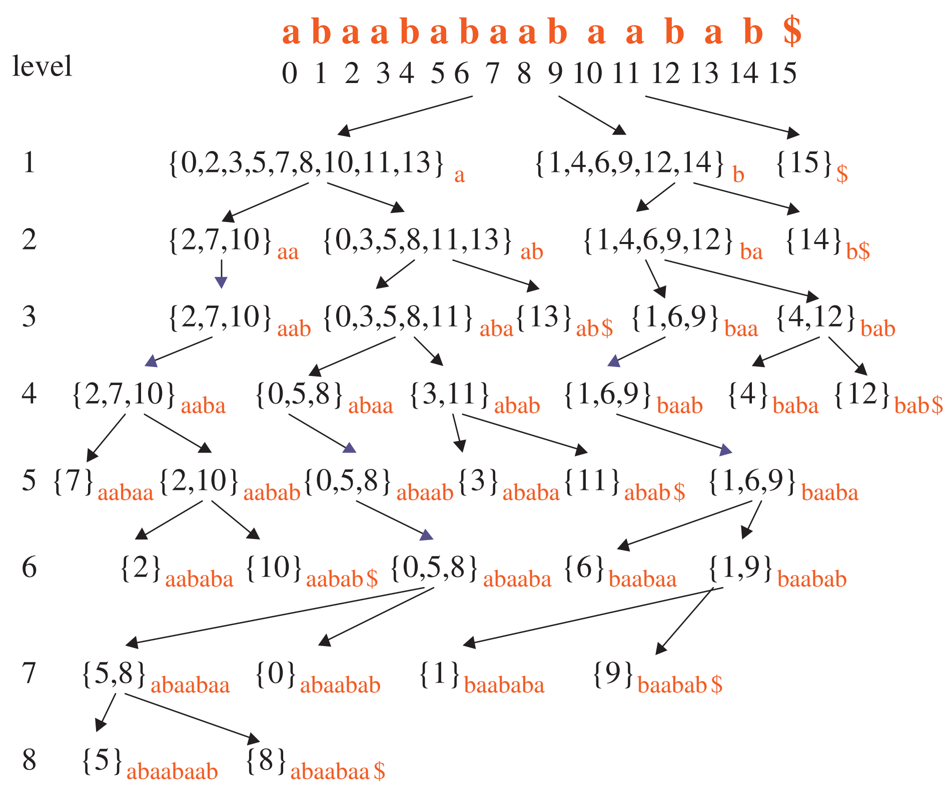


Figure 1: Classes of equivalence and their refinements for a string **abaababaabaabab**

Thus the computation of the classes and identification of repeats of the same "gap" as the level (period) being computed lei in the heart of Crochemore algorithm. A naive approach following the scheme of Fig.1 would lead to an $O(n^2)$ algorithm, as there are potentially $\leq n$ classes on each level and there can be potentially $\leq \frac{n}{2}$ levels.

The first level is computed directly by a simple left-to-right scan of the input string - of course we are assuming that the input alphabet is $\{0, \cdots, n-1\}$, if it is not, in $O(n \log n)$ the alphabet of the input string can be transformed to it.

Each follow-up level is computed from the previous level by refinement of the classes of the previous level (in Fig.1 indicated by arrows). Once a class decreases to a singleton (as $\{15\}$ on level 1, or $\{14\}$ on level 2), it is not refined any further. After a level $p$ is computed, the equivalent positions with "gap" are identified, extended to maximum, and reported. Note that the levels do not to be saved, all we need is a previous level to compute the new level (which will become the previous level in the next round). When all classes reach its final singleton stage, the algorithm terminates.

How to compute next level from the previous level – refinement of a class by class. Consider a refinement of a class $\mathcal{C}$ on level $L$ by class $\mathcal{D}$ on level $L$: take $i,j \in \mathcal{C}$, if $i{+}1, j{+}1 \in \mathcal{D}$, then we leave them together, otherwise we must separate them.
For instance, let us refine class $\mathcal{C} = \{0,2,3,5,7,8,10,11,13\}$ by class $\mathcal{D} = \{1,4,6,9,12,14\}$ on level 1. 0 and 2 must be separated as 1,3 are not both in $\mathcal{D}$, 0 and 3 will be in the same class, since 1,4 are both in $\mathcal{D}$. In fact $\mathcal{C}$ will be refined into two classes, one consisting of $\mathcal{D}$ shifted one position to the left ($\{0,3,5,8,11,13\}$), and the ones that were separated ($\{2,7,10\}$). If we use all classes for refinement, we end up with the next level.

A major trick is not to use all classes for refinement. For each "family" of classes (classes that were formed as a refinement of a class on the previous level – for instance classes $\{2,7,10\}$ and $\{0,3,5,8,11,13\}$ on level 2 form a family as they are a refinement of the class $\{0,2,3,5,7,8,10,11,13\}$ on level 1). In each family we identify the largest class and call all the others small. By using only small classes for refinement, $O(n \log n)$ complexity is achieved as each element belongs only to $O(\log n)$ small classes.

Many linked lists are needed to be maintained to kept track of classes, families, the largest classes in families, and gaps. Care must be taken to avoid traversing any of these structures not to destroy the $O(n \log n)$ complexity. It was estimated that an implementation of Crochemore algorithm requires about 20*$n$ machine words. FSX03 ([FSX03]) managed to trim it down to 15*$n$ using memory multiplexing and virtualization without sacrificing either the complexity or the performance.

# 4 Extending Crochemore algorithm to compute runs

One of the features of Crochemore algorithm is that

(a) repetitions are reported level by level, i.e. all repetitions of the same period are reported together, and
(b) there is no order of repetition reporting with respect to the starting positions of repetitions (this is a product of the process of refinement).

and thus the repetitions must be "collected" and "joined" into runs. For instance, for a string $\boldsymbol{x}$=**abaababaabaabab$**, the order of repetitions as reported by the algorithm is shown in Fig.2; it also shows some of the repetitions that have to be joined into runs.

The first aspect of Crochemore algorithm (see (a) above) is good for computing runs, for all candidates of joining must be of the same period. The second aspect (see (b) above) is detrimental, for it is needed to check for joining two repetitions with "neigbouring" starts.

## 4.1 Variant A

In this variant all repetitions for a level are collected, joined into runs, and reported.

The high level logic:

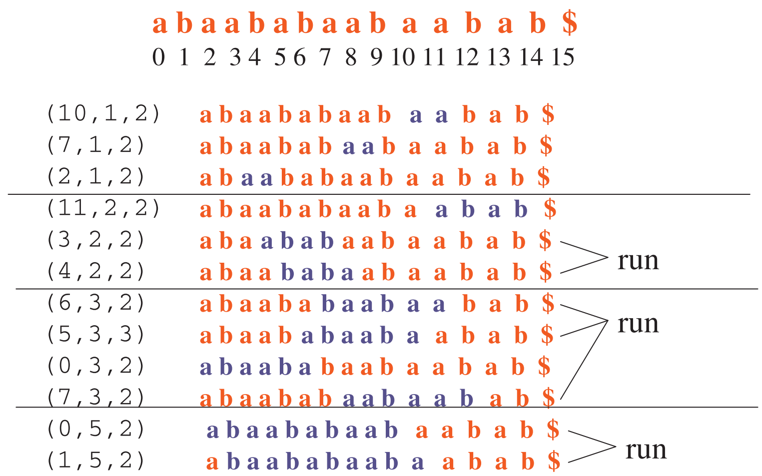


Figure 2: Reporting repetitions for string **abaababaabaabab$**

1. Collect the runs in a binary search tree based on the starting position. There is no need to record the period, as all the repetitions and all the runs dealt with are of the same period.
2. When a new repetition is reported, find if it should be inserted in the tree as a new run, or if it should be joined with an existing run.
3. When all repetitions of the period had been reported, traverse the tree and report all runs (if depth first traversal is used, the runs will be reported in order of their starting positions).

The rules for joining:

1. Descend the tree as if searching for a place to insert the newly reported repetition.
2. For every run encountered, check if the repetition should be joined with it.
   (a) If the repetition is a substring of the run, ignore the repetition and terminate the search.
   (b) If the run is a substring of the repetition, replace the run with the repetition and terminate the search.
   (c) If the run's starting position is to the left of the starting position of the repetition, if the run and the repetition have an overlap of size $\geq p$, the run's tail must be updated to accommodate the repetition (i.e. the run is extended to the right). On the other hand, if the overlap is of size $< p$ or empty, continue search.
   (d) If the run's starting position is to the right of the starting position of the repetition, if the repetition and the run have an overlap of size $\geq p$, the run's starting position must be updated to accommodate the repetition (i.e. the run is extended to the left). On the other hand, if the overlap is of size $< p$ or empty, continue search.

For technical reasons and to lower memory requirements, the runs are recorded in the search tree as pairs $(\boldsymbol{s}, \boldsymbol{d})$ where $\boldsymbol{s}$ is the starting position of the run, while $\boldsymbol{d}$ is the end position of the run (let us remark again that we do not need to store the period $\boldsymbol{p}$). Note that we can easily compute the exponent: $\boldsymbol{e} = (\boldsymbol{d}\text{-}\boldsymbol{s}+1) \ / \ \boldsymbol{p}$, and the tail $\boldsymbol{t} = (\boldsymbol{d}\text{-}\boldsymbol{s}+1) \ \% \ \boldsymbol{p}$.

To avoid dynamic memory allocation and the corresponding deterioration of performance, the search tree is emulated by 4 integer arrays of size $n$, named `RunLeft[]` (emulating pointers to the

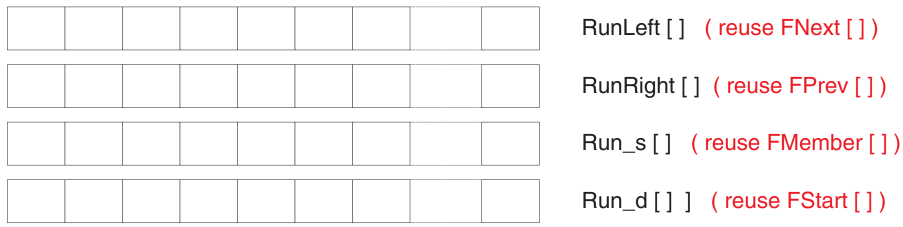


Figure 3: Data structures for Variant A, in red are the arrays that can be reused.

left children), `RunRight[]` (emulating pointers to the right children), `Run_s[]` (emulating storing of the starting position in the node), and `Run_d[]` (emulating storing of the end position in the node), see Fig. 3. Since the four arrays, `FNext[]`, `FPrev[]`, `FMember[]`, and `FStart[]`, are used in the underlying Crochemore algorithm only for class refinement, and at the time of repetition reporting they can be used safely (as long as they are properly "cleaned" after the use), we do not need any extra memory.

Thus the variant A does not need any extra memory as each search tree is "destroyed" after the runs have been reported, however there is an extra penalty of traversing a branch of the search tree for each repetition reporting, i.e. an extra $O(\log n)$ steps, leading to the complexity of $O(n \log^2 n)$.

## 4.2 Variant B

In this variant all repetitions for all levels are collected, joined into runs, and reported together at the end.

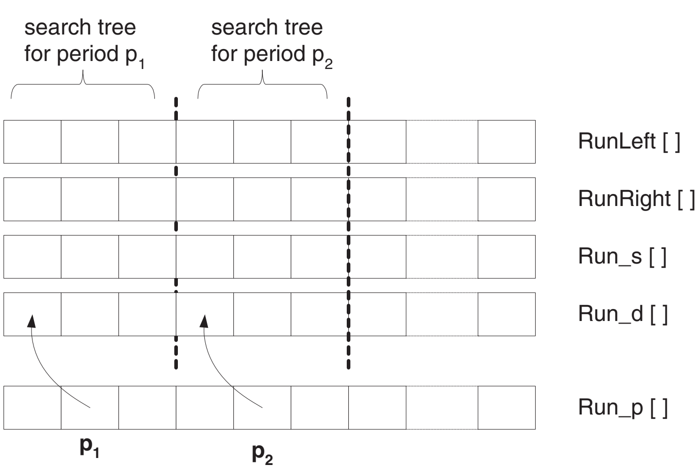


Figure 4: Data structures for Variant B.

The basic principles are the same as for variant A. However, for each level we build a separate search tree and keep it till the repetitions of all levels (periods) have been reported. We cannot use any of the data structures from the underlying Crochemore algorithm as we did for variant A, so the memory requirement grows by additional $5{*}n$ machine words. The time-complexity is the same as for variant A, i.e. $O(n \log^2 n)$.

How do we know that all the runs can fit into the search trees with $n$ nodes together? We do not know, for it is just a conjecture that the maximum number of runs $< n$. However, if we run out of the space (there is a safeguard), we will have found a counterexample to the conjecture on the maximum number of runs (see e.g. [CI08]).

## 4.3 Variant C

In this variant all repetitions for all levels are collected, joined into runs, and reported together at the end.

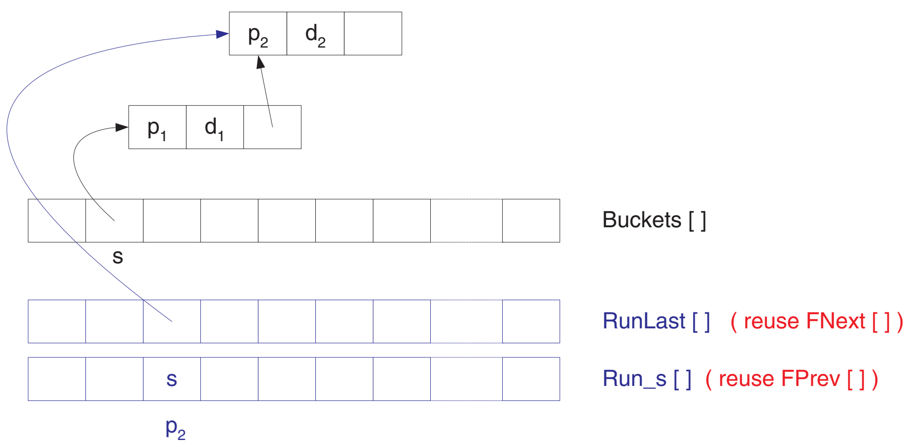


Figure 5: Data structures for Variant C.

The repetitions are collected in a simple data structure consisting of an array `Buckets[]`. In the bucket `Buckets[s]` we store a simple singly-linked list of all repetitions that start at position ***s***. To avoid as much as possible dynamic allocation, so-called "allocation-from-arena" technique is used for the linked lists (`Buckets[]` is allocated with other structures) and $3{*}N$ words is allocated in chunks as needed. The memory requirement for collecting and storing all the repetitions is $\leq 4n{*}\log n$ words, however an expected memory requirement is $4n$ words as the expected number of repetitions is $n$ ($3n$ for the links, $n$ for the buckets).

After all repetitions had been reported and collected, `Buckets[]` is traversed from left to right and all repetitions are joined into runs. In another traversal, the runs can be reported. During the traversal, everything to the left of the current index are runs, while everything to the right and including the current index are repetitions. For the joining business, we need for each period to remember the rightmost run with that period, that is the role of the array `RunLast[]` (we can reuse `FNext[]`). Thus when traversing the linked list in the bucket `Buckets[i]` and currently dealing with a repetition with period ***p2***, `RunLast[p2]` points to the last run of period ***p2*** so we can decide if the current repetition is to be "promoted" to a run (with a zero tail), or joined with the run. Since the starting position of the last run of period ***p2*** is not stored in the run, we need one more array `Run_s[]` in which we store the starting position (we can reuses `FPrev[]`).

Since storing a repetition in `Buckets[]` takes a constant time, and there are $O(n \log n)$ repetitions, and since the joining business is also constant time, the overall time complexity is $O(n \log n) +$

$O(n \log n)$, i.e. $O(n \log n)$.

## 5 Experimental results

Implementations of the three variants were compared as to their performance. The testing was rather informal, just to give indication how the three variants compare.
Hardware: Sony VAIO laptop with Intel Core-2 Duo CPU T5800 @ 2.00 GHz, 4GB of RAM
Software: Windows Vista Home Premium SP1. Code was written in C++ and was compiled using the GNU g++ compiler.
Each run was repeated five times, the minimum numbers are recorded in the following table (random2.txt is a file of random strings on a binary alphabet, while random21.txt is a file of random strings on an alphabet of size 21):

| Data set | File name | String length | Time (seconds) | | |
|---|---|---|---|---|---|
| | | | variant A | variant B | variant C |
| DNA | dna.dna4 | 510976 | 105.87 | 110.15 | 3.12 |
| English | bible.txt | 4047392 | 63.27 | 62.65 | 23.90 |
| Fibonacci | fibo.txt | 305260 | 173.30 | 177.00 | 2.39 |
| Periodic | fss.txt | 304118 | 159.61 | 168.78 | 2.44 |
| Protein | p1Mb.txt | 1048576 | 47.93 | 53.23 | 5.15 |
| Protein | p2Mb.txt | 2097152 | 189.20 | 189.98 | 11.42 |
| Random | random2.txt | 510703 | 193.01 | 189.28 | 4.42 |
| Random | random21.txt | 510703 | 7.69 | 7.46 | 1.89 |

The following table records the performance averaged per a character of the input string:

| Data set | File name | Time (μsec / letter) | | |
|---|---|---|---|---|
| | | variant A | variant B | variant C |
| DNA | dna.dna4 | 207.18 | 215.57 | 6.11 |
| English | bible.txt | 15.63 | 15.48 | 5.91 |
| Fibonacci | fibo.txt | 567.70 | 579.83 | 7.82 |
| Periodic | fss.txt | 524.84 | 554.98 | 8.01 |
| Protein | p1Mb.txt | 45.71 | 50.76 | 4.91 |
| Protein | p2Mb.txt | 90.22 | 90.59 | 5.45 |
| Random | random2.txt | 377.93 | 370.62 | 8.64 |
| Random | random21.txt | 15.06 | 14.60 | 3.70 |
| Overall average | | 230.53 | 236.55 | 6.32 |

The results allow for a quick conclusion:

1. Overall, variant C is significantly faster than variants A and B. In fact by 3643 %!
2. Even though variant A requires less additional memory, speed-wise does not do much better than B.
3. The speed of variants A and B is not proportional to the string length. Rather, it mostly depends on the type of the string. It works better on strings with large alphabet size and low periodicity. This is intuitively clear, as for high periodicity strings the height of the search trees are large.

# 6 Conclusion and further research

We extended Crochemore repetition algorithm to compute runs. Of the three variants, variant C is by far more efficient time-wise, but requiring $O(n \log n)$ additional memory. However, its performance warranties further investigation and an implementation that further reduces memory requirements. Variant C should be the one used for the parallelization. Let us remark that variant C could be used as an extension of any repetition algorithm.

# References


[AKO4] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*, J. Discr. Algorithms 2 (2004), 53-86

[CPS07] G. Chen, S.J. Puglisi & W. F. Smyth, *Fast & practical algorithms for computing all the runs in a string*, Proc. 18th Annual Symposium on Combinatorial Pattern Matching (2007) 307-315.

[C81] M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inform. Process. Lett., 5 (5) 1981, pp. 297–315.

[CI08] M. Crochemore and L. Ilie: *Maximal repetitions in strings.* Journal of Computer and System Sciences 74-5 (2008), 796–807.

[F97] M. Farach: *Optimal suffix tree construction with large alphabets*, $38^{th}$ IEEE Symp. Found. Computer Science (1997) 137–143.

[FSX03] F. Franek, W.F. Smyth, and X. Xiao: *A note on Crochemore's repetitions algorithm, a fast space-efficient approach*, Nordic J. Computing 10-1 (2003) 21-28.

[KK00] R. Kolpakov and G. Kucherov: *On maximal repetitions in words.* J. of Discrete Algorithms, (1) 2000, pp. 159–186.

[KS03] J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, Proc. $30^{th}$ Internat. Colloq. Automata, Languages & Programming (2003), 943–955.

[KA03] P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, Proc. $14^{th}$ Annual Symp. Combinatorial Pattern Matching, R. Baeza-Yates, E. Chávez, and M. Crochemore (eds.), LNCS 2676, Sprinegr-Veralg (2003), 200-210.

[M89] M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Maths., (25) 1989, pp. 145–153.

[MM93] U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput. 22 (1993), 935–938

[PST05] S. J. Puglisi, W. F. Smyth, and A. Turpin: *Some restrictions on periodicity in strings*, in Proceedings of 16th Australasian Workshop on Combinatorial Algorithms AWOCA 2005, University of Ballarat, Victoria, Australia, September 18-21 2005, pp. 263–268.

[W73] P. Weiner: *Linear pattern matching algorithms*, in Proceedings of 14th Annual IEEE Symp. Switching & Automata Theory (1973) 1–11. University of Ballarat, Victoria, Australia, September 18-21 2005, pp. 263–268.