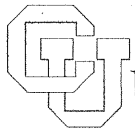
An 0 (n log n) Algorithm for Finding All Repetitions in a String

Michael Main
Richard J. Lorentz

CU-CS-241-82



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

AN O(n log n) ALGORITHM FOR FINDING ALL REPETITIONS IN A STRING

by

Michael G. Main *

and

Richard J. Lorentz **

CU-CS-241-82 November, 1982

*Department of Computer Science University of Colorado at Boulder, Boulder, Colorado 80309

**Department of Mathematics, Claremont-McKenna College, Claremont, CA 91711

AN $O(n \log n)$ ALGORITHM FOR FINDING ALL REPETITIONS IN A STRING*

Michael G. Main
Department of Computer Science
University of Colorado
Boulder, CO 80309

Richard J. Lorentz
Department of Mathematics
Claremont-McKenna College
Claremont, CA 91711

ABSTRACT

Any nonempty string of the form xx is called a *repetition*. An $O(n \log n)$ algorithm is presented to find all repetitions in a string of length n. The algorithm is based on a linear algorithm to find all the new repetitions formed when two strings are concatenated. This linear algorithm is possible because new repetitions of equal length must occur in blocks with consecutive starting positions. The linear algorithm uses a variation of the Knuth-Morris-Pratt algorithm to find all partial occurrences of a pattern within a text string. It is also shown that no algorithm based on comparisons of symbols can improve $O(n \log n)$. Finally, some open problems and applications are suggested.

*This work was supported in part by National Science Foundation Grants MCS-7708486 and MCS-8003433.

1. INTRODUCTION

A *repetition* is an immediately repeated nonempty string, e.g., aa, $abab$, $wallawalla$. Any string which contains a repetition as a substring is called *repetition-containing*. A string which is not repetition-containing is *repetition-free*. In this paper we address the problem of finding all substrings of a string which are repetitions. The $O(n \log n)$ algorithm we give can also be used to find the longest repetition in a string, or just to determine when a string is repetition-free.

The study of repetitions in strings dates back to the pioneering work of Axel Thue in the first decade of this century [21,22]. Using homomorphisms which preserve repetition-free strings, he constructed infinite repetition-free sequences on an alphabet of three characters. Since then, others have presented this same result, applying it in diverse fields [3,9,10,14,16,17]. Several applications of Thue's result to modern formal language theory have appeared in the last decade [4,8,18,20].

The problem of determining when a string is repetition-free is similar to several other pattern matching problems. Work by Seiferas and Galil [19] has provided algorithms to determine whether a string has a repetition conforming to specific length restrictions. Various patterns involving palindromes have also been the subject of recognition algorithms [7,15].

The algorithm we present finds all repetitions in a string in time $O(n \log n)$, where n is the string's length. The algorithm is based on a linear procedure for finding all new repetitions that are formed when two strings are concatenated. This linear procedure is somewhat surprising, since there may be $\Omega(|uv|^2)$ new repetitions formed when the strings u and v are concatenated*. The linear time is obtained by taking advantage of the fact that new repetitions of equal length occur in blocks with consecutive starting positions. By using a variation of the Knuth-Morris-Pratt pattern matching algorithm

* A function $f(n)$ is called $\Omega(g(n))$ when there is a constant c such that for any sufficiently large n, $f(n) \geq cg(n)$.

[13] to precompute some information, all the blocks of new repetitions are found in $O(|uv|)$. This linear algorithm can be used to construct an $O(n \log n)$ algorithm to find all repetitions. The $O(n \log n)$ algorithm is easily modified to find a maximal length repetition or just to determine whether a string is repetition-free. In fact, it can find all repetitions of length $\frac{n}{k}$ or greater in time $O(n \log k)$. Finally, $O(n \log n)$ is shown to be a lower bound for any algorithm which uses symbol comparisons to determine whether a string is repetition-free.

An alternate solution to the problem of finding all repetitions in a string has recently been given by Max Crochemore [5]. His $O(n \log n)$ algorithm is based on finding equivalence relations of the positions of the string.

Our notation follows Aho, Hopcroft and Ullman [1]. The length of a string x is denoted by $|x|$. A substring of x beginning at the i^{th} character and ending at the j^{th} character is written as $x_i \cdots x_j$.

2. FINDING ALL PARTIAL MATCHES OF A PATTERN

Let *pattern* and *text* be strings of length m and n, respectively. This section shows how to find all partial matches of *text* with the beginning of *pattern* in $O(n)$ time. The algorithm is a variation of the Knuth-Morris-Pratt pattern matching algorithm* [13]. Specifically, we can compute the following arrays:

lppattern: $lppattern[i]$ is the length of the longest substring of *pattern* which begins at position i and is a prefix of *pattern* $(1 < i \leq m)$. This array may be computed in time $O(m)$.

lptext: $lptext[i]$ is the length of the longest substring of *text* which begins at position i and is a prefix of *pattern* $(1 \leq i \leq n)$. This array may be computed in time $O(n)$.

We begin with the computation of *lppattern*. The algorithm computes the array in order of increasing subscripts, in such a way that no backtracking is done. Suppose we have calculated $lppattern[2] \cdots lppattern[i-1]$ and now we want to find $lppattern[i]$. Suppose also that we have remembered the value of k in the range $2 \leq k < i$ which maximizes the sum $k + lppattern[k]$. If $i < k + lppattern[k]$, then the value of $lppattern[i-k+1]$ (which is already computed) gives some information on what $lppattern[i]$ is. The reason is this: When i is in this range, the substring $x = pattern_i \cdots pattern_{k+lppattern[k]-1}$ is identical to the substring $pattern_{i-k+1} \cdots pattern_{lptext[k]}$. Hence, if $lppattern[i-k+1] < |x|$, then $lppattern[i] = lppattern[i-k+1]$. On the other hand, if $lppattern[i-k+1] \geq |x|$, then $lp[i]$ is at least $|x|$. Algorithm 1 uses this idea to compute *lppattern*.

*The variation of the KMP algorithm which we use to calculate *lppattern* was suggested by a referee. We are grateful for this suggestion, which greatly simplifies the presentation of our algorithm in section 3.

Algorithm 1: Calculating *lppattern*.

Input: A string *pattern* of length m. It is assumed that *pattern* is padded with an unmatchable character, so that an attempt to match $pattern_{m+1}$ always results in a mismatch.

Output: $lppattern[2] \cdots lppattern[m]$, an array where $lppattern[i]$ is the length of the longest substring of *pattern* which begins at position i and is a prefix of *pattern*.

```
/* First find lppattern[2]. */
j ← 0;
while (pattern_{j+1} = pattern_{j+2}) do j ← j+1;
lppattern[2] ← j;
k ← 2;

for i ← 3 to m do begin
        /* calculate lppattern[i] */
        /* length is the length of pattern_i ··· pattern_{k+lppattern[k]-1} */
        length ← k + lppattern[k] − i;
        if (lppattern[i−k+1] < length)
        then lppattern[i] ← lppattern[i−k+1]
        else begin
                /* j is a provisional value of lppattern[i] */
                if (i ≥ k + lppattern[k]) then j ← 0
                else j ← length;
                while (pattern_{j+1} = pattern_{j+i}) do j ← j+1;
                lppattern[i] ← j;
                k ← i
        end
end
```

The **for** loop of the *lppattern* algorithm requires at most $O(m)$ time. This can be shown by analysis of the inner **while** loop. In each iteration of the **while** loop, the character $pattern_{j+i}$ has not yet successfully been matched against another character of *pattern*. Hence, there are at most m iterations of the **while** loop with a *true* test and also at most m iterations with a *false* test.

Once the *lppattern* array has been computed, it is easy to modify algorithm 1 to find *lptext*. The required modification is shown in algorithm 2.

Algorithm 2: Calculating *lptext*.

Input: A string *pattern* of length m and a string *text* of length n. Also, the *lppattern* array, computed by algorithm 1. It is assumed that *pattern* and *text* have been padded with an unmatchable character, so that attempts to match $pattern_{m+1}$ or $text_{n+1}$ always result in a mismatch.

Output: $lptext[1] \cdots lptext[n]$, an array where $lptext[i]$ is the longest substring of *text* which begins at position i and is a prefix of *pattern*.

```
/* First find lptext[1]. */
j ← 0;
while (pattern_{j+1} = text_{j+1}) do j ← j+1;
lptext[1] ← j;
k ← 1;

for i ← 2 to n do begin
        /* calculate lptext[i] */
        /* length is the length of the substring text_i ··· text_{k+lptext[k]-1} */
        length ← k + lptext[k] − i;
        if (lppattern[i−k+1] < length)
        then lptext[i] ← lppattern[i−k+1]
        else begin
                /* j is a provisional value of lptext[i] */
                if (i ≥ k + lptext[k]) then j ← 0
                else j ← length;
                while (pattern_{j+1} = text_{j+i}) do j ← j+1;
                lptext[i] ← j;
                k ← i
        end
end
```

The *lptext* algorithm is $O(n)$ for the same reason that calculating *lppattern* was $O(m)$. With the precomputing of *lppattern*, the total time to find *lptext* is $O(m+n)$ -- although in practice this will be $O(n)$, since if $m > n$, then we need only compute the first n values of *lppattern*.

Finally, note that the same idea used to compute *lptext* can be used to calculate the longest suffix of *pattern* that ends at each position i in *text*. The next section uses longest suffix and longest prefix arrays to find repetitions.

3. DETECTING FORMATION OF REPETITION

For this section, u and v are any strings. We give an $O(|uv|)$ algorithm to find all the *new* repetitions that are formed when u is concatenated to v -- i.e., those repetitions in uv which begin in u and end in v. Considering that there could be $\Omega(|uv|^2)$ such repetitions, a linear algorithm is an unexpected result. The algorithm is possible only because repetitions of equal length occur in blocks with consecutive starting positions.

The new repetitions in uv are classified into two groups, according to where their centers lie. A *right* repetition has its center at or to the right of the boundary between u and v. A *left* repetition has its center at or to the left of this boundary. The important observation for us is that every repetition in uv is either *right* or *left* (or both). Hence, to find all new repetitions in uv, we need only find all new right repetitions and all new left repetitions.

The following describes a procedure *newright*(u,v) which finds all new right repetitions in time $O(|v|)$. By symmetry, all the new left repetitions can be found in time $O(|u|)$. The procedure *newright*(u,v) makes use of the following information:

- $ls[i]$ $(1 \le i \le |v|)$ is the length of the longest suffix of u which occurs in v, ending at position i.
- $lp[i]$ $(2 \le i \le |v|+1)$ is the length of the longest prefix of v which occurs in v, beginning at position i $(lp[|v|+1] \equiv 0)$.

As shown in the previous section, both these arrays can be computed in time $O(|v|)$.

The *newright* procedure detects all of the new right repetitions of a fixed length at once. In particular, it finds the positions in v at which these repetitions end. The first observation is that if a new right repetition of length $2n$ ends at position i in v, then $n \le i < 2n$. (If i is less than n, then it isn't a right repetition; if i is $2n$ or larger, then it isn't a new repetition.) The following lemma tells us how to use the arrays ls and lp to determine which positions in this range are the ends of repetitions with length $2n$.

Repetition Lemma. *Let n and i be integers with $1 \le n \le |v|$ and $n \le i < 2n$. There is a repetition of length $2n$ in uv ending at position i of v iff $2n - ls[n] \le i \le n + lp[n+1]$.*

Proof. Let n and i be integers in the given ranges. A repetition of length $2n$ in uv, that ends at v_i, must begin at $u_{|u|-2n+i+1}$. The beginning of its second half must be at v_{i-n+1}. Thus, a necessary and sufficient condition for such a repetition to occur is that the substring $u_{|u|-2n+i+1} \cdots u_{|u|} v_1 \cdots v_{i-n}$ matches $v_{i-n+1} \cdots v_i$. Equivalently:

(1) $u_{|u|-2n+i+1} \cdots u_{|u|} = v_{i-n+1} \cdots v_n$, and

(2) $v_1 \cdots v_{i-n} = v_{n+1} \cdots v_i$. (This second condition is vacuous if $i = n$.)

The first condition is exactly the statement that the string $v_{i-n+1} \cdots v_n$ of length $2n - i$ is a suffix of u, which is true iff $ls[n] \ge 2n - i$. This may be rewritten as $2n - ls[n] \le i$. Similarly, the second condition reduces to $i \le n + lp[n+1]$. □

The repetition lemma is the basis of algorithm 3 to find all new right repetitions in uv:

Algorithm 3. Procedure to find all new right repetitions in string uv.

Input: Two strings u and v.

Output: The ending positions of all new right repetitions in uv.

procedure *newright*(u,v) **begin**

calculate $ls[1] \cdots ls[|v|]$ and $lp[2] \cdots lp[|v|+1]$.

for $n \leftarrow 1$ **to** $|v|$ **do begin**

/* *first* is the first position in v where a new right repetition may end */
$first \leftarrow 2n - ls[n]$;

/* *last* is the last position in v where a new right repetition may end */
$last \leftarrow$ **minimum**$(2n-1, n + lp[n+1])$;

if $(last < first)$ **then** there are no new right repetitions of length $2n$
else new right repetitions of length $2n$ end at v_{first} through v_{last};

end

end.

The calculation of *ls* and *lp* requires time $O(|v|)$, as shown in the previous section. The **for** loop also requires $O(|v|)$, hence *newright* takes $O(|v|)$ time to find all new right repetitions in uv. A symmetric algorithm *newleft*(u,v) takes $O(|u|)$ time to find all new left repetitions in uv. The procedures *newright* and *newleft* can be combined to find all new repetitions in uv in $O(|uv|)$ time. In the next section, we call this linear algorithm *new*(u,v), and use it to find all repetitions in a string.

4. FINDING ALL REPETITIONS IN A STRING

This section describes an algorithm to find all repetitions in a string of length n, in time $O(n \log n)$. It uses the procedure *new* of the last section.

Let w be the string of length n. The repetitions in w are of these three sorts:

(1) The repetitions which end at or before the $\lceil n/2 \rceil^{th}$ position of w.

(2) The repetitions which start after the $\lceil n/2 \rceil^{th}$ position of w.

(3) the repetitions which start at or before the $\lceil n/2 \rceil^{th}$ position of w, and end after it.

In algorithm 4, the procedure *findreps*(w) finds repetitions of the third kind with a call to *new*$(w_1 \cdots w_{\lfloor n/2 \rfloor}, w_{\lfloor n/2 \rfloor+1} \cdots w_n)$. (Recall that *new*$(u,v)$ finds all repetitions which span the border between u and v.) Repetitions of the first two sorts are found with recursive calls *findreps*$(w_1, \cdots, w_{\lfloor n/2 \rfloor})$ and *findreps*$(w_{\lfloor n/2 \rfloor+1}, \cdots, w_n)$.

Algorithm 4. Procedure to find all repetitions in a string.

Input: Any string w.

Output: The positions of all repetitions in w.

```
procedure findreps(w) begin
    if (|w| ≤ 1) then w is repetition-free
    else begin
        new(w_1 ··· w_⌊n/2⌋, w_⌊n/2⌋+1 ··· w_n);
        findreps(w_1 ··· w_⌊n/2⌋);
        findreps(w_⌊n/2⌋+1 ··· w_n)
    end
end.
```

Let $T(n)$ be the time *findreps* takes on an input of length n. The function $T(n)$ gives rise to the recurrence:

$$T(1) = d$$

$$T(n) = cn + 2T(n/2),$$

for some constants c and d. The cn comes from the call to *new*, which takes linear time. The $2T(n/2)$ arises from the two recursive calls to *findreps*. The solution to this recurrence is $O(n \log n)$ [1, theorem 2.1], hence on an input of length n, *findreps* takes $O(n \log n)$ time.

The *findreps* algorithm is easily modified to find an instance of a maximal length repetition, or to simply determine if a string in repetition-free. It can also find all repetitions of length $\frac{n}{k}$ or greater in time $O(n \log k)$.

If we know nothing about the size of the alphabet, then $O(n \log n)$ is optimal for algorithms based on comparisons of symbols. In fact, no algorithm can determine if a string is repetition-free in less time*. To show this, consider a string w such that no two symbols of w are equal. What information does a single comparison between two positions w_i and w_j provide? The only information it can provide is $w_i \neq w_j$. It cannot tell us whether w_i is equal to any other symbols of w, since inequality is not transitive. Hence, the only repetitions eliminated by this comparison are those that require w_i to equal w_j. If k is the absolute value of $(i-j)$, then there are k such repetitions, each having length $2k$.

Now, if $|w| = n$, then there are $n-2k+1$ possible different repetitions of length $2k$, for $1 \leq k \leq n/2$. Hence, to eliminate all possible repetitions of length $2k$ we must make at least $(n-2k+1)/k$ comparisons. To determine that w is repetition-free requires at least the following number of comparisons:

$$\sum_{k=1}^{n/2} (n-2k+1)/k$$

The dominating term is:

$$\left\{ \sum_{k=1}^{n/2} n/k \right\} = nH_{n/2},$$

where $H_{n/2}$ is the sum of the harmonic series to $n/2$. The value of $H_{n/2}$ is $O(n \log n)$ [12], hence the minimum number of comparisons required is $O(n \log n)$.

*This optimality proof is an idea of David Benson and Karl Winklmann.

5. CONCLUSION

Given a string w, of length n, the algorithm of the previous section finds all repetitions in w in time $O(n \log n)$. If we know nothing about the size of the alphabet, then this is optimal for algorithms based on comparisons of symbols. In fact, it is optimal for any algorithm which determines if w is repetition-free.

A primary application of the algorithm is in string combinatorics research. For example, Bean *et al.* [2] needed to show that a certain set of strings (each of which had more than 600 characters) contained no repetition-containing strings. The algorithm could also be useful in string compaction (i.e., replace a substring xx by a single copy of x and a repeat instruction) and in DNA pattern recognition.

Several open problems remain. If we restrict the size of the alphabet, is there a faster algorithm to determine when a string is repetition-free? In particular, for the first nontrivial case, an alphabet of three characters, is there a faster recognition algorithm?

A second open question involves the detection of permuted repetitions, or *permutations*. A permutation is a nonempty string of the form $x\hat{x}$, where $\hat{x}$ contains the same characters as x, but possibly in a different order. A string which has a substring that is a permutation is called *permutation-containing*. Otherwise it is *permutation-free*. Evdokimov first constructed infinite permutation-free sequences using a 25-character alphabet [6]. Pleasants reduced the alphabet size to 5 characters [17]. The question of a 4-character alphabet is still open, although Justin has constructed permutation-free strings of length 7500 on this alphabet [11]. How quickly can an algorithm determine whether a string is permutation-free? How fast can all permutations be found? For both questions, the fastest algorithm we know of is $O(n^2)$. A faster algorithm would be useful in computer research toward constructing infinite permutation-free sequences on 4 characters.

ACKNOWLEDGEMENTS

The idea to precompute the array *lp* was suggested by a referee. From there, the idea of also precomputing *ls* was straightforward. Much to our delight, we found this allowed us to find all repetitions. (The original version only determined if a string was repetition-free.) The optimality proof was a joint effort of David Benson and Karl Winklmann. In addition to these people, we want to thank Narsingh Deo, George Marsaglia and Denbigh Starkey for reading preliminary versions of this paper and making suggestions on how to improve the presentation.

Bibliography

(1) Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1976.

(2) Bean, D.R., A. Ehrenfeucht and G.F. McNulty, Avoidable patterns in strings of symbols, *Pacific J. Math.*, 85 (1979), 261-294. A study of words which contain no nonempty substring of the form x^k (k fixed), and of the homomorphisms which preserve this property. One interesting result is a repetition-free homomorphism from strings over a four letter alphabet to strings over a three letter alphabet. The images of the letters are of length 202, 209, 209 and 216.

(3) Braunholtz, C.H., Solution to problem 5030, *Am. Math. Mo.* 70 (1963), 675-676. A clever construction of an infinite repetition-free sequence, beginning with the sequence of integers.

(4) Brzozowski, J.A., K. Culik II and A. Gabrielian, Classification of noncounting events, *JCSS* 5 (1971) 41-53. The existence of infinite repetition-free sequences is used to construct a noncounting language of order 2 that is not recursively enumerable.

(5) Crochemore, M., Determination optimale des repetitions dans un mot, Technical Report 80-53, Laboratoire Informatique Theorique et Programmation, University of Paris (1981).

(6) Evdokimov, A.A., Strongly asymmetric sequences generated by a finite number of symbols, *Soviet Math. Dokl.*, 9 (1968), 536-539. Evdokimov constructs an infinite permutation-free sequence on an alphabet of 25 letters.

(7) Galil, Z., and J. Seiferas, A linear-time on-line recognition algorithm for palstar, *JACM* 25 (1978), 102-111.

(8) Goldstin, J., Bounded AFLS, *JCSS* 12 (1976) 399-419. Uses the existence of infinite repetition-free sequences to build a context-sensitive language that's not generable from bounded languages using only AFL operations and intersection.

(9) Harrison, M., *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA., (1978), 36-40. Gives Thue's construction of an infinite repetition-free sequence, for modern formal language theorists.

(10) Hedlund, G.A., Remarks on the work of Axel Thue on Sequences, *Nord. Mat. Tidskr.* 16 (1967), 148-150. MR 37 (1959), #4454. Hedlund summarizes some of the many discoveries and rediscoveries of Thue's work, including his own paper with Marston Morse.

(11) Justin, J., Generalisation du theoreme de van de Waerden sur les semi-groupes repetitifs, *J. Comb. Theory* (A) 12 (1972), 357-367. Justin asks whether there are infinite permutation-free sequences on an alphabet of four letters. In support of the claim that such strings exist is his computer construction of a permutation-free string of length 7500.

(12) Knuth, E.E., *Fundamental Algorithms*, Addison-Wesley (1973), 73-77.

(13) Knuth, D.E., J.H. Morris, Jr., and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* 6 (1977), 323-350.

(14) Leech, J., A problem on strings of beads, *Math. Gaz.* 41 (1957), 277-278. Gives yet another infinite repetition-free sequence.

(15) Manacher, G., A new linear-time on-line algorithm for finding the smallest initial palindrome of a string, *JACM* 22 (1975), 346-351.

(16) Morse, M. and G.A. Hedlund, Unending chess, symbolic dynamics and a problem in semigroups, *Duke Math. J.* 11 (1944), 1-7. MR 5 (1944) #202. This paper constructs an infinite repetition-free sequence on three letters. The method uses a repetition-free sequence on two characters and was used to establish the existence of nonperiodic recurrent motions in symbolic dynamics.

(17) Pleasants, P.A.B., Nonrepetitive sequences, *Proc. Cambridge Phil. Soc.* 68 (1970), 267-274. Pleasants constructs an infinite permutation-free sequence on five letters, and poses the question of whether this can be done with just four letters.

(18) Reutenauer, C., A new characterization of the regular languages, in *Eighth International Conference on Automata and Programming Languages*, LNCS 115, Springer-Verlag, Berlin, 1981, 175-813. Uses infinite repetition-free sequences to construct a language that is periodic but not regular.

(19) Seiferas, J., and Z. Galil, Real-time recognition of substring repetition and reversal, *Math. Sys. Theory* 11 (1977), 111-146. This linear algorithm recognizes strings of the form $wxyxz$, where the lengths of the substrings w,x,y and z meet certain restrictions.

(20) Shyr, H.J., A strongly primitive word of arbitrary lengh and its applications, *Int. J. Comp. Math.* A-6 (1977), 165-170. Uses a simple infinite repetition-free sequence on four letters to construct a language which is noncounting, but not t.v.a.a.

(21) Thue, A., Uber unendliche Zeichenreihen, *Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania)* (1906), Nr. 7, 1-22.

(22) Thue, A., Uber die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, *Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania)* (1912), Nr. 1, 1-67.