String Range Matching

Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi

Department of Computer Science, University of Helsinki Helsinki, Finland
firstname.lastname@cs.helsinki.fi

Abstract. Given strings X and Y the *exact string matching* problem is to find the occurrences of Y as a substring of X. An alternative formulation asks for the lexicographically consecutive set of suffixes of X that begin with Y. We introduce a generalization called *string range matching* where we want to find the suffixes of X that are in an arbitrary lexicographical range bounded by two strings Y and Z. The problem has applications in distributed suffix sorting, where Y and Z are themselves suffixes of X. Exact string matching can be solved in linear time and constant extra space under the standard comparison model. Our conjecture is that string range matching is a harder problem and cannot be solved within the same time–space complexity. In this paper, we trace the upper bound on the complexity of string range matching by describing algorithms that are within a logarithmic factor of the time–space complexity of exact string matching, as well as variants of the problem and the model that can be solved in linear time and constant extra space.

1 Introduction

Exact string matching, the problem of finding all the occurrences of a string $\mathsf{Y}[0..m)$ (the pattern) in a larger string $\mathsf{X}[0..n)$ (the text) is a foundational problem in computer science, and has applications throughout modern computer software. Among the numerous algorithms for exact string matching [5], there are several that are optimal in space ($\mathrm{O}(1)$ extra space) as well as in time ($\mathrm{O}(n+m)$) under the comparison model [9, 8, 3, 2, 4]. We have recently considered a generalization called *longest prefix matching*, where we want to find the occurrences of the longest prefix of the pattern that occurs in the text, and have shown that (at least) one of these algorithms can be generalized to longest prefix matching within the same optimal time–space complexity [11].

In this paper we introduce a further generalization based on an alternative view of the above problems. If we regard the text as a collection of its suffixes, the above problems can be stated as reporting the starting positions of all suffixes that begin with the pattern or with the longest prefix of the pattern producing a non-empty result. The resulting subset of suffixes is lexicographically consecutive and the query could be expressed as a lexicographical range query on the set of suffixes. A natural generalization, which we call *string range matching*, then asks for suffixes in an arbitrary lexicographical range: Given strings $\mathsf{X}[0..n)$, $\mathsf{Y}[0..m_1)$ and $\mathsf{Z}[0..m_2)$ report all i such that $\mathsf{Y} \leq \mathsf{X}[i..n) < \mathsf{Z}$.

We describe two basic algorithms for string range matching. One is based Crochemore's string matching algorithm [2] (or its simplification in [11]) and solves the problem in $O(n \log(m_1 + m_2))$ time using constant extra space. For certain important special cases the algorithm can be made to run in linear time, in particular when the strings Y and Z share a prefix of length $\epsilon(m_1+m_2)$ for any constant $\epsilon > 0$. The second algorithm is based on the string matching algorithm by Galil and Seiferas [9] and solves the *counting* version of the problem in linear time using $O(\log(m_1+m_2))$ extra space. In both cases, the modification to solve string range matching is non-trivial.

Furthermore, we show that the problem can be solved in linear time using just constant extra space by slightly cheating about the extra space aspect. We describe two distinct "cheats": (i) The algorithm can overwrite the strings Y and Z with other data but must restore the strings to their original state at the end. The overwritten memory does not count as extra space. (ii) The algorithm has read access to its own output, which does not count as extra space. The algorithm works for most reasonable output formats but only for the reporting version of the problem, not for the counting version.

We conjecture that string range matching is a harder problem than exact string matching and cannot be solved in linear time and constant extra space in the comparison model without cheating, i.e., when the only access to the input is by character comparisons and the output is write-only.

Applications to Suffix Sorting. The suffix array [13] and the Burrows–Wheeler transform [1] are central to modern string processing and the key task in their construction is sorting the suffixes of a string. For long strings we may want to split the task into smaller subtasks in order to distribute the work load or reduce the memory requirements. One approach partitions the lexicographical space [10]. For each partition, we then need to collect all suffixes within the given lexicographical range. The collecting problem is exactly string range matching.

Another approach to suffix sorting for large strings is to split the string into smaller blocks, sort the suffixes of each block separately, and then merge [6]. Assume the string is ABC, where B is the current block. When sorting the suffixes of B, what we really want is the lexicographical ordering of the $|\mathsf{B}|$ longest suffixes of BC, which can differ significantly from the ordering of the suffixes of B as a standalone string. The correct ordering can be obtained without accessing C if we know which of the $|\mathsf{B}|$ longest suffixes of BC are smaller than C. This information can be obtained using a one-sided string range matching query.

In both of the papers mentioned above [10, 6], the string range matching problem is solved using an algorithm introduced in [10], which is based on the Knuth–Morris–Pratt [12] exact string matching algorithm, and is the only prior work on string range matching that we are aware of. However, the $O(m_1 + m_2)$ extra space needed by the algorithm is a problem since Y and Z are potentially very long suffixes of X. The techniques used in those papers to reduce the space requirement (a global data structure [10] and sequential processing of the blocks [6]) are not well suited for distributed computation. Thus our small space algorithms open up new possibilities for distributed suffix sorting.

2 General Framework

Consider a string $\mathsf{X} = \mathsf{X}[0..n) = \mathsf{X}[0]\mathsf{X}[1]\ldots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from an ordered alphabet. Here and elsewhere we use $[i..j)$ as a shorthand for $[i..j-1]$. For $i \in [0..n)$ we write X_i to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}_i = \mathsf{X}[i..n) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n-1]$. We also generalize the notation to sets of suffixes: for any $S \subseteq [0..n)$, $\mathsf{X}_S = \{\mathsf{X}_i \mid i \in S\}$. The *empty string* is denoted by ε. For strings X and Y, we use $\mathsf{lcp}(\mathsf{X},\mathsf{Y})$ to denote the length of the *longest common prefix* of X and Y. If $\mathsf{X} \le \mathsf{Y}$, we write $[\mathsf{X},\mathsf{Y})$ to denote the *lexicographical range* of strings between X and Y, i.e., the set $\{\mathsf{Z} \mid \mathsf{X} \le \mathsf{Z} < \mathsf{Y}\}$. For $\mathsf{X} \le \mathsf{Y} \le \mathsf{Z}$, we have $[\mathsf{X},\mathsf{Z}) = [\mathsf{X},\mathsf{Y}) \cup [\mathsf{Y},\mathsf{Z})$, where $\cup$ denotes the *disjoint union*. A positive integer p is a *period* of X if $\mathsf{X}[i] = \mathsf{X}[i+p]$ for all $i \in [0..n-p)$. If p and q are periods of X and $p+q \le |\mathsf{X}|$, then $\gcd(x,y)$ is a period of X too (Weak Periodicity Lemma [7]). The smallest period of X is denoted $\mathsf{per}(\mathsf{X})$. A string X is called *primitive* if it cannot be written as $\mathsf{X} = \mathsf{Y}^k$ for an integer $k > 1$.

We focus on the one-sided string range matching problem of computing the set $\mathsf{X}_{[0..n)} \cap [\varepsilon,\mathsf{Y})$, to which the two-sided version can be reduced since $[\mathsf{Y},\mathsf{Z}) = [\varepsilon,\mathsf{Z}) \setminus [\varepsilon,\mathsf{Y})$. All our algorithms use a similar basic approach. In a generic step, they compute $\ell = \mathsf{lcp}(\mathsf{X}_i,\mathsf{Y})$ and then i is incremented by h, where either $h = p = \mathsf{per}(\mathsf{Y}[0..\ell))$ or $h < p$ and $h = \Theta(\ell)$. The efficient computation of ℓ and h is based on well-known exact string matching algorithms. For exact string matching, nothing else is needed as none of the skipped suffixes X_{i+j}, $j \in [1..h)$, can have Y as a prefix. Our contribution is to show how to add order comparisons between Y and the suffixes. Checking whether $\mathsf{X}_i < \mathsf{Y}$ needs just one symbol comparison. Comparing the skipped suffixes to Y is based on the following lemma.

Lemma 1. *Let* $\ell = \mathsf{lcp}(\mathsf{X}_i,\mathsf{Y})$ *and* $p = \mathsf{per}(\mathsf{Y}[0..\ell))$. *Then* $\mathsf{X}_{i+j} < \mathsf{Y}$ *iff* $\mathsf{Y}_j < \mathsf{Y}$ *for any* $j \in [1..p)$.

Proof. Fix $j \in [1..p)$. If we had $\mathsf{lcp}(\mathsf{X}_{i+j},\mathsf{Y}) \ge \ell - j$, this would imply that j is a period of $\mathsf{Y}[0..\ell)$, which violates the assumption that $p = \mathsf{per}(\mathsf{Y}[0..\ell))$. Thus $\mathsf{lcp}(\mathsf{X}_{i+j},\mathsf{Y}) < \ell - j$ and the claim follows since $\mathsf{lcp}(\mathsf{X}_{i+j},\mathsf{Y}_j) = \ell - j$. □

In other words, the status of the suffixes in the skipped segment depends only on Y enabling the use of precomputed information.

3 Linear Time and Logarithmic Extra Space

Our first algorithm solves the one-sided counting variant of the string range matching problem, that is, computes the value of $|\mathsf{X}_{[0..n)} \cap [\varepsilon,\mathsf{Y})|$. The two-sided problem needs two calls to the algorithm. The algorithm is built on the exact string matching algorithm of Galil and Seiferas that uses $\mathrm{O}(\log m)$ extra space [9] (or more precisely on the cleaner formulation due to Crochemore and Rytter [4]).

Assume that $k \ge 3$ is an integer constant. A prefix P of Y is called a *k-highly repeating prefix* (k-hrp) of Y if P is primitive and P^k is a prefix of Y. With each k-hrp P of Y we associate the interval of positions $\big[2|\mathsf{P}|, \mathsf{lcp}(\mathsf{Y},\mathsf{Y}_{|\mathsf{P}|})\big)$ called the *scope* of P. Scopes have a number of useful properties:

Lemma 2 ([4]). *A string* Y *of length* m *has* $\mathrm{O}(\log m)$ *scopes and they can be computed in* $\mathrm{O}(m)$ *time and* $\mathrm{O}(\log m)$ *extra space. Two distict scopes do not overlap and for any prefix* $\mathsf{Y}[0..\ell)$ *of* Y *it holds that:*

$$\begin{array}{ll} \mathsf{per}(\mathsf{Y}[0..\ell)) = b/2 & \text{if } \ell \in [b,e) \text{ and } [b,e) \text{ is a scope of some } k\text{-hrp of } \mathsf{Y}\text{; and} \\ \mathsf{per}(\mathsf{Y}[0..\ell)) > \ell/k & \text{otherwise.} \end{array}$$

Using the scopes, we can compute the skip value h in the generic matching step as either $b/2$ or $\lfloor \ell/k \rfloor + 1$, which is sufficient for exact string matching [4]. For string range matching, we need additional precomputed information. First, for each k-hrp P of Y we precompute, in addition to its scope $[b,e)$, the value $c = |\mathsf{Y}_{[0..|\mathsf{P}|)} \cap [\varepsilon, \mathsf{Y})|$. We store all triples (b,e,c) in a sorted list $\mathcal{S}_p$, which allows us to count all suffixes that are skipped-over whenever $h = \mathsf{per}(\mathsf{Y}[0..\ell))$. If the match length ℓ does not belong to a scope, the Galil–Seiferas algorithm would use a skip value $h = \lfloor \ell/k \rfloor + 1$. Our strategy in this case is to do a shorter (by a constant factor) skip h for which we know the value $|\mathsf{Y}_{[0..h)} \cap [\varepsilon, \mathsf{Y})|$. To be able to do that, we will precompute a second list $\mathcal{S}_n = ((b_1, c_1), \ldots, (b_t, c_t))$ of length $\mathrm{O}(\log m)$ that satisfies: $c_i = |\mathsf{Y}_{[0..b_i)} \cap [\varepsilon, \mathsf{Y})|$ for $i \in [1..t]$ and $2b_i \le b_{i+1} < 4b_i$ for $i \in [1..t-1]$. Moreover $4b_t > m$.

Pseudo-code for the algorithm is given in Fig. 1. The algorithm precomputing lists $\mathcal{S}_p$ and $\mathcal{S}_n$ is given in Fig. 2. It is essentially the same as in Fig. 1, except we are now matching the pattern against itself.

Algorithm `OneSidedStringRangeCounting`(X, Y)

Input: strings $\mathsf{X}[0..n)$, $\mathsf{Y}[0..m)$.
Output: $|\mathsf{X}_{[0..n)} \cap [\varepsilon, \mathsf{Y})|$.

```
1:  (S_p, S_n) ← Precompute(Y)
2:  count ← i ← ℓ ← 0
3:  while i < n do        // Invariant: count = |X_[0..i) ∩ [ε, Y)|
4:      while i + ℓ < n and ℓ < m and X[i + ℓ] = Y[ℓ] do ℓ ← ℓ + 1
5:      (b, e, c) ← contains(S_p, ℓ)
6:      if ℓ < m and (i + ℓ = n or X[i + ℓ] < Y[ℓ]) then
7:          count ← count + 1
8:      if b ≠ 0 then     // per(Y[0..ℓ)) = b/2
            // c = |Y_[1..b/2) ∩ [ε, Y)| = |X_[i+1..i+b/2) ∩ [ε, Y)|
9:          count ← count + c
10:         i ← i + b/2; ℓ ← ℓ − b/2
11:     else    // per(Y[0..ℓ)) > ℓ/k
12:         (b, c) ← pred(S_n, ⌊ℓ/k⌋ + 1)    // (⌊ℓ/k⌋ + 1)/4 < b
            // c = |Y_[1..b) ∩ [ε, Y)| = |X_[i+1..i+b) ∩ [ε, Y)|
13:         count ← count + c
14:         i ← i + b; ℓ ← 0
15: return count
```

Fig. 1. String range counting in linear time and logarithmic space.

Algorithm `Precompute(Y)`
Input: a string $\mathsf{Y}[0..m)$.
Output: lists $\mathcal{S}_p$ and $\mathcal{S}_n$.
1: $\mathcal{S}_p \leftarrow ()$; $\mathcal{S}_n \leftarrow ((1,0))$
2: $i \leftarrow last \leftarrow 1$; $\ell \leftarrow count \leftarrow 0$
3: **while** $i < m$ **do** // Invariant: $count = |\mathsf{Y}_{[0..i)} \cap [\varepsilon, \mathsf{Y})|$
4: **while** $i + \ell < m$ **and** $\mathsf{Y}[i+\ell] = \mathsf{Y}[\ell]$ **do** $\ell \leftarrow \ell + 1$
5: $(b, e, c) \leftarrow \texttt{contains}(\mathcal{S}_p, \ell)$
6: **if** $k \cdot i \leq i + \ell$ **and** $b = 0$ **then**
7: $(b, e, c) \leftarrow (2i, i + \ell, count)$
8: $\texttt{add}(\mathcal{S}_p, (b, e, c))$ // $\mathsf{Y}[0..i)$ is a new k-hrp
9: **if** $2 \cdot last \leq i$ **then**
10: $\texttt{add}(\mathcal{S}_n, (i, count))$; $last \leftarrow i$
11: **if** $i + \ell = m$ **or** $\mathsf{Y}[i+\ell] < \mathsf{Y}[\ell]$ **then** $count \leftarrow count + 1$
12: **if** $b \neq 0$ **then**
13: $count \leftarrow count + c$
14: $i \leftarrow i + b/2$; $\ell \leftarrow \ell - b/2$
15: **else**
16: $(b, c) \leftarrow \texttt{pred}(\mathcal{S}_n, \lfloor \ell/k \rfloor + 1)$
17: $count \leftarrow count + c$
18: $i \leftarrow i + b$; $\ell \leftarrow 0$
19: **return** $(\mathcal{S}_p, \mathcal{S}_n)$

Fig. 2. Computation of lists $\mathcal{S}_p$ and $\mathcal{S}_n$.

The algorithms use the following additional procedures: $\texttt{add}(\mathsf{L}, x)$ adds an element x at the end of a list L, $\texttt{pred}(\mathcal{S}_n, x)$ returns $(b, c) \in \mathcal{S}_n$, $b \leq x$ with the largest b, and $\texttt{contains}(\mathcal{S}_p, x)$ returns $(b, e, c) \in \mathcal{S}_p$ such that $b \leq x < e$ or $(0, \cdot, \cdot)$ if no such triple belongs to $\mathcal{S}_p$.

Theorem 1. *The algorithm in Figure 1 solves the counting version of the string range matching problem in linear time and* $\mathrm{O}(\log(m_1 + m_2))$ *extra space.*

Proof. The correctness of the algorithm follows from Lemmas 1 and 2 as explained above and detailed in the invariants included in the code. The time complexity is linear by the same arguments as in [4]. In particular, the shifts made by our algorithm may be shorter but only by a constant factor. □

4 $\mathrm{O}(n \log(m_1 + m_2))$ Time and Constant Extra Space

In this section, we first describe an algorithm solving the string range matching problem in linear time and constant extra space for the special case, where the query range is of the form $[\mathsf{Y}[0..r), \mathsf{Y})$ with $\lfloor 2m/3 \rfloor \leq r < m = |\mathsf{Y}|$. We will then show how the general string range matching problem can be solved using $\mathrm{O}(\log m)$ calls to the special case algorithm.

The backbone of the algorithm is the exact string matching algorithm of Crochemore [2] (see [11] for a simplified version, which we use here). Similarly

to the Galil–Seiferas algorithm, it computes the skip value h in the generic step either as $h = \mathsf{per}(\mathsf{Y}[0..\ell)) \leq \ell/3$ or $h = \lfloor \ell/3 \rfloor + 1 \leq \mathsf{per}(\mathsf{Y}[0..\ell))$, but now using only O(1) extra space. Our addition is two precomputed values $q = \mathsf{per}(\mathsf{Y}[0..r))$ and $e = q + \mathsf{lcp}(\mathsf{Y}, \mathsf{Y}_q)$, i.e., $\mathsf{Y}[0..e)$ is the longest prefix of Y having period q. Both values can be computed in linear time using constant extra space [2]. Determining of the status of the skipped suffixes is based on Lemma 1 and the following result. The pseudo-code is given in Figure 3.

Lemma 3. *Let $h \leq \lfloor m/3 \rfloor + 1$ be a positive integer and $g = \min(\lceil \frac{h-q}{q} \rceil, \lfloor \frac{e-r}{q} \rfloor)$. Then:*

$$\{j \in [0..h) \mid \mathsf{Y}[0..r) \leq \mathsf{Y}_j < \mathsf{Y}\} = \begin{cases} \{q, 2q, \ldots, gq\} & \text{if } \mathsf{Y}_q < \mathsf{Y} \\ \emptyset & \text{otherwise} \end{cases}$$

Proof. First note that $\mathsf{Y}[0..r) \leq \mathsf{Y}_j < \mathsf{Y}$ iff $\mathsf{Y}_j < \mathsf{Y}$ and $\mathsf{lcp}(\mathsf{Y}_j, \mathsf{Y}) \geq r$. If $\mathsf{lcp}(\mathsf{Y}_j, \mathsf{Y}) \geq r$, then j is a period of $\mathsf{Y}[0..r)$. Since $q = \mathsf{per}(\mathsf{Y}[0..r))$ and $j + q \leq 2j \leq 2\lfloor m/3 \rfloor \leq r$, the Weak Periodicity Lemma implies that j must be a multiple of q. Thus let $j = kq$ for some integer k such that $kq < h \leq e$. Then $\mathsf{lcp}(\mathsf{Y}_j, \mathsf{Y}) = e - j$ and $\mathsf{Y}_j < \mathsf{Y}$ iff $\mathsf{Y}_q < \mathsf{Y}$. Thus, if $\mathsf{Y}_q \geq \mathsf{Y}$, the set in the lemma is empty. Otherwise, the set includes all $j = kq$ such that $kq < h$ and $\mathsf{lcp}(\mathsf{Y}_j, \mathsf{Y}) \geq r$, which give the limits $k \leq \lceil (h-q)/q \rceil$ and $k \leq \lfloor (e-r)/q \rfloor$, respectively. □

Algorithm `RestrictedOneSidedStringRangeReporting`$(\mathsf{X}, \mathsf{Y}, r)$

Input: strings $\mathsf{X}[0..n)$, $\mathsf{Y}[0..m)$ and an integer $r \geq \lfloor 2m/3 \rfloor$.
Output: the set $\mathcal{R} = \{j \in [0..n) \mid \mathsf{Y}[0..r) \leq \mathsf{X}_j < \mathsf{Y}\}$.

```
1:  (q, e) ← Precompute(Y, r)
2:  R ← ∅
3:  i ← ℓ ← s ← p ← 0
4:  while i < n do        // Invariant: R = {j ∈ [0..i) | Y[0..r) ≤ X_j < Y}
5:      while i + ℓ < n and ℓ < m and X[i + ℓ] = Y[ℓ] do
6:          UpdateMS(Y, ℓ, s, p)    // Increments ℓ by one
7:      if r ≤ ℓ < m and (i + ℓ = n or X[i + ℓ] < Y[ℓ]) then R ← R ∪ {i}
8:      if 0 < p ≤ ℓ/3 and Y[0..s) = Y[p..p + s) then    // per(Y[0..ℓ)) = p
9:          h ← p; ℓ ← ℓ − p
10:     else    // per(Y[0..ℓ)) > ℓ/3
11:         h ← ⌊ℓ/3⌋ + 1; (ℓ, s, p) ← (0, 0, 0)
12:     if e < m and Y[e] < Y[e mod q] then    // Y_q < Y
13:         g ← min(⌈(h − q)/q⌉, ⌊(e − r)/q⌋)
14:         R ← R ∪ {i + q, i + 2q, ..., i + gq}
15:     i ← i + h
16: return R
```

Fig. 3. Reporting suffixes $\mathsf{X}_j \in [\mathsf{Y}[0..r), \mathsf{Y})$ in linear time and constant extra space. Due to lack of space, we omit the code for `UpdateMS`(Y, ℓ, s, p), which can be found in [11], and the precomputation of q and e.

Let us now consider more general forms of queries. First, consider the query $[\mathsf{Y}[0..\ell), \mathsf{Y})$ with $\ell < r = \lfloor 2|\mathsf{Y}|/3 \rfloor$. We can break the query into two distinct subqueries: $[\mathsf{Y}[0..\ell), \mathsf{Y}) = [\mathsf{Y}[0..\ell), \mathsf{Y}[0..r)) \cup [\mathsf{Y}[0..r), \mathsf{Y})$. This leads to a tail-recursive algorithm that makes $\mathrm{O}(\log(|\mathsf{Y}|/\ell))$ calls to the algorithm in Fig. 3 and thus runs in $\mathrm{O}(n \log(|\mathsf{Y}|/\ell))$ time still using only constant extra space. An alternative formulation for the query $[\mathsf{Y}[0..\ell), \mathsf{Y})$ is to compute the set $\{\mathsf{X}_j \mid \mathsf{X}_j < \mathsf{Y} \text{ and } \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) \geq \ell\}$. By a symmetrical procedure, we can also compute the set $\{\mathsf{X}_j \mid \mathsf{Y} \leq \mathsf{X}_j \text{ and } \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) \geq \ell\}$ in $\mathrm{O}(n \log(|\mathsf{Y}|/\ell))$ time and constant extra space.

Consider now the fully general query $[\mathsf{Y}, \mathsf{Z})$ and let $\ell = \mathsf{lcp}(\mathsf{Y}, \mathsf{Z})$. We can partition the result set $\mathsf{X}_{[0..n)} \cap [\mathsf{Y}, \mathsf{Z})$ into three disjoint sets $\mathcal{R}_1 = \{\mathsf{X}_j \mid \mathsf{X}_j < \mathsf{Z} \text{ and } \mathsf{lcp}(\mathsf{X}_j, \mathsf{Z}) > \ell\}$, $\mathcal{R}_2 = \{\mathsf{X}_j \mid \mathsf{Y} \leq \mathsf{X}_j \text{ and } \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) > \ell\}$ and $\mathcal{R}_3 = \{\mathsf{X}_j \mid \mathsf{Y} \leq \mathsf{X}_j < \mathsf{Z} \text{ and } \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) = \mathsf{lcp}(\mathsf{X}_j, \mathsf{Z}) = \ell\}$. The sets $\mathcal{R}_1$ and $\mathcal{R}_2$ can be computed as described above. $\mathcal{R}_3$ can be found using any time–space optimal exact string matching algorithm (e.g. [2]). We have proven the following result.

Theorem 2. *It is possible to solve the string range matching problem using constant extra space in* $\mathrm{O}(n \log((m_1 + m_2)/(1 + \mathsf{lcp}(\mathsf{Y}, \mathsf{Z}))))$ *time.*

The worst case time complexity is $\mathrm{O}(n \log(m_1 + m_2))$. If $\mathsf{lcp}(\mathsf{Y}, \mathsf{Z}) \geq \epsilon(m_1 + m_2)$ for some constant $\epsilon > 0$, the algorithm runs in linear time.

5 Linear Time and Constant Extra Space

In this section, we describe two algorithms for string range matching that run in linear time and constant extra space by taking advantage of the space reserved for input or output, which we do not count as extra space. Both algorithms are based on Crochemore's string matching algorithm.

We will describe the algorithms for one-sided queries $[\varepsilon, \mathsf{Y})$. Since these algorithms produce their output in a sequential order, we can answer two-sided queries by running two one-sided queries in parallel: $[\mathsf{Y}, \mathsf{Z}) = [\varepsilon, \mathsf{Z}) \setminus [\varepsilon, \mathsf{Y})$. To ease the description of the algorithms, we assume that the output is produced as a bitvector $\mathsf{B}[0..n)$ satisfying, for all $i \in [0..n)$, $\mathsf{B}[i] = 1$ iff $\mathsf{X}_i < \mathsf{Y}$. We will later describe how to produce output in other formats.

Copying Output. Consider the actions of Crochemore's algorithm as it compares the pattern Y against the text suffix X_i. It computes $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ and then shifts the pattern forward by h positions. With one character comparison we can determine $\mathsf{B}[i]$, so the problem now is computing the output for the skipped-over positions $\mathsf{B}[i + 1..i + h)$.

Let $\ell_{\max}$ be the longest previously found match, which occurred at position j, i.e., $\ell_{\max} = \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y})$. If $\ell \leq \ell_{\max}$, then by Lemma 1 we have $\mathsf{B}[i+1..i+h) = \mathsf{B}[j+1..j+h)$. Thus, if we have access to the previous output, we can simply copy the missing output. Note that this works even if the two intervals $[i + 1..i + h)$ and $[j + 1..j + h)$ overlap.

If $\ell > \ell_{\max}$, we cannot obtain all the missing values by copying. In this case, we set ℓ to $\ell_{\max}$ (after computing $\mathsf{B}[i]$) and continue as if the match had been a shorter one. This shorter match length leads to a shorter shift and allows copying the output for all the skipped-over positions. Shortening the match length does not violate the correctness of the algorithm as long as the state of algorithm is adjusted accordingly. The full match length becomes the new value of $\ell_{\max}$, i.e., we are actually swapping the values of ℓ and $\ell_{\max}$.

Algorithm `OneSidedStringRangeReporting`(X, Y)

Input: strings $\mathsf{X}[0..n)$ and $\mathsf{Y}[0..m)$.
Output: bitvector $\mathsf{B}[0..n)$, where $\mathsf{B}[i] = 1$ iff $\mathsf{X}[i..n) < \mathsf{Y}$.

```
1:  B ← (0, 0, ..., 0)
2:  i ← ℓ ← p ← s ← 0
3:  i_max ← ℓ_max ← p_max ← s_max ← 0
4:  while i < n do
5:      while i + ℓ < n and ℓ < m and X[i + ℓ] = Y[ℓ] do
6:          (ℓ, s, p) ← UpdateMS(Y, ℓ, s, p)    // Increments ℓ by one
        // ℓ = lcp(X_i, Y)
7:      if ℓ < m and (i + ℓ = n or X[i + ℓ] < Y[ℓ]) then B[i] = 1
8:      j ← i_max
9:      if ℓ > ℓ_max then
10:         swap (ℓ, s, p) and (ℓ_max, s_max, p_max)
11:         i_max ← i
12:     if 0 < p ≤ ℓ/3 and Y[0..s) = Y[p..p + s) then    // per(Y[0..ℓ)) = p
13:         B[i + 1..i + p) ← B[j + 1..j + p)
14:         i ← i + p
15:         ℓ ← ℓ − p
16:     else    // per(Y[0..ℓ)) > ℓ/3
17:         h ← ⌊ℓ/3⌋ + 1
18:         B[i + 1..i + h) ← B[j + 1..j + h)
19:         i ← i + h
20:         (ℓ, s, p) ← (0, 0, 0)
21: return B
```

Fig. 4. Linear time, constant extra space algorithm.

The pseudocode for the algorithm is given in Fig. 4. The key changes to the original algorithm in [11, Figure 2] are the computation of B on lines 7, 13 and 18, and the swap of ℓ and $\ell_{\max}$ on line 10. The two other variables, s and p, swapped together with ℓ represent the rest of the state of the algorithm that needs to change when ℓ changes. We refer to [11] for their explanation. Here it suffices to know that they are functions of ℓ (and Y).

Theorem 3. *The algorithm in Fig. 4 solves the string range reporting problem in linear time using constant extra space.*

Proof. The correctness of the algorithm has been established above, and the algorithm clearly uses only constant extra space. In [11], it is shown that the time spent in any round of the original algorithm is proportional to the increase of the value $6i + \ell + s$ during that round. Because of the swap on line 11, this might not hold for the modified algorithm, but this can be corrected by using the value $6i + \ell + s + \ell_{\max} + s_{\max}$ instead. The copying of the output does not add more than $O(n)$ time. Thus the time complexity is still $O(n + m)$. □

The algorithm can be easily modified to handle other output formats such as a sequence of starting positions. On the other hand, the algorithm cannot be used if the output is streamed or compressed in such a way that it cannot be quickly decompressed from an arbitrary position. Also, the counting version of the problem cannot be solved this way without extra space.

Overwriting Input. Suppose we have separate storage space of m bits and consider the following variant of the algorithm in Fig. 4. Whenever $i_{\max}$ changes we write $B[i_{\max} + 1..i_{\max} + m)$ to our separate storage at the same time as it is being written to the output. Whenever we need to copy a part of the output, we can copy from our separate storage instead of from the output itself. Now the algorithm never needs to access the output, and so can stream the output, encode it in any way, or just count the number of occurrences without producing any output until the end.

Let $\hat{\ell}$ be the length of the longest prefix of Y that also occurs elsewhere in Y. We will use the space occupied by $\mathsf{Y}[0..\hat{\ell})$ as a separate storage of at least $\hat{\ell}$ bits. Any access to the characters of $\mathsf{Y}[0..\hat{\ell})$ are redirected to that other occurrence. Clearly this can be made to work even if that other occurrence overlaps $\mathsf{Y}[0..\hat{\ell})$. The value $\hat{\ell}$ is easily computed in $O(m)$ time using $O(1)$ extra space.

The largest number of bits the algorithm may need to copy from the separate storage at a given stage is at most $\ell_{\max}/3$, since the longest possible shift is $\lfloor \ell_{\max}/3 \rfloor + 1$. Thus as long as $\ell_{\max} \leq 3\hat{\ell}$, the algorithm can operate as described.

Now consider a round i of the algorithm with $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y}) > 3\hat{\ell}$. We call this a *long match*. In this case, the algorithm behaves as it would for a shorter match except the value of $\ell_{\max}$ (and the associated $s_{\max}$ and $p_{\max}$) is not modified. Thus $\ell_{\max}$ never grows too large for the separate storage. The extra time spent in round i because of this is at most $O(\ell)$, and the total extra time is bounded by the total length of the long matches. Thus the following lemma shows that the time complexity of the algorithm is still $O(n)$.

Lemma 4. *The total length of all long matches is at most* $1.5n$.

Proof. Consider a long match of length ℓ at position i, i.e., $\mathsf{X}[i..i + \ell) = \mathsf{Y}[0..\ell)$, and assume that the next long match starts at $j > i$ and has length ℓ'. Then $\min\{\ell, \ell'\} > 3\hat{\ell}$. First observe that we must have $j + \ell' > i + \ell$. Otherwise, $\mathsf{Y}[0..\ell')$ has another occurrence as $\mathsf{Y}[j - i..j - i + \ell')$ and $\hat{\ell} \geq \ell'$. Furthermore, we must have $j > i + \ell - \lceil \ell/3 \rceil$. Otherwise, $\mathsf{Y}[0..\lceil \ell/3 \rceil)$ has another occurrence as $\mathsf{Y}[j - i..j - i + \lceil \ell/3 \rceil)$ and $3\hat{\ell} \geq \ell$. Thus at most one third of a long match can be overlapped by a later long match, which proves the lemma. □

We have proven the following.

Theorem 4. *The string range matching problem can be solved in linear time and constant extra space if each character of the range boundary strings occupies at least one bit and that bit can be overwritten by the algorithm.*

Note that the algorithm can restore the overwritten part of Y at the end as long as equal characters are always represented by identical bit patterns. The restoration might not be possible in some cases, for example, when upper and lower case letters are considered equal in comparisons.

6 Concluding Remarks

This article is the first to directly consider the string range matching problem. Many interesting avenues for future work remain, but perhaps the most challenging open problem is to establish if the algorithms described in this paper for the general setting (read-only input, inaccessible output) are optimal, or if linear-time constant extra-space methods indeed exist.

References

1. M. Burrows and D.J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
2. M. Crochemore. String-matching on ordered alphabets. *Theor. Comp. Sci.*, 92:33–47, 1992.
3. M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.
4. M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995.
5. S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comp. Surv.*, 45(2):13, 2013.
6. P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.
7. N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16(1):109–114, 1965.
8. Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comp. Sys. Sci.*, 26:280–294, 1983.
9. Z. Galil and J. I. Seiferas. Saving space in fast string-matching. *SIAM J. Comp.*, 9(2):417–438, 1980.
10. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theor. Comp. Sci.*, 387(3):249–257, 2007.
11. J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Crochemore's string matching algorithm: Simplification, extensions, applications. In *Proc. PSC 2013*, pages 168–175, Czech Technical University, 2013.
12. D. Knuth, J. H. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6(2):323–350, 1977.
13. U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.