Fast and Simple Jumbled Indexing for Binary Run-Length Encoded Strings*

Luís Cunha[1], Simone Dantas[2], Travis Gagie[3], Roland Wittler[4], Luis Kowada[5], and Jens Stoye[6]

1 **Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil; and Universidade Federal Fluminense, Niterói, Brazil**
lfignacio@cos.ufrj.br
2 **Universidade Federal Fluminense, Niterói, Brazil**
sdantas@im.uff.br
3 **CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Santiago, Chile; and School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile**
travis.gagie@gmail.com
4 **Universität Bielefeld, Bielefeld, Germany**
roland.wittler@uni-bielefeld.de
5 **Universidade Federal Fluminense, Niterói, Brazil**
luis@ic.uff.br
6 **Universidade Federal Fluminense, Niterói, Brazil; and Universität Bielefeld, Bielefeld, Germany**
jens.stoye@uni-bielefeld.de

Abstract

Important papers have appeared recently on the problem of indexing binary strings for jumbled pattern matching, and further lowering the time bounds in terms of the input size would now be a breakthrough with broad implications. We can still make progress on the problem, however, by considering other natural parameters. Badkobeh et al. (IPL, 2013) and Amir et al. (TCS, 2016) gave algorithms that index a binary string in $O(n + \rho^2 \log \rho)$ time, where n is the length and ρ is the number of runs, and Giaquinta and Grabowski (IPL, 2013) gave one that runs in $O(n + \rho^2)$ time. In this paper we propose a new and very simple algorithm that also runs in $O(n + \rho^2)$ time and can be extended either so that the index returns the position of a match (if there is one), or so that the algorithm uses only $O(n)$ bits of space instead of $O(n)$ words.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases string algorithms, indexing, jumbled pattern matching, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.19

1 Introduction

Since its introduction at the 2009 Prague Stringology Conference [6, 8], the problem of indexed binary jumbled pattern matching has been discussed in many top conferences and

* This work was partially supported by CAPES, CNPq and FAPERJ.

28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017).
Editors: Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter; Article No. 19; pp. 19:1–19:9

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

journals. It asks us to preprocess a binary string such that later, given a number of 0s and a number of 1s, we can quickly report whether there exists a substring with those numbers of 0s and 1s and, optionally, return the position of one such substring or possibly even all of them. The naïve preprocessing algorithm takes quadratic time but researchers have reduced that bound to $O(n^2/\log n)$ [5, 16], $O(n^2/\log^2 n)$ [17], $O(n^2/2^{\Omega(\sqrt{\log n/\log\log n})})$ [4, 14] and finally $O(n^{1.859})$ with randomization or $O(n^{1.864})$ without [7].

Researchers have also looked at indexing for approximate matching [9, 10], indexed jumbled pattern matching over larger alphabets [2, 15], indexing labelled trees and other structures [9, 11, 12], and how to index faster when the (binary) input string is compressible. Gagie et al. [12] gave an algorithm that runs in $O(g^{2/3}n^{4/3})$ when the input is represented as a straight-line program with g rules, and Badkobeh et al. [3] gave one that runs in $O(n+\rho^2\log\rho)$ time when the input consists of ρ runs, i.e., maximal unary substrings (we will denote later as ρ the number of maximal substrings of 1s, for convenience). Giaquinta and Grabowski [13] gave two algorithms: one runs in $O(\rho^2\log k + n/k)$ time, where k is a parameter, and produces an index that uses $O(n/k)$ extra space and answers queries in $O(\log k)$ time; the other runs in $O(n^2\log^2(w)/w)$ time, where w is the size of a machine word. Amir et al. [1] gave an algorithm that runs in $O(\rho^2\log\rho)$ time when the input is a run-length encoded binary string, or $O(n+\rho^2\log\rho)$ time when it is a plain binary string; it builds an index that takes $O(\rho^2)$ words and answers queries in $O(\log\rho)$ time, however. Very recently, Sugimoto et al. [19] considered the related problems of finding Abelian squares, Abelian periods and longest common Abelian factors, also on run-length encoded strings.

We first review some preliminary notions in Section 2. We present our main result in Section 3: a new and very simple indexing algorithm that runs in $O(n+\rho^2)$ time, which matches Giaquinta and Grabowski's algorithm with the parameter $k=1$ and is thus tied as the fastest known when $\rho=\Omega(n^{0.5})\cap o(n^{0.932})$ and the smallest straight-line program for the input has $\omega(\rho^3/n^2)$ rules. For an input string of up to ten million bits, for example, if the average run-length is three or more then $\rho < n^{0.932}$. While Giaquinta and Grabowski found an efficient way to construct the Corner Index of Badkobeh et al. [3], our algorithm constructs a more direct index and takes only 17 lines of pseudocode, making it a promising starting point for investigating other possible algorithmic features. In Section 4, for example, we show how to extend our algorithm to store information that lets us report the position of a match (if there is one). Finally, in Section 5, we show how we can alternatively adapt it to use only $O(n)$ bits of space.

2 Preliminaries

Consider a string $s\in\{0,1\}^n$. We denote by $s[i\cdots j]$ the substring of s consisting of the ith through jth characters, for $1\le i\le j\le n$; if $i=j$, we can also write simply $s[i]$. Cicalese et al. [6, 8] observed that, if we slide a window of length k over s, the number of 1s in the window can change by at most 1 at each step. It follows that if $s[i\cdots i+k-1]$ contains x copies of 1 and $s[j\cdots j+k-1]$ contains z copies of 1 with $i\le j$ then, for y between x and z (notice x could be smaller than, larger than, or equal to z), there is a substring of length k in $s[i\cdots j+k-1]$ with exactly y copies of 1. This immediately implies the following theorem:

▶ **Theorem 1.** *Let x and z be the minimum and maximum numbers of 1s in any substring of length k. There is a substring of length k with y copies of 1 if and only if $x\le y\le z$.*

By Theorem 1, if we compute and store, for $1\le k\le n$, the minimum and maximum numbers of 1s in a substring of s of length k then later, given a number of 0s and a number

of 1s, we can report in constant time whether there exists a substring with that many 0s and 1s. For example, if $s = 010101110011$ then, as k goes from 1 to $n = 12$, the minimum and maximum numbers of 1s are $0, 0, 1, 2, 2, 3, 4, 4, 5, 5, 6, 7$ and $1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7$, respectively. Since the fifth numbers in these lists are 2 and 4, we know that there are substrings of length 5 with exactly 2, 3 and 4 copies of 1, but none with 0, 1 or 5 (or more than 5, obviously).

Cicalese et al. [9] noted that, if we also store the positions of the substrings with the minimum and maximum numbers of 1s and a bitvector for s that supports constant time rank queries, then via binary search in $O(\log n)$ time we can find an example of a substring with any desired numbers of 0s and 1s, called a *witness* if such a substring exists. (The query $\text{rank}(i)$ returns the number of 1s in $s[1 \cdots i]$; see, e.g., [18] for more details of rank queries on bitvectors.) For example, suppose we want to find a substring of length 5 with exactly 3 copies of 1 in our example string s. We have stored that there are substrings of length 5 with 2 and 4 copies of 1 starting at positions 1 and 4, respectively, so we know there is a substring of length 5 with exactly 3 copies of 1 starting in $s[1 \cdots 4]$. We choose $\lfloor (1 + 4)/2 \rfloor = 2$ and check how many 1s there are in $s[2 \cdots 2 + 5 - 1 = 6]$ via two rank queries. In this case, the answer is 3, so we have found a witness in one step; otherwise, we would know there is a witness starting in $s[3 \cdots 4]$ and we would recurse on that interval.

The same authors noted that in each step, the lists of minimum and maximum numbers can only stay the same or increment, so we can represent each list as a bitvector of length n and support access to it using rank queries. For example, the bitvector for the list of minimum numbers in our example is 001101101011, so $\text{rank}(i)$ returns the ith number in the list. Since an n-bit bitvector takes $O(n)$ bits of space, it follows that we can store our index in $O(n)$ bits and still support constant-time queries, if we do not want a witness. We note, however, that even though the input s takes n bits and the resulting index takes $O(n)$ bits, all previous constructions have used $\Omega(n)$ *words* of workspace in the worst case.

A *run* in s is a maximal unary substring and the run-length encoding $\text{rle}(s)$ is obtained by replacing each run by a copy of the character it contains and its length. Although ρ is usually used to denote the number of runs, for convenience, we use it to denote only the number of runs of 1s – about half its normal value for binary strings – and consider s to begin and end with (possibly empty) runs of 0s. For example, for our example string the run-length encoding is $0^1 1^1 0^1 1^1 0^1 1^3 0^2 1^2 0^0$ and $\rho = 4$ (instead of 9). We denote the lengths of the runs of 0s and 1s as $z[0], \ldots, z[\rho]$ and $o[1], \ldots, o[\rho]$, respectively.

3 Basic Indexing

Since finding substrings with the minimum numbers of 1s is symmetric to finding substrings with the maximum numbers of 1s (e.g., by taking the complement of the string), we describe how, given a binary run-length encoded string $s[1 \cdots n]$, we can build a table $T[1 \cdots n]$ such that $T[k] = f(k)$, where $f(k)$ denotes the maximum number of 1s in a substring of s of length k.

The complete pseudo-code of our algorithm – only 17 lines – is shown as Algorithm 1. The starting point of our explanation and proof of correctness is the observation that, if the bit immediately to the left of a substring is a 1, we can shift the substring one bit left without decreasing the number of 1s; if the first bit of the substring is a 0, then we can shift the substring one bit right (shortening it on the right if necessary) without decreasing the number of 1s. It follows that, for $1 \leq k \leq n$, there is a substring of length at most k containing $f(k)$ copies of 1 and starting at the beginning of a run of 1s. Since we can remove

Algorithm 1: Building the index table T of string s.

```
for i = 1, ..., n do
    T[i] = 0
for i = 1, ..., ρ do
    ones = o[i]
    zeros = 0
    T[ones] = ones
    for j = i + 1, ..., ρ do
        ones += o[j]
        zeros += z[j − 1]
        if ones > T[ones + zeros] then
            T[ones + zeros] = ones
for i = n − 1, ..., 1 do
    if T[i] < T[i + 1] − 1 then
        T[i] = T[i + 1] − 1
for i = 2, ..., n do
    if T[i] < T[i − 1] then
        T[i] = T[i − 1]
```

any trailing 0s from such a substring also without changing the number of 1s, there is such a substring that also ends in a run of 1s. Therefore we have the following lemma:

▶ **Lemma 2.** *For $1 \leq k \leq n$, there is a substring of length at most k containing $f(k)$ copies of 1, starting at the beginning of a run of 1s and ending in a run of 1s.*

Applying Lemma 2 immediately yields an $O(n\rho)$-time algorithm: set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ in a run of 1s, set $T[j-i+1] = \max(T[j-i+1], s[i] + \cdots + s[j])$; finally, because f is non-decreasing, make a pass over T from $T[2]$ to $T[n]$ setting each $T[i] = \max(T[i], T[i-1])$. Computing the number $s[i] + \cdots + s[j]$ of 1s in a substring $s[i \ldots j]$ starting at the beginning of a run of 1s and ending in a run of 1s is easy to do from the run-length encoding in amortized constant time.

To speed this preliminary algorithm up to run in $O(n + \rho^2)$ time, we first observe that, if ℓ is the length of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1, and $d > \ell$ is the length of a substring starting at the beginning of a run of 1s and ending at the end of a run of 1s, then $f(\ell) \geq f(d) - d + \ell$. (In fact this is true for any ℓ and $d \geq \ell$, simply because $f(x+1) \leq f(x) + 1$ for all x.) We then observe that, for some such d, we have $f(\ell) = f(d) - d + \ell$. To see why, consider any substring $s[i \cdots j]$ of length ℓ starting at the beginning of a run of 1s, ending within a run of 1s and containing $f(\ell)$ copies of 1: let d be the length of the substring starting at $s[i]$ and ending at the end of the run of 1s containing $s[i + \ell - 1]$, so $f(\ell) = f(d) - d + \ell$.

▶ **Lemma 3.** *If ℓ is the length of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1, and $d > \ell$ is the length of a substring starting at the beginning of a run of 1s and ending at the end of a run of 1s, then $f(\ell) \geq f(d) - d + \ell$. Furthermore, for some such d, we have $f(\ell) = f(d) - d + \ell$.*

With Lemma 3, we can compute the number $s[i] + \cdots + s[j]$ of 1s in each substring $s[i \ldots j]$ starting at the beginning of a run of 1s and ending in a run of 1s, in a total of $O(n + \rho^2)$ time:

again, set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ at the end of a run of 1s, set $T[j-i+1] = \max(T[j-i+1], s[i]+\cdots+s[j])$; make a pass over T from $T[n-1]$ to $T[1]$ setting each $T[i] = \max(T[i], T[i+1]-1)$. Computing the number $s[i]+\cdots+s[j]$ of 1s in a substring $s[i \cdots j]$ starting at the beginning of a run of 1s and ending at the end of a run of 1s is again easy to do from the run-length encoding in amortized constant time.

Combining Lemmas 2 and 3, we have a complete algorithm for computing T in $O(n+\rho^2)$ time: set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ at the end of a run of 1s, set $T[j-i+1] = \max(T[j-i+1], s[i]+\cdots+s[j])$; make a pass over T from $T[n-1]$ to $T[1]$ setting each $T[i] = \max(T[i], T[i+1]-1)$ (which sets $T[\ell]$ correctly for every length ℓ of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1); and make a pass over T from $T[2]$ to $T[n]$ setting each $T[i] = \max(T[i], T[i-1])$ (which sets every entry in T correctly). Once we have T, we can convert it into a bitvector in $O(n)$ time. Summarizing our results so far, we have the following theorem, which we adapt in later sections:

▶ **Theorem 4.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-bit index for constant-time jumbled pattern matching in $O(n+\rho^2)$ time.*

Now we examine how our algorithm works on our example $s = 010101110011$. First we set all entries of T to 0, then we loop through the runs of 1s and, for each, loop through the runs of 1s not earlier, computing distance from the start of the first to the end of the second and the number of 1s between those positions. While doing this, we set $T[1] = 1$, the number of 1s from the start to the end of the first run of 1s; $T[3] = 2$, the number of 1s from the start of the first run of 1s to the end of the second run of 1s; $T[7] = 5$, the number of 1s from the start of the first run of 1s to the end of the third run of 1s; $T[11] = 7$, the number of 1s from the start of the first run of 1s to the end of the fourth run of 1s; $T[5] = 4$, the number of 1s from the start of the second run of 1s to the end of the third run; etc. When we have finished this stage, $T = [1, 2, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0]$. We then make a pass over T from right to left, setting each $T[i] = \max(T[i], T[i+1]-1)$. After this stage, $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 0]$. Finally, we make a pass over T from left to right, setting each $T[i] = \max(T[i], T[i-1])$. This fills in $T[12]$ and leaves T correctly computed as $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7]$.

4 Witnessing Index

As described in Section 2, if together with computing the minimum and maximum number of 1s in a substring of length k for $1 \leq k \leq n$, we also store the positions of substrings of length k with those numbers of 1s, and a single bitvector for s, then, together with confirming that s contains a substring with a given number of 0s and 1s (if it does), we can give the starting position of such a substring, still in constant time.

In this section, we show how to modify our algorithm from Section 3 to build also a table $P[1..n]$ such that $P[k]$ is the starting position of a substring of length k containing $f(k)$ copies of 1s. Computing and storing the starting position of a substring of length k with the minimum number of 1s is symmetric.

First, notice that during the first stage of Algorithm 1, whenever we set $T[k] = f(k)$, we have found a substring of length k containing $f(k)$ copies of 1, so we can set $P[k]$ at the same time. Now consider the second stage of the algorithm, in which we make a right-to-left pass over T setting $T[i] = \max(T[i], T[i+1]-1)$ for $1 \leq i \leq n-1$. When we start this stage, for every positive entry in T we have set the corresponding entry in P. Therefore, by

induction, whenever we set $T[i] = T[i+1] - 1$, we have $P[i+1]$ set to the starting position of a substring of length $i+1$ containing $T[i+1]$ copies of 1. The substring of length i starting at $P[i+1]$ contains at least $T[i+1] - 1$ copies of 1, so we can set $P[i] = P[i+1]$. In the last stage of the algorithm, in which we make a left-to-right pass over T, we can almost use the same kind of argument and simply copy P values when we copy T values, except that we must ensure the starting positions we copy are far enough to the left of the end of the string (i.e., that the substrings have the correct lengths). Our modified algorithm is shown as Algorithm 2 – still only 25 lines – and we now have the following theorem:

▶ **Theorem 5.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-word index for constant-time jumbled pattern matching with $O(\log n)$ time witnessing in $O(n + \rho^2)$ time.*

Running our modified algorithm on our example $s = 010101110011$, in the first stage we set $T = [1, 2, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0]$ and, simultaneously, $P = [2, 11, 6, 0, 4, 0, 2, 0, 4, 0, 2, 0]$, where 0 indicates an unset value in P. In the second stage, we set $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 0]$ and $P = [2, 11, 6, 4, 4, 2, 2, 4, 4, 2, 2, 0]$. Finally, in the third stage, we fill in $T[12] = T[11]$, but we cannot just set $P[12] = P[11] = 2$ because $s[2 \cdots n = 12]$ has length only 11, so we set $P[12] = 1$.

5 Reducing Workspace

It is frustrating that both s and the index described in Theorem 4 take $O(n)$ bits, but we use $O(n)$ words to build the index. In this section, we show how to reduce this workspace to $O(n)$ bits also, without increasing the time bound for construction by more than a constant factor.

Suppose we divide T into blocks of size $\lg(n)/2$ and modify our algorithm such that, whenever we set a value $T[i]$, we ensure that each value $T[j]$ in the same block with $j < i$ is at least $T[i] - i + j$ and each value $T[j]$ in the same block with $i < j$ is at least $T[i]$. Since we would eventually set each such $T[j]$ to a value at least as great during the normal execution of the algorithm, this does not change its correctness, apart from perhaps slowing it down by an $O(\log n)$ factor.

For any two consecutive values $T[i]$ and $T[i+1]$ in the same block now, however, we have $T[i] \leq T[i+1] \leq T[i] + 1$. We can thus store each block by storing its first value and a binary string of length $\lg(n)/2$ whose bits indicate where the values in the block increase. Therefore, we need a total of only $O(n)$ bits to store all the blocks.

Notice that, if we increase a value $T[i]$ by more than $\lg(n)/2$, we reset the first value $T[h]$ of the block to be $T[i] - i + h$, set the leading bits of the block to 1s to indicate that the values increase until reaching $T[i]$, and set the later bits of the block to 0s to indicate that the values remain equal to $T[i]$ until the end of the block. Therefore, we can speed the algorithm up to run in $O(n + \rho^2)$ time again, by using a universal table of size $2^{\lg(n)/2} \log^{O(1)} n = o(n^{1/2+\epsilon})$ to decide how to update blocks when we set values in them.

▶ **Theorem 6.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-bit index for constant-time jumbled pattern matching in $O(n + \rho^2)$ time using $O(n)$ bits of workspace.*

In fact, it seems possible to make the algorithm run in $O(n + \rho^2)$ time and $O(n)$ bits of space even without a universal table, using AC0 operations on words that are available on standard architectures.

Algorithm 2: Building the tables T and P for s.

```
1  for i = 1, ..., n do
2      T[i] = 0
3  p = z[0]
4  for i = 1, ..., ρ do
5      ones = o[i]
6      zeros = 0
7      T[ones] = ones
8      P[ones] = p
9      for j = i + 1, ..., ρ do
10         ones += o[j]
11         zeros += z[j − 1]
12         if ones > T[ones + zeros] then
13             T[ones + zeros] = ones
14             P[ones + zeros] = p
15     p += ones[i] + zeros[i]
16 for i = n − 1, ..., 1 do
17     if T[i] < T[i + 1] − 1 then
18         T[i] = T[i + 1] − 1
19         P[i] = P[i + 1]
20 for i = 2, ..., n do
21     if T[i] < T[i − 1] then
22         T[i] = T[i − 1]
23         P[i] = P[i − 1]
24         if P[i] + i > n then
25             P[i] = n − i
```

This workspace reduction makes little sense for a string as small as our example $s = 010101110011$ but, for the sake of argument, suppose we partition our array T for it into three blocks of length 4 each. We keep $T[1]$, $T[5]$ and $T[9]$ stored explicitly and represent the other entries of T implicitly with three 3-bit binary strings B_1, B_2 and B_3. Initially we set $T[1] = T[5] = T[9] = 0$ and $B_1 = B_2 = B_3 = 000$. Recall from Section 3 that we first set $T[1] = 1$, the number of 1s from the start to the end of the first run of 1s. At this point, we do not need to change B_1. We then set $T[3] = 2$ – the number of 1s from the start of the first run of 1s to the end of the second run of 1s – by setting $B_1 = 010$: starting from $T[1] = 1$, this encodes $T[2] = T[1] + 0 = 1$, $T[3] = T[1] + 0 + 1 = 2$ and $T[4] = T[1] + 0 + 1 + 0 = 2$. Next we set $T[7] = 5$ – the number of 1s from the start of the first run of 1s to the end of the third run of 1s – by setting $T[5] = 3$ and $B_2 = 110$: starting from $T[5] = 3$, this encodes $T[6] = T[5] + 1 = 4$, $T[7] = T[5] + 1 + 1 = 5$ and $T[8] = T[5] + 1 + 1 + 0 = 5$. Continuing like this, we set $T[11] = 7$ by setting $T[9] = 5$ and $B_3 = 110$; set $T[5] = 4$ and $B_2 = 010$; etc. When we are finished this stage, $T[1] = 1$, $T[5] = 4$ and $T[9] = 6$, and $B_1 = 110$, $B_2 = 010$ and $B_3 = 010$, encoding $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7]$. In this case the final right-to-left and left-to-right passes have no effect, but there are cases (e.g., when we do not set any values in a certain block) when they are still necessary.

References

1 Amihood Amir, Alberto Apostolico, Tirza Hirst, Gad M. Landau, Noa Lewenstein, and Liat Rozenberg. Algorithms for jumbled indexing, jumbled border and jumbled square on run-length encoded strings. *Theor. Comput. Sci.*, 656:146–159, 2016. doi:10.1016/j.tcs.2016.04.030.

2 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of *LNCS*, pages 114–125. Springer, 2014. doi:10.1007/978-3-662-43948-7_10.

3 Golnaz Badkobeh, Gabriele Fici, Steve Kroon, and Zsuzsanna Lipták. Binary jumbled string matching for highly run-length compressible texts. *Inf. Process. Lett.*, 113(17):604–608, 2013. doi:10.1016/j.ipl.2013.05.007.

4 David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Pătraşcu, and Perouz Taslakian. Necklaces, convolutions, and $X + Y$. *Algorithmica*, 69(2):294–314, 2014. doi:10.1007/s00453-012-9734-3.

5 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. In Paolo Boldi and Luisa Gargano, editors, *Proceedings of the 5th International Conference on Fun with Algorithms (FUN 2010)*, volume 6099 of *LNCS*, pages 89–101. Springer, 2010. doi:10.1007/978-3-642-13122-6_11.

6 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On approximate jumbled pattern matching in strings. *Theory Comput. Syst.*, 50(1):35–51, 2012. doi:10.1007/s00224-011-9344-5.

7 Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing (STOC 2015)*, pages 31–40. ACM, ACM, 2015. doi:10.1145/2746539.2746568.

8 Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2009)*, pages 105–117, 2009. URL: http://www.stringology.org/event/2009/p10.html.

9 Ferdinando Cicalese, Travis Gagie, Emanuele Giaquinta, Eduardo Sany Laber, Zsuzsanna Lipták, Romeo Rizzi, and Alexandru I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013)*, volume 8214 of *LNCS*, pages 56–63. Springer, 2013. doi:10.1007/978-3-319-02432-5_10.

10 Ferdinando Cicalese, Eduardo Laber, Oren Weimann, and Raphael Yuster. Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence. In Juha Kärkkäinen and Jens Stoye, editors, *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*, pages 149–158. Springer, 2012. doi:10.1007/978-3-642-31265-6_12.

11 Stephane Durocher, Robert Fraser, Travis Gagie, Debajyoti Mondal, Matthew Skala, and Sharma V. Thankachan. Indexed geometric jumbled pattern matching. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 110–119. Springer, Springer, 2014. doi:10.1007/978-3-319-07566-2_12.

12 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6.

13 Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14):538–542, 2013. doi:10.1016/j.ipl.2013.04.013.

14 Danny Hermelin, Gad M. Landau, Yuri Rabinovich, and Oren Weimann. Binary jumbled pattern matching via all-pairs shortest paths, 2014. arXiv:1401.2065.

15 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Proceedings of the 21st Annual European Symposium on Algorithms (ESA 2013)*, volume 8125 of *LNCS*, pages 625–636. Springer, 2013. doi:10.1007/978-3-642-40450-4_53.

16 Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110(18-19):795–798, 2010. doi:10.1016/j.ipl.2010.06.012.

17 Tanaeem M. Moosa and M. Sohel Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Algorithms*, 10:5–9, 2012. doi:10.1016/j.jda.2011.08.003.

18 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. doi:10.1017/CBO9781316588284.

19 Shiho Sugimoto, Naoki Noda, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing Abelian regularities on RLE strings, 2017. arXiv:1701.02836.