On Hardness of Jumbled Indexing

Amihood Amir[1⋆], Timothy M. Chan[2⋆⋆], Moshe Lewenstein[⋆⋆⋆3], and Noa Lewenstein[4]

[1] Bar-Ilan University and Johns Hopkins University
[2] University of Waterloo
[3] Bar-Ilan University
[4] Netanya College

Abstract. Jumbled indexing is the problem of indexing a text T for queries that ask whether there is a substring of T matching a pattern represented as a Parikh vector, i.e., the vector of frequency counts for each character. Jumbled indexing has garnered a lot of interest in the last four years; for a partial list see [2, 6, 13, 16, 17, 20, 22, 24, 26, 30, 35, 36]. There is a naive algorithm that preprocesses all answers in $O(n^2|\Sigma|)$ time allowing quick queries afterwards, and there is another naive algorithm that requires no preprocessing but has $O(n \log |\Sigma|)$ query time. Despite a tremendous amount of effort there has been little improvement over these running times.

In this paper we provide good reason for this. We show that, under a 3SUM-hardness assumption, jumbled indexing for alphabets of size $\omega(1)$ requires $\Omega(n^{2-\epsilon})$ preprocessing time or $\Omega(n^{1-\delta})$ query time for any $\epsilon, \delta > 0$. In fact, under a stronger 3SUM-hardness assumption, for any constant alphabet size $r \geq 3$ there exist describable fixed constant ϵ_r and δ_r such that jumbled indexing requires $\Omega(n^{2-\epsilon_r})$ preprocessing time or $\Omega(n^{1-\delta_r})$ query time.

1 Introduction

Equal length strings are said to *jumble-match* if they are commutatively equivalent (sometimes called Abelian equivalent), i.e., if one string can be obtained from the other by permuting its characters. A jumble match can be described using *Parikh vectors* which are vectors maintaining the frequency count of each alphabet character. Two strings jumble-match if they have the same Parikh vector. We also say that a string jumble-matches a Parikh vector ψ if the string's Parikh vector is the same as ψ.

Parikh vectors were introduced in [37] and have been used to analyze grammars [31] and characterize commutative languages [27]. Furthermore, jumbled

⋆ Supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.
⋆⋆ Part of the work of this author was done while visiting the Department of Computer Science and Engineering, Hong Kong University of Science and Technology.
⋆⋆⋆ This research was done in part while the author was on sabbatical in the U. of Waterloo. The research is supported by BSF grant 2010437 and GIF grant 1147/2011.

pattern matching appears in various applications of computational biology, such as SNP discovery [10], analysis of similarities among different protein sequences [28], and automatic pattern discovery in biosequencing applications [21]. It has also been examined in the streaming model [33].

Jumbled pattern matching on its own can easily be solved by using a sliding window in linear time for the alphabet $\{1, \ldots, O(n)\}$, or $O(n \log |\Sigma|)$ time for a general alphabet Σ. In contrast, the exact pattern matching problem can only be solved in linear time via more complex techniques (e.g., see [29, 11]). Jumbled pattern matching has also been studied along with other metrics (e.g., see [14, 15, 1]).

1.1 Jumbled Indexing

Jumbled indexing (JI), currently under very active research, asks whether one can "index" jumbled matching. The goal is to preprocess a given text S efficiently so that when given a Parikh vector ψ one can quickly check whether there exists a substring of S that jumble-matches ψ.

For classical exact matching, text indexing paradigms of linear size and with near linear query time (in the query size) exist since the introduction of suffix trees [40]. Many other efficient text indexing structures have been studied since then, such as suffix array [34]. Other matching problems have also been successfully transformed into efficient indexing paradigms. For example, *parameterized matching* allows parametric symbols that are required to map to characters in a consistent manner. Parameterized matching was introduced by Baker [7, 8] for detection of repetitive similar modules in software and has applications for color images [3, 5, 39] and approximate image search [25]. Parameterized matching can be solved in linear time [4]. In [7] a parameterized suffix tree was introduced. Both the preprocessing and the query times are near-linear (where the latter is linear in the query size). Another example is order-preserving matchings, where two numerical strings match if their order is preserved. Efficient order-preserving matchings were presented in [32] and recently an order-preserving index was introduced [19] that can also be preprocessed in linear time with linear time queries (in the query size). Indexing with errors [18] has proven to be somewhat harder.

Given that jumbled matching can be trivially solved in linear time, for the above-mentioned alphabets, one would expect that jumbled indexing would be a relatively easy problem. However, jumbled indexing is surprisingly difficult.

There are two naive methods to solve jumbled indexing. One is to use the sliding window technique mentioned above for every query that arrives. This can be done in $O(n)$ time if the alphabet is a subset of $[n]$, where n is the text size. Another method is to preprocess all possible answers in advance by computing the Parikh vectors of every substring in $O(n^2|\Sigma|)$ time. Improving upon this has proven to be challenging even for constant-sized alphabets.

In an effort to make progress on JI the simplest version of the problem was considered, that of a binary alphabet. A neat property of a binary alphabet is that a Parikh vector (i, j) appears in text T iff i is between the minimum and maximum number of 1s over all substrings of length $i+j$. This was used in [16] to

obtain efficient query time by storing the minimum and maximum values of all possible lengths, yielding an index of $O(n)$ space and $O(1)$ query time. However, the preprocessing still took $O(n^2)$ time.

Burcsi et al. [13] and, independently, Moosa and Rahman [35] succeeded in improving the preprocessing time by a log factor to $O(\frac{n^2}{\log n})$. They achieved this by reducing binary JI to (min,+)-convolution which can be solved in $O(\frac{n^2}{\log n})$ time [12]. Later, Moosa and Rahman [36] improved this to $O(\frac{n^2}{\log^2 n})$ by using the four-Russians trick. Recently, Hermelin et al. [26] reduced the problem to (min,+)-matrix multiplication or all-pairs shortest paths, but a similar reduction has already appeared in an earlier paper by Bremner et al. [12]; with the latest breakthrough by Williams [41] on all-pairs shortest paths, the preprocessing time for binary JI becomes $O(\frac{n^2}{2^{\Omega((\log n/\log\log n)^{0.5})}})$. For the binary case there are also algorithms for run-length encoded strings [6, 24] and for an approximate version of the problem [17]. The binary case was also extended to trees [22].

Lately, there has been some progress also for non-binary alphabets. Kociumaka et al. [30] presented a solution for JI for any constant-sized alphabet Σ that uses $O(\frac{n^2\log^2\log n}{\log n})$ preprocessing time and space and answers queries in $O((\frac{\log n}{\log\log n})^{2|\Sigma|-1})$ time. Amir et al. [2] proposed a solution for constant-sized alphabets that preprocesses in $O(n^{1+\epsilon})$ time and answers queries in $\tilde{O}(m^{\frac{1}{\epsilon}})$ time, where m is the sum of the Parikh vector elements. In an even newer paper, Durocher et al. [20] considered alphabet size $|\Sigma| = o((\frac{\log n}{\log\log n})^2)$ and showed how to construct an index in $O(|\Sigma|(\frac{n}{\log_{|\Sigma|} n})^2)$ time and answer queries in $O(n^\epsilon + |\Sigma|)$ time, where $\epsilon > 0$ is an arbitrary small constant. This still leaves us in a sad state of affairs. In all the (exact) solutions mentioned for $|\Sigma| \geq 3$ the time complexity of preprocessing or the time complexity of querying is always within polylogarithmic factors of one of the above two naive algorithms. The question that has troubled the community in these last few years is whether jumbled indexing could be solved with $O(n^{2-\epsilon})$ preprocessing time and $O(n^{1-\delta})$ for some constants $\epsilon, \delta > 0$.

In this paper we show that for alphabets of $\omega(1)$ size this is impossible under a 3SUM-hardness assumption. We further show that for any constant alphabet size $r \geq 3$ there exist describable fixed constants ϵ_r and δ_r such that jumbled indexing requires $\Omega(n^{2-\epsilon_r})$ preprocessing time or $\Omega(n^{1-\delta_r})$ query time under a stronger 3SUM-hardness assumption.

1.2 3SUM

Numerous algorithmic problems have polynomial time upper bounds that we suspect are the best obtainable but proving matching lower bounds is difficult in classical computational models. Recently, a different approach for showing hardness (e.g. [42]) has been to choose an algorithmic problem that seem harder than others, e.g. maximum flow, APSP, edit distance, or 3SUM, and to use them as a hard primitive and reduce them to other problems that we would like to show are hard.

In this paper we use the *3SUM problem* defined as follows.

- **Input:** $x_1, x_2, \ldots, x_n$.
- **Output:** yes, if distinct i, j and k exist such that $x_i + x_j = x_k$. No, otherwise.

As far back as the mid 90's there were reductions from 3SUM, especially within the computational geometry community. Gajentaan and Overmars [23] were the first to reduce from 3SUM in order to provide evidence for near-quadratic complexity for computational geometry problems such as minimum-area triangle, finding 3 collinear points, and determining whether n axis-aligned rectangles cover a given rectangle. Others followed and quite a few problems are now known to be 3SUM-hard.

Pătraşcu [38] pointed out that most of the reductions transform the condition $x_i + x_j = x_k$ into some geometric or algebraic condition by common arithmetic, but it is difficult to use 3SUM for reductions to purely combinatorial problems, such as those on graphs or strings. To overcome this he defined *Convolution-3SUM*, a more restricted 3SUM version, which is just as hard as 3SUM in the sense that an $O(n^{2-\epsilon})$-time solution for Convolution-3SUM for some $\epsilon > 0$ would imply an $O(n^{2-\epsilon'})$-time solution for 3SUM for some $\epsilon' > 0$ [38].

The *Convolution-3SUM problem* is defined as follows.

- **Input:** $x_1, \ldots, x_n$.
- **Output:** Yes, if there are distinct i and j such that $x_i + x_j = x_{i+j}$. No, otherwise.

By shuffling and changing indices an alternative equivalent output is:

- **Output:** Yes, if there are distinct i and j such that $x_i - x_j = x_{i-j}$. No, otherwise.

We consider these problems in the RAM model with the elements belonging to an integer set $\{-u, \ldots, u\}$ as was assumed by others, e.g. [9, 38]. It is possible to achieve an algorithm of $O(u \log u)$ time for the 3SUM problem [9] by Fast Fourier transform. This can easily be transformed into an $O(nu \log(nu))$ time algorithm for Convolution-3SUM. Pătraşcu [38] pointed out that the techniques of Baran et al. [9] yield a (randomized) reduction, for the 3SUM problem, from a large domain $\{-u, \ldots, u\}$ to the domain of $\{-n^3, \ldots, n^3\}$. This reduction can be adapted for Convolution-3SUM from $\{-u, \ldots, u\}$ to $\{-n^2, \ldots, n^2\}$. The reason is that in 3SUM there are n^3 triples to consider when bounding the number of false positives, but in Convolution-3SUM there are only n^2 triples ($x_i + x_j = x_{i+j}$) to consider.

Hence, Convolution-3SUM for input $\subset \{-u, \ldots, u\}$ is hard if $u \geq n^2$ and is easier than $O(n^2)$ for $u \ll n$. Convolution-3SUM for inputs $\subset \{-n, \ldots, n\}$ seems (though has not proven) to be as hard as the general case. This leads us to state two hardness assumptions. For both we assume, as in [9, 38], the Word RAM model with words of $O(\log n)$ bits.

- **3SUM-hardness assumption:** Any algorithm for Convolution-3SUM requires $n^{2-o(1)}$ time in expectation to determine whether a set $\{x_1, \ldots, x_n\} \subset \{-n^2, \ldots, n^2\}$ contains a pair x_i, x_j such that $x_i - x_j = x_{i-j}$.
- **Strong 3SUM-hardness assumption:** Any algorithm for Convolution-3SUM requires $n^{2-o(1)}$ time in expectation to determine whether a set $\{x_1, \ldots, x_n\} \subset \{-n, \ldots, n\}$ contains a pair x_i, x_j such that $x_i - x_j = x_{i-j}$.

1.3 Preliminaries and Definitions

Let S be a string of length n over an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. An integer i is a *location* or a *position* in S if $i \in \{1, \ldots, |S|\}$. The substring $S[i\,..\,j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j. The string generated by a character a repeated r times is shorthanded with a^r.

The *Parikh vector* of a string S is $\psi(S) = (c_1(S), c_2(S), \ldots, c_{|\Sigma|}(S))$, where $c_i(S)$ is the count of occurrences of the i-th character of Σ. Two strings (of equal length) S and S' are said to *jumble-match* if they have the same Parikh vector. For a text T and pattern P we say that P *jumble-matches at location* i if the substring $T[i\,..\,i+|P|-1]$ jumble-matches P. *Jumbled pattern matching* refers to the problem where one is given a pattern and text and seeks all locations where the pattern jumble-matches. For a Parikh vector $\psi = (c_1, \ldots, c_{|\Sigma|})$, we denote its length with $|\psi|$ which is $\Sigma_{i=1}^{|\Sigma|} c_i$.

Jumbled indexing (JI, for short), also known as *histogram indexing*, *Parikh indexing*, or *permutation indexing*, is defined as follows.

- **Preprocess:** a text S over alphabet Σ.
- **Query:** Given a vector $\psi \in \mathbb{N}^{|\Sigma|}$, decide whether there is a substring S' such that the Parikh vector $\psi(S')$ is equal to ψ.

2 Hardness of Jumbled Indexing

2.1 Outline

We will show that, under the 3SUM-hardness assumption, one cannot improve the running time over the naive methods mentioned in the introduction by any polynomial factors for alphabets of super-constant size; and for alphabets of constant size there are polynomial time lower bounds, dependent on the alphabet size.

To achieve these results we reduce from 3SUM to JI. Naturally, we use Convolution-3SUM, which is more appropriate for problems with structure. A very high-level description of our reduction is as follows. A Convolution-3SUM input is transformed to JI by hashing the input values to much smaller sized values by using mod over a collection of primes. These are then novelly transformed to a string. The queries on the string simulate testing matchings mod primes in parallel. Using mod primes causes several problems, which lead to interesting ideas to overcome these obstacles.

2.2 Setup

Let $x_1, x_2, \ldots, x_n$ be the input of the Convolution-3SUM problem such that each $x_i \in \{-n^2, \ldots, n^2\}$. Under the strong 3SUM-hardness assumption, each $x_i \in \{-n, \ldots, n\}$.

We choose a collection of roughly equal-sized primes $p_1, \ldots, p_k$ (for some choice of k) with their product $p_1 \cdots p_k > n^2$ (or, under the strong 3SUM-hardness assumption, with $p_1 \cdots p_k > n$). It is possible to choose $p_1, \ldots, p_k \in \Theta(n^{2/k})$ (or, for the strong assumption, $p_1, \ldots, p_k \in \Theta(n^{1/k})$) to satisfy this requirement for any given $k \leq \frac{\log n}{\log \log n}$, because of the density of the primes.

The alphabet of JI in the reduction will consist of a character for each prime we choose, plus two more special characters we introduce later. Therefore, the JI alphabet size will be $|\Sigma| = k + 2$.

The lemma below follows directly from properties of mod and will be instrumental in obtaining our result.

Lemma 1. *Let $p_1, \ldots, p_k$ be a set of primes such that $p_1 \cdots p_k > u$ (with $u = n^2$ or $u = n$ depending on the hardness assumption). Let $i > j$. Then* $x_i - x_j = x_{i-j} \iff \forall r : (x_i - x_j) \bmod p_r = x_{i-j} \bmod p_r$

$$\iff \forall r : (x_i \bmod p_r) - (x_j \bmod p_r) \in \begin{cases} (x_{i-j} \bmod p_r) \\ (x_{i-j} \bmod p_r) - p_r \end{cases}$$

2.3 Reduction

In the reduction to the JI instance we will generate an input string S to be preprocessed and a set of n queries $Q_1, \ldots, Q_n$ which we now describe.

JI Input String We generate an input string S based on the Convolution-3SUM input $x_1, \ldots, x_n$. For every prime p_j we create a character a_j and for each x_i we create a substring

$$S_i = a_1^{EXP(i,1)} a_2^{EXP(i,2)} \cdots a_k^{EXP(i,k)},$$

where

$$EXP(i, j) = (x_{i+1} \bmod p_j) - (x_i \bmod p_j).$$

We note that, for the sake of simplicity, we are cheating since the exponent of a character in a string cannot be negative. We will shortly explain how to fix this.

Finally, we define

$$S = \$\#\ S_1\ \#\$\#\ S_2\ \#\$\#\ \cdots\ \#\$\#\ S_{n-1}\ \#\$,$$

where # and \$ are separator characters.

The structure of S is such that substrings beginning and ending within separators #\$# have the property that the number of occurrences of each character is reminiscent of the requirements of Lemma 1.

Lemma 2. *Consider the substring of S, $R_{(j,i)} = \$\#\ S_j\ \#\$\#\ \ldots\ \#\$\#\ S_{i-1}\ \#\$$. Each character a_ℓ has exactly $(x_i \bmod p_\ell) - (x_j \bmod p_\ell)$ occurrences in $R_{(j,i)}$.*

Proof. The character a_ℓ has $EXP(j,\ell)$ occurrences in S_j, $EXP(j+1,\ell)$ occurrences in $S_{j+1},\ldots$, $EXP(i-1,\ell)$ occurrences in S_{i-1}. Hence, we have $\Sigma_{d=j}^{i-1} EXP(d,\ell)$ occurrences of a_ℓ in $R_{(j,i)}$. Then $\Sigma_{d=j}^{i-1} EXP(d,\ell) = \Sigma_{d=j}^{i-1}(x_{d+1} \bmod p_\ell) - (x_d \bmod p_\ell)$, which telescopes to $(x_i \bmod p_\ell) - (x_j \bmod p_\ell)$. □

By combining Lemmas 1 and 2 we can deduce the following.

Corollary 1. *There is a solution $x_i - x_j = x_{i-j}$ to the Convolution-3SUM iff the number of occurrences of each character a_ℓ in $R_{(j,i)}$ is in*

$$\{(x_{i-j} \bmod p_\ell), (x_{i-j} \bmod p_\ell) - p_\ell\}.$$

To fix the problem of the negative exponent we set $D = \max_{i=1}^{k} p_i$ and change $EXP(i,j) = (x_{i+1} \bmod p_j) - (x_i \bmod p_j) + D$. Now, the exponent is not negative, but is still of order $\Theta(n^{2/k})$ (or $\Theta(n^{1/k})$ under the strong 3SUM-hardness assumption), which is the size of each prime. We leave it as an easy exercise to verify that Lemma 2 can be modified so that each character a_ℓ has exactly $(x_i \bmod p_\ell) - (x_j \bmod p_\ell) + D(i-j)$ occurrences in $R_{(j,i)}$ and, in turn, that Corollary 1 can be modified so that $a_\ell \in \{(x_{i-j} \bmod p_\ell) + D(i-j), (x_{i-j} \bmod p_\ell) - p_\ell + D(i-j)\}$.

JI Queries We generate n queries $\psi_1,\ldots,\psi_n$ for the jumbled indexing instance such that each ψ_L represents x_L, an element of the Convolution-3SUM input. The query ψ_L will imitate a query on the Convolution-3SUM data asking whether there exist i and j such that

(a) $x_L = x_i - x_j$ and
(b) $L = i - j$.

We will also embed the query with data requiring that

(c) any substring that jumble-matches the query ψ must be of the form $R_{(j,i)}$ from Lemma 2.

Obviously, answers to all queries ψ_L will be sufficient to derive a solution to Convolution-3SUM.

To enforce (c) and (b) we use the separators of S, #, and \$. For (c) we require the form of $R_{(j,i)}$ and for (b) we require $L = i - j$, which means that each potential substring $R_{(j,i)}$ should contain exactly L parts S_h.

Observation 1 *Any substring of S that jumble-matches the query ψ, where ψ has $L+1$ for \$ and $2L$ for #, must be of the form $R_{(j,i)}$ (of Lemma 2) and must satisfy $L = i - j$.*

Proof. Let R be a substring of S such that R jumble-matches ψ. Then each set of separators #$# fully contained in R contributes twice as many #'s than $'s. Since our query asks for $L+1$ $'s but only $2L$ #'s, it must be that R begins and ends with a $ and hence is of the form $R_{(j,i)}$. Moreover, since there are $L+1$ $'s and R begins and ends with a $, there must be exactly L parts S_h in R, implying that $L = i - j$. □

It remains to show how to adapt the query in order to enforce (a) $x_L = x_i - x_j$. Here we will use Corollary 1. It is sufficient to find the substrings $R_{(j,i)}$ such the number of occurrences of each character a_ℓ is either $((x_i - x_j) \bmod p_\ell) + DL$ or $((x_i - x_j) \bmod p_\ell) - p_\ell + DL$. However, checking two options (for each a_ℓ) cannot be done with one JI query. So, we split the query ψ_L into 2^k queries $\psi_L^{(1)}, \ldots, \psi_L^{(2^k)}$ for the 2^k different equalities that satisfy Corollary 1.

Hence, we have overall $2^k n$ JI queries. These queries provide a full answer to the Convolution-3SUM problem.

Theorem 2. *Consider the jumbled indexing problem with text size s and alphabet size $r \geq 5$. Then under the 3SUM-hardness assumption, one of the following holds for any fixed $\epsilon > 0$:*

1. *the preprocessing time is $\Omega(s^{2-\frac{4}{r}-\epsilon})$, or*
2. *the query time is $\Omega(s^{1-\frac{2}{r}-\epsilon})$.*

Proof. Without loss of generality, assume that $r \leq \frac{\log s}{\log\log s}$ (otherwise, $s^{\frac{1}{r}} = \tilde{\Theta}(1)$ and we may as well make r equal to $\frac{\log s}{\log\log s}$).

Let $x_1, \ldots, x_n \in \{-n^2, \ldots, n^2\}$ be the input of the Convolution-3SUM problem. We apply the above reduction and generate the string S as described. Denote its length by s. Recall that the alphabet size is $r = |\Sigma| = k + 2$, where k is the number of primes (the 2 is for the separators $ and #). Since each prime $p_i \in \Theta(n^{2/k})$, we have $s = O(kn^{2/k}n) = \tilde{O}(n^{\frac{r}{r-2}})$ for $k + 2 = r \in o(\log n)$. In other words, $n = \tilde{\Omega}(s^{1-\frac{2}{r}})$.

Applying the preprocessing and subsequently answering all $2^k n$ defined queries yields a solution to the Convolution-3SUM problem. Letting $P(s)$ and $Q(s)$ be the preprocessing and query time, we then have $P(s) + 2^k n Q(s) \geq \Omega(n^{2-\epsilon})$.

For $k + 2 = r \in o(\log n)$ we note that $2^k \in o(n^\epsilon)$. We must thus have $P(s) \geq \Omega(n^{2-\epsilon}) = \Omega(s^{2(1-\frac{2}{r})-O(\epsilon)})$ or $Q(s) \geq \Omega(n^{1-O(\epsilon)}) = \Omega(s^{1-\frac{2}{r}-O(\epsilon)})$. □

Note that for $r \leq 4$, the bound in the above theorem becomes vacuous. We can get somewhat better bounds under the strong 3SUM-hardness assumption.

Theorem 3. *Consider the jumbled indexing problem with text size s and alphabet size $r \geq 4$. Then under the strong 3SUM-hardness assumption, one of the following holds for any fixed $\epsilon > 0$:*

1. *the preprocessing time is $\Omega(s^{2-\frac{2}{r-1}-\epsilon})$, or*

2. the query time is $\Omega(s^{1-\frac{1}{r-1}-\epsilon})$.

The proof is the same as in the previous theorem, but with $p_i \in \Theta(n^{1/k})$.

By the same proof and further calculations, we can also get a slightly strengthened lower bound of $\Omega(s^2/2^{O(\sqrt{\log s})})$ preprocessing time or $\Omega(s/2^{O(\sqrt{\log s})})$ query time for alphabet size $r = \Theta(\sqrt{\log s})$, under the assumption that 3SUM has an $\Omega(n^2/2^{O(\sqrt{\log n})})$ lower bound.

2.4 Hardness of JI with Alphabet Size 3

The reduction we have presented contains two separators in the string S. Recall that for every prime we also construct a character. We require that the multiplication of the primes be $> n$ for strong 3SUM-hardness and $> n^2$ for 3SUM-hardness. However, if a prime is of order $\Omega(n)$ then the size of the string S would be $\Omega(n^2)$, too large to gain anything from the reduction. Hence, we need at least two primes for strong 3SUM-Hardness and three primes for 3SUM-hardness. In this section we generate a string which requires only one separator, and for 2 primes p and q of size $\Theta(\sqrt{n})$ this yields a nontrivial result under the strong 3SUM-hardness assumption.

While we construct a different string for JI and need to argue a claim similar to Lemma 2 and Observation 1, the structure of the proof remains the same.

Let a be a character representing prime p, and b be a character that represents prime q, and # be a separator character. Let $D = \max\{p, q\}$. Define

$S_i = (a\#)^{(x_{i+1} \bmod p)-(x_i \bmod p)+D}(b\#)^{(x_{i+1} \bmod q)-(x_i \bmod q)+D}$ and
$S = \#^D a^{2D} \#^D \, S_1 \, \#^D a^{2D} \#^D \, S_2 \, \#^D a^{2D} \#^D \, \cdots \, \#^D a^{2D} \#^D \, S_{n-1} \, \#^D a^{2D} \#^D$,
where $\#^D a^{2D} \#^D$ is the separator (a has a double role).

Define $R_{(j,i)} = \#^D \, S_j \, \#^D a^{2D} \#^D \, \cdots \, \#^D a^{2D} \#^D \, S_{i-1} \, \#^D$.

It is easy to verify, similar to Lemma 2, that a has exactly $(x_i \bmod p) - (x_j \bmod p) + D(i-j) + 2D(i-j-1)$ occurrences in $R_{(j,i)}$ and b has exactly $(x_i \bmod q) - (x_j \bmod q) + D(i-j)$ occurrences in $R_{(j,i)}$. Hence, as in Corollary 1, there is a solution $x_i - x_j = x_{i-j}$ to Convolution-3SUM iff the number of occurrences of a in $R_{(j,i)}$ is in $\{((x_i - x_j) \bmod p) + D(3i-3j-2), ((x_i - x_j) \bmod p) - p + D(3i-3j-2)\}$ and the number of occurrences of b in $R_{(j,i)}$ is in $\{((x_i - x_j) \bmod p) + D(i-j), ((x_i - x_j) \bmod p) - p + D(i-j)\}$.

The tricky part is to obtain an alternative to Observation 1. The difficulty stems from the fact that a and # appear both in the separator part and in the S_i's.

Observation 4 *Say we have a Parikh vector* $\psi = (n_1, n_2, n_1 + n_2 + 4D)$ *for* $(a, b, \#)$, *with* $L = (n_1 \text{ div } 3D) + 1$. *Then any substring of* S *which jumble-matches* ψ *is of the form* $R_{(j,i)}$. *Moreover* $i - j = L$.

Proof. Define $\Delta = 4D$ the difference between the number of #'s and the number of a's and b's (put together). Let x be a substring of S which jumble-matches ψ.

Any S_l or separator $\#^D a^{2D} \#^D$ fully contained in x has a balanced number of #'s and non-#'s and, hence, does not affect Δ. So, at either end there is a part of a separator or an S_l which both together contributes to Δ. It is straightforward to confirm that the only way this is possible is having x begin with $\#^D$, the prefix of a separator, and end with $\#^D$, the suffix of a separator. Hence, x has the required form $R_{(j,i)}$.

We show that $i-j = L = (n_1 \text{ div } 3D)+1$. We have claimed that the number of a's in $R_{(j,i)}$ is $(x_i \bmod p) - (x_j \bmod p) + D(3i-3j-2)$. Hence, since $R_{(j,i)}$ jumble-matches ψ, we have $n_1 = (x_i \bmod p) - (x_j \bmod p) + D + 3D(i-j-1)$. Since each $(x_i \bmod p) - (x_j \bmod p) + D \in [0, 2D)$ it follows that $(n_1 \text{ div } 3D) = i-j-1$. □

Finally, for a given L all of our queries have $n_1 = (x_L \bmod p) + D(3L-2)$ or $n_1 = (x_L \bmod p) - p + D(3L-2)$ in location a of the Parikh vector. In both cases, $L = (n_1 \text{ div } 3D) + 1$, satisfying the requirement of Observation 4. Hence,

Theorem 5. *Consider the jumbled indexing problem with text size s and alphabet size 3. Under the strong 3SUM-hardness assumption, one of the following holds for any fixed $\epsilon > 0$:*

1. *the preprocessing time is $\Omega(s^{\frac{4}{3}-\epsilon})$, or*
2. *the query time is $\Omega(s^{\frac{2}{3}-\epsilon})$.*

Proof. Note that $p, q \in \Theta(\sqrt{n})$. Hence, S is of length $s = O(n^{\frac{3}{2}})$. Following the same arguments as in Theorem 2 yields the result. □

Epilogue. In a forthcoming work, the second and third author will present new improved algorithms for the jumbled indexing problem for any constant alphabet size $r \geq 2$ that achieves truly sublinear query time and $O(n^{2-\frac{2}{r+O(1)}})$ preprocessing time, thus nearly matching the lower bound in Theorem 3.

References

1. Amihood Amir, Alberto Apostolico, Gad M. Landau, and Giorgio Satta. Efficient text fingerprinting via Parikh mapping. *J. Discrete Algorithms*, 1(5-6):409–421, 2003.
2. Amihood Amir, Ayelet Butman, and Ely Porat. On the relationship between histogram indexing and block-mass indexing. *Philosophical Transactions A*, to appear.
3. Amihood Amir, Kenneth Ward Church, and Emanuel Dar. Separable attributes: a technique for solving the sub matrices character count problem. In *SODA*, pages 400–401, 2002.
4. Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994.
5. G. Phanendra Babu, Babu M. Mehtre, and Mohan S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, 1995.

6. 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.
7. Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.
8. Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.
9. Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3SUM. In *WADS*, pages 409–421, 2005.
10. Sebastian Böcker. Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. *Bioinformatics*, 23(2):5–12, 2007.
11. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
12. 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*, 69:294–314, 2014.
13. Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. In *FUN*, pages 89–101, 2010.
14. Ayelet Butman, Revital Eres, and Gad M. Landau. Scaled and permuted string matching. *Inf. Process. Lett.*, 92(6):293–297, 2004.
15. Ayelet Butman, Noa Lewenstein, and Ian J. Munro. Permuted scaled matching. In *CPM 2014, to appear*.
16. Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In *Prague Stringology Conference*, pages 105–117, 2009.
17. Ferdinando Cicalese, Eduardo Sany Laber, Oren Weimann, and Raphael Yuster. Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence. In *CPM*, pages 149–158, 2012.
18. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.
19. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *SPIRE*, pages 84–95, 2013.
20. Stephane Durocher, J. Ian Munro, Debajyoti Mondal, and Sharma V. Thankachan. Jumbled pattern matching over large alphabets. Manuscript, personal communication, 2014.
21. Revital Eres, Gad M. Landau, and Laxmi Parida. Permutation pattern discovery in biosequences. *Journal of Computational Biology*, 11(6):1050–1060, 2004.
22. Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. In *ESA*, pages 517–528, 2013.
23. Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.
24. Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14-16):538–542, 2013.
25. Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms*, 3(3), 2007.
26. Danny Hermelin, Gad M. Landau, Yuri Rabinovich, and Oren Weimann. Binary jumbled pattern matching via all-pairs shortest paths. Manuscript, http://arxiv.org/abs/1401.2065, 2014.

27. Stepan Holub. Parikh test sets for commutative languages. *ITA*, 42(3):525–537, 2008.
28. Xiaolu Huang, H. Ali, A. Sadanandam, and R. Singh. SRPVS: a new motif searching algorithm for protein analysis. In *Computational Systems Bioinformatics Conference, 2004. CSB 2004. Proceedings. 2004 IEEE*, pages 674–675, 2004.
29. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
30. Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *ESA*, pages 625–636, 2013.
31. Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *LICS*, pages 80–89, 2010.
32. Marcin Kubica, Tomasz Kulczynski, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013.
33. Lap-Kei Lee, Moshe Lewenstein, and Qin Zhang. Parikh matching in the streaming model. In *SPIRE*, pages 336–341, 2012.
34. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
35. Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110(18-19):795–798, 2010.
36. 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.
37. Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.
38. Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610, 2010.
39. Michael J. Swain and Dana H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.
40. Peter Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11, 1973.
41. Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *STOC*, 2014, to appear.
42. Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654, 2010.