Recognizing Well-Parenthesized Expressions in the Streaming Model*

Frédéric Magniez[†] Claire Mathieu[‡] Ashwin Nayak[§]

Abstract

Motivated by a concrete problem and with the goal of understanding the relationship between the complexity of streaming algorithms and the computational complexity of formal languages, we investigate the problem $\textsc{Dyck}(s)$ of checking matching parentheses, with s different types of parentheses.

We present a one-pass randomized streaming algorithm for $\textsc{Dyck}(2)$ with space of $\mathrm{O}(\sqrt{n \log n}\,)$ bits, time per letter $\mathrm{polylog}(n)$, and one-sided error. We prove that this one-pass algorithm is optimal, up to a $\log n$ factor, even when two-sided error is allowed.

Surprisingly, the space requirement shrinks drastically if we have access to the input stream *in reverse*. We present a two-pass randomized streaming algorithm for $\textsc{Dyck}(2)$ with space of $\mathrm{O}((\log n)^2)$, time $\mathrm{polylog}(n)$ and one-sided error, where the second pass is in the reverse direction. Both algorithms can be extended to $\textsc{Dyck}(s)$ since this problem is reducible to $\textsc{Dyck}(2)$ for a suitable notion of reduction in the streaming model. Except for an extra $\mathrm{O}(\sqrt{\log s}\,)$ multiplicative overhead in the space required in the one-pass algorithm, the resource requirements are of the same order.

For the lower bound, we exhibit hard instances $\textsc{Ascension}(m)$ of $\textsc{Dyck}(2)$ with length in $\Theta(mn)$. We embed these in what we call a "one-pass" communication problem with $2m$-players, where $m \in \tilde{\mathrm{O}}(n)$. To establish the hardness of $\textsc{Ascension}(m)$, we follow the "information cost" approach, but with a few twists. We prove a direct sum result that reduces $\textsc{Ascension}(m)$ to a *two-player* protocol for $\textsc{Mountain}$, which is in fact a variant of $\textsc{Index}$, a fundamental problem in communication complexity. We finish the argument with a new information cost lower bound for $\textsc{Mountain}$.

1 Introduction

The area of streaming algorithms has experienced tremendous growth in many applications since the late 1990s. Streaming algorithms sequentially scan the whole input piece by piece in one pass, or in a small number of passes (i.e., they do not have random access to the input), while using sublinear memory space, ideally polylogarithmic in the size of the input. The design of streaming algorithms is motivated by the explosion in the size of the data that algorithms are called upon to process in everyday real-time applications. Examples of such applications occur in bioinformatics for genome decoding, in Web databases for the search of documents, or in network monitoring. The analysis of Internet traffic [AMS99], in which traffic logs are queried, was one of the first applications of this kind of algorithm. Although these would have ramifications for massive data such as DNA sequences and large XML files, few studies have been made in the context of formal languages. For instance, in the context of databases, properties decidable by streaming algorithms

*A preliminary version of this work appeared in *Proceedings of 42nd ACM Symposium on Theory of Computing*, pages 261–270, 2010.

[†]LIAFA, Univ. Paris 7, CNRS; F-75205 Paris, France. `frederic.magniez@univ-paris-diderot.fr`. Supported in part by the French ANR Blanc project ANR-12-BS02-005 (RDAM).

[‡]Département d'Informatique UMR CNRS 8548, École Normale Supérieure, Paris, France. `cmathieu@di.ens.fr`. Part of this work was funded by NSF grant CCF-0728816 and by the French ANR Blanc project ANR-12-BS02-005 (RDAM).

[§]C&O and IQC, U. Waterloo, 200 University Ave. W., Waterloo ON N2L 3G1, Canada. `ashwin.nayak@uwaterloo.ca`. Work done in part while visiting the Center for Computational Intractability, Rutgers University and DIMACS, with support from NSF grants CCF-832797 and CCF 832787. Research also supported in part by NSERC Canada.

have been studied [SV02, SS07], but only in the restricted case of deterministic and constant memory space algorithms.

Motivated by a concrete problem and with the goal of understanding the relationship between the complexity of streaming algorithms and the computational complexity of formal languages, we investigate the problem $\textsc{Dyck}(s)$ of checking matching parentheses, with s different types of parentheses. Regular languages are by definition decidable by deterministic streaming algorithms with constant space. The $\textsc{Dyck}$ languages are some of the simplest context-free languages and yet already powerful. These languages play a central role in the theory of context-free languages, since every context-free language L can be mapped to a subset of $\textsc{Dyck}(s)$ [CS63], for some s. In addition to its theoretical importance, the problem of checking matching parentheses is encountered frequently in database applications, for instance in verifying that an XML file is well-formed.

The problem of deciding membership in $\textsc{Dyck}(s)$ has already been addressed in the massive data setting, more precisely through property testing algorithms. An ε-property tester [BK95, BLR93, GGR98] for a language L accepts all strings of L and rejects all strings which are ε-far from strings in L with respect to the normalized Hamming distance. For every fixed $\varepsilon > 0$, $\textsc{Dyck}(1)$ is ε-testable in constant time [AKNS01], whereas for $s > 1$, $\textsc{Dyck}(s)$ is ε-testable in time $\tilde{\mathrm{O}}(n^{2/3})$, with a lower bound of $\tilde{\Omega}(n^{1/11})$ [PRR03]. Feigenbaum, Kannan, Strauss, and Viswanathan [FKSV02] have compared property testers and streaming algorithms. Property testers are constrained to read only small portions of the input due to expectation of small processing time. In contrast, streaming algorithms have the advantage of access to the entire string, albeit not in a random access fashion.

With random access to the input, context-free languages are known to be recognizable in space $\mathrm{O}((\log n)^2)$ [HU69]. In the special case of $\textsc{Dyck}(s)$, logarithmic space is sufficient, as we may run through all possible levels of nesting, and check parentheses at the same level. This scheme does not seem to translate easily to streaming algorithms, even with a small number of passes over the input.

In the streaming model, $\textsc{Dyck}(1)$ has a one-pass streaming algorithm with logarithmic space, using a height counter. Using the linear lower bound for two-way deterministic communication protocols for $\textsc{Equality}$, we can deduce that $\textsc{Dyck}(2)$ requires space $\Omega(n/T)$ for deterministic streaming algorithms with T passes. In particular, $\textsc{Dyck}(2)$ requires linear space for deterministic one-pass streaming algorithms. A relaxation of $\textsc{Dyck}(s)$ is $\textsc{Identity}(s)$ in the free group with s generators, where local simplifications $\overline{a}a = \epsilon$ are allowed in addition to $a\overline{a} = \epsilon$, for every type of parenthesis $(a, \overline{a})$. There is a logarithmic space algorithm for recognizing the language $\textsc{Identity}(s)$ [LZ77] that can easily be massaged into a one-pass streaming algorithm with polylogarithmic space. Again, this algorithm does not extend to $\textsc{Dyck}(s)$.

We show that $\textsc{Dyck}(s)$ is reducible to $\textsc{Dyck}(2)$, for a suitable notion of reduction in the streaming model, with a $\log s$ factor expansion in the input length. First, we present a one-pass randomized streaming algorithm for $\textsc{Dyck}(2)$.

Theorem 1. *Let $c > 0$ be any constant. There is a one-pass randomized streaming algorithm that checks if a stream of length n belongs to $\textsc{Dyck}(2)$, uses space $\mathrm{O}(\sqrt{n \log n}\,)$ and time $\mathrm{polylog}(n)$ per letter. If the stream belongs to $\textsc{Dyck}(2)$ then the algorithm accepts it with certainty; otherwise it rejects it with probability at least $1 - n^{-c}$.*

If the length of the stream x is not known in advance, we may use standard techniques to extend the algorithm. Namely, we use an estimate for the length that is scaled geometrically as needed. The extended algorithm continues to accept streams in $\textsc{Dyck}(2)$ with certainty. The error probability of the resulting algorithm when $x \notin \textsc{Dyck}(2)$ is guaranteed to be smaller than δ, for any pre-specified constant $\delta > 0$.

If we had no space constraints, deciding $\textsc{Dyck}(2)$ would be very simple: when we encounter an upstep (a or b), push it on a stack, when we encounter a downstep ($\overline{a}$ or $\overline{b}$), pop the top item from the stack and check that they match. However the stack may grow to linear size in this process. To avoid this growth, the basic strategy of our algorithm is to use a linear hash function to periodically (every $\sqrt{n/\log n}$ letters) compress stack information. As long as we compress sequences of only upsteps or only downsteps, all at different heights, we are able to detect mismatches with high probability. The algorithm has one-sided error; it accepts words that belong to the language with certainty. Although it is simple, we show that this appealing algorithm is nearly optimal in its space usage, even when two-sided error is allowed.

Corollary 1. *Every one-pass randomized streaming algorithm for* DYCK(2) *with (two-sided) error* $\mathrm{O}(1/n \log n)$ *on inputs of length* n *uses* $\Omega(\sqrt{n \log n})$ *space.*

In the preliminary version of this article [MMN10], we conjectured that a similar lower bound continues to hold if we read the stream several times, but always in the same direction. This conjecture has since been confirmed; we elaborate on this in Section 5. Surprisingly, the situation is drastically different if we can read the data stream *in reverse*. We present a second algorithm, a randomized two-pass streaming algorithm for DYCK(2) with polylogarithmic space and time, where the second pass is in the reverse direction.

Theorem 2. *Let* $c > 0$ *be a constant. There is a bidirectional two-pass randomized streaming algorithm for* DYCK(2) *with space* $\mathrm{O}((\log n)^2)$ *and time* $\mathrm{polylog}(n)$ *for inputs of length* n*. If the input belongs to* DYCK(2) *then the algorithm accepts it with certainty; otherwise it rejects it with probability at least* $1 - n^{-c}$*.*

The above algorithm may be extended to streams of unknown length in the same manner as the unidirectional one. The rejection probability for inputs not in DYCK(2) is then only guaranteed to be at least $1 - \delta$, for any pre-specified constant δ, whereas inputs in DYCK(2) are accepted with certainty.

The bidirectional algorithm uses a hierarchical decomposition of the stream into blocks; whenever the algorithm reaches the end of a block, it compresses the information about subwords from within the block. This compression is what reduces the stack size from $\Theta(\sqrt{n \log n}\,)$ down to $\mathrm{O}(\log n)$, but prevents us from checking that certain matching pairs of parentheses are well-formed. However, given the profile of the word (i.e., the sequence of heights), we can pinpoint exactly the matching pairs that do not get checked. As it turns out, a pair that does not get checked when scanning the input left to right is necessarily checked when scanning in the reverse direction. Like the one-pass algorithm, this algorithm has only one-sided error, and always accepts words that belong to the language. We note that it is straightforward to extend the algorithms so that they recognize the language of substrings (which are subwords of consecutive letters) of DYCK(2).

As mentioned above, we also investigate the lower bound on the space required for any one-pass randomized streaming algorithm. Such a lower bound is by nature hard to prove because of the connection of the problem with IDENTITY(2). Moreover, proving a non-trivial lower bound based on two-party communication complexity is hopeless: the related communication problem automatically reduces to EQUALITY after local checks and simplifications by both players, leading to only an $\Omega(\log n)$ lower bound. Instead, we build hard instances ASCENSION(m) of DYCK(2) with length in $\Theta(mn)$, that we embed in a "one-pass" communication problem with $2m$ players, where $m \in \tilde{\Theta}(n)$. The constraint is that the length of each message in the protocol be less than σ, a function of n. Our main lower bound result (**Theorem 4**) is that such a protocol requires $\sigma \in \Omega(n)$, which proves that our one-pass algorithm is optimal for probability of error of order $1/n \log n$, and within an $\mathrm{O}(\log n)$ factor of optimal for constant error (**Corollary 1**).

To establish the hardness of ASCENSION(m), we follow the "information cost" approach taken in Refs. [CSWY01, SS02, BJKS04, JKS03, JRS03b], among other works before and since. The technique comes with a few twists in our case. We prove a *direct sum* result that captures the relationship of $2m$-player problem ASCENSION(m) to solving m instances of an intermediate problem MOUNTAIN, which involves only *two* players. MOUNTAIN is a variant of INDEX, a fundamental problem in communication complexity. This variant has been studied in the one-way communication model as "serial encoding" [ANTV99, Nay99], and in later works on streaming and sketching as "Augmented Index" (see, e.g., Refs. [KNW10, DBIPW10]).

We adapt the notion of information cost to suit both the nature of streaming algorithms and of our problem. The idea is to focus on the information about a part of the input contained in a part of the protocol transcript, given the remaining inputs. Using this notion of information cost, we prove the direct sum result (**Lemma 7**). A remarkable device here, originally developed by Bar-Yossef, Jayram, Kumar, and Sivakumar [BJKS04], is the use of an "easy" distribution for the information cost for protocols, that are correct with high probability in the worst case. The use of an easy distribution "collapses" ASCENSION(m) to an instance of MOUNTAIN, which may be planted in any one of the m coordinates. Finally, we prove a new information cost lower bound for MOUNTAIN using a medley of combinatorial and information-theoretic means.

In protocols for ASCENSION(m) we allow access to only public coins by all players, whereas in protocols for MOUNTAIN we allow one of the players, Bob, access also to private coins (while Alice, the other player, may only access public coins). This mixture between public and private coins for MOUNTAIN arises from a balancing act between the direct sum result and our lower bound for MOUNTAIN (**Theorem 3**). Namely, we prove the lower bound for MOUNTAIN when Alice only uses public coins, whereas the direct sum only holds, with our definition of information cost, when Bob has access to additional private coins. The mixing of public and private coins in the analysis of information cost has also been observed and similarly tackled in earlier works (see, e.g., Ref. [CCM08]).

We note that as a bonus, our lower bound (**Theorem 4**) provides a $\tilde{\Omega}(\sqrt{n})$ lower bound for the problem of checking priority queues in the one-pass streaming model, solving an open problem posed by Chu, Kannan, and McGregor [CKM07].

Acknowledgements

For earlier discussions, F.M. would like to thank Michel de Rougemont, Miklos Santha, Umesh Vazirani, and especially Pranab Sen, who, among other things, noticed that the logarithmic space algorithm for IDENTITY(s) in [LZ77] can be converted to a one-pass randomized streaming algorithm with logarithmic space. We would also like to thank an anonymous STOC'10 referee for pointing out a $\sqrt{\log n}$ factor improvement of our original one-pass algorithm, and the anonymous SICOMP referees for their meticulous comments.

2 Definitions and preliminaries

Definition 1 (DYCK)**.** *Let s be a positive integer. Then* DYCK(s) *denotes the language over alphabet $\Sigma = \{a_1, \overline{a}_1, \ldots, a_s, \overline{a}_s\}$ defined recursively by:*

$$\textsc{Dyck}(s) = \epsilon + \sum_{i \leq s} a_i \cdot \textsc{Dyck}(s) \cdot \overline{a}_i \cdot \textsc{Dyck}(s).$$

We also denote by DYCK(s) the problem of deciding whether a word $w \in \Sigma^*$ is in the language DYCK(s).

In streaming algorithms, a *pass* on an input $x \in \Sigma^n$ means that x is presented as an *input stream* $x_1, x_2, \ldots, x_n$, which arrives sequentially, i.e., letter by letter in this order. For simplicity, we assume throughout this article that the length n of the input is always given to the algorithm in advance. Nonetheless, all our algorithms can be adapted using standard techniques to the case in which n is unknown until the end of a pass. See [Mut05] for an introduction to streaming algorithms.

Definition 2 (Streaming algorithm)**.** *Fix an alphabet Σ. A k-pass deterministic (resp. randomized)* streaming algorithm ***A*** *with space $s(n)$ and time $t(n)$ is a deterministic (resp. randomized) algorithm such that for every input stream $x \in \Sigma^n$:*

1. ***A*** *performs k sequential passes on x;*
2. ***A*** *maintains a memory space of size $s(n)$ bits while reading x;*
3. ***A*** *has running time at most $t(n)$ per letter x_i;*
4. ***A*** *has preprocessing and postprocessing time $t(n)$.*

*We say that **A** is* bidirectional *if it is allowed to access to the input in the reverse order, after reaching the end of the input. Then the parameter k is the total number of passes in either direction.*

Definition 3 (Streaming reduction)**.** *Fix two alphabets Σ_1 and Σ_2. A problem P_1 is $f(n)$-streaming reducible to a problem P_2 with space $s(n)$ and time $t(n)$, if for every input $x \in \Sigma_1^n$, there exists $y = y_1 y_2 \ldots y_n$, with $y_i \in \Sigma_2^{f(n)}$, such that:*

1. *y_i can be computed deterministically from x_i using space $s(n)$ and time $t(n)$;*

2. *From a solution of P_2 with input y, a solution on P_1 with input x can be computed with space $s(n)$ and time $t(n)$.*

The following is immediate.

Proposition 1. *Let P_1 be $f(n)$-streaming reducible to a problem P_2 with space $s_0(n)$ and time $t_0(n)$. Let $\boldsymbol{A}$ be a k-pass streaming algorithm for P_2 with space $s(n)$ and time $t(n)$. Then there is a k-pass streaming algorithm for P_1 with space $s(n \times f(n)) + s_0(n)$ and time $t(n \times f(n)) + t_0(n)$ with the same properties as $\boldsymbol{A}$ (deterministic/randomized, unidirectional/bidirectional).*

Moreover, we need only study $\textsc{Dyck}(s)$ with $s = 2$:

Proposition 2. $\textsc{Dyck}(s)$ *is $\lceil \log s \rceil$-streaming reducible to* $\textsc{Dyck}(2)$ *with space and time* $\mathrm{O}(\log s)$*.*

Proof. We encode a parenthesis a_i by a word of length $l = \lceil \log s \rceil$ with only parentheses of type b, c. We let $f(a_i)$ be the binary expansion of i over l bits where 0 is replaced by b and 1 by c. Then $f(\overline{a}_i)$ is defined similarly, except that we write the binary expansion of i in the opposite order and replace 0 by $\overline{b}$ and 1 by $\bar{c}$. Then $x_1 \dots x_n$ is in $\textsc{Dyck}(s)$ if and only $f(x_1) \dots f(x_n)$ is in $\textsc{Dyck}(2)$. $\square$

Since the parameter s is a constant independent of the length of the input stream, the above reduction can be implemented with constant space and time. For example, in parsing XML files, given an upstep (*start-tag*) `<`w`>` (respectively, a downstep (*end-tag*) `</`w`>`), where w is an ASCII string denoting the type of parenthesis (*tag*), we can generate the above encoding of w into b, c (respectively, into $\overline{b}, \overline{c}$), while reading w as a stream itself, i.e., character by character.

By Proposition 2, it is enough to design streaming algorithms for $\textsc{Dyck}(2)$. That is the objective of the next section.

3 Algorithms

From now on we consider $\textsc{Dyck}(2)$ where the input is a stream of n letters $x_1 x_2 \dots x_n$ in the alphabet $\Sigma = \{a, \overline{a}, b, \overline{b}\}$. We first introduce a few definitions. An *upstep* is a letter a or b, a *downstep* is a letter $\overline{a}$ or $\overline{b}$. For integers $i \le j$, we denote by $[i, j]$ the set of integers $\{i, i+1, \dots, j\}$, and by $x[i, j]$ the subword $x_i x_{i+1} \dots x_j$. We also use the notation $x[i]$ for x_i when we also consider sequences of words. For ease of notation, we identify an increasing sequence $i_1 < i_2 < \cdots < i_m$ of indices with the corresponding subword $x_{i_1} x_{i_2} \dots x_{i_m}$ of x. We also use this correspondence in reverse when the indices of the subword are clear from the context. The number of occurrences of the letter p in a word x is denoted by $|x|_p$. In the absence of any subscript, $|x|$ denotes the length of the word.

Definition 4 (Height, Matching pair, Well-formed)**.** *Let $x \in \Sigma^n$.*
The height *of x is* $\mathrm{height}(x) = |x|_a + |x|_b - |x|_{\overline{a}} - |x|_{\overline{b}}$.
For $1 \le i < j \le n$, (i, j) is a matching pair *for x if* $\mathrm{height}(x[1, i-1]) = \mathrm{height}(x[1, j])$ *and* $\mathrm{height}(x[1, k]) > \mathrm{height}(x[1, i-1])$ *for all $k \in \{i, \dots, j-1\}$.*
The height of a matching pair (i, j) *is* $\mathrm{height}(x[1, i-1])$.
A matching pair (i, j) for x is well-formed*, if $(x[i], x[j])$ equals $(a, \overline{a})$ or $(b, \overline{b})$,* ill-formed *otherwise.*

It follows that any index i forms a matching pair with at most one other index, and that a matching pair consists of an upstep and a downstep. These definitions are extended to subsets $I \subseteq [1, n]$ of indices of letters of x. For instance, we say that I is a *matching set* for x, if I is the union of $\{i, j\}$ over the matching pairs (i, j) for x. Observe that when $i < j$ we have the following equivalence: (i, j) is a matching pair for x if and only if $\{i, i+1, i+2, \dots, j\}$ is a matching set for x.

Define a partial order $\prec$ between words such that $u \prec v$ if and only if u is obtained from v by removing zero or more of its matching pairs. This order is well defined, and in particular transitive, since matching pairs of u are still matching pairs of v, up to reindexing. (This may be proven by a straightforward inductive argument.)

Proposition 3. *Let u, v be words such that $u \prec v$ and $u = u_1 u_2 \cdots u_m = v_{i_1} v_{i_2} \cdots v_{i_m}$ is obtained by removing the matching set $[1, n] \backslash \{i_1, i_2, \ldots, i_m\}$ from v. If $(j, k) \in [1, m]^2$ is a matching pair for u, then (i_j, i_k) is a matching pair for v.*

To prove correctness of our algorithms, we use the following characterization of Dyck(2).

Proposition 4. *Let $x \in \Sigma^n$. Then*

1. *$[1, n]$ is a (possibly ill-formed) matching set for x if and only if* $\mathrm{height}(x) = 0$ *and the height of every prefix of x is nonnegative;*
2. *$[1, n]$ is a well-formed matching set for x if and only if $x \in$* Dyck(2).

Proof. The proof is by induction on the length n of x. The first part may be established by a straightforward induction on n. We prove the second part. For $n = 0$ the result is true since the empty word is in Dyck(2) and $\emptyset$ is a well-formed set.

Let $x \in$ Dyck(2) of length n. By Definition 1, there exist $y, z \in$ Dyck(2) such that $x = cy\bar{c}z$, where $c \in \{a, b\}$. Then $(1, 2 + |y|)$ is a well-formed pair. By the inductive hypothesis, $[1, |y|]$ and $[1, |z|]$ are well-formed matching sets for y and z, respectively. All together gives the well-formed set $[1, n]$ for x, after appropriate translation.

Conversely, assume that $[1, n]$ is a well-formed set for x. Let j_1 be such that $(1, j_1)$ is a (well-formed) matching pair for x. We prove that every matching pair (i, j) for x satisfies: $1 < i, j < j_1$ or $j_1 < i, j \leq n$. Thus the matching set $[1, n]$ is partitioned into $\{1, j_1\}$, $[2, j_1 - 1]$ (which is a translation of the matching set for $x[2, j_1 - 1]$), and $[j_1 + 1, n]$ (which is a translation of the matching set for $x[j_1 + 1, n]$). By the inductive hypothesis, $x[2, j_1 - 1], x[j_1 + 1, n] \in$ Dyck(2), and the statement follows.

We return to the property of matching pairs (i, j) described above. Assume, for a contradiction, that there is a matching pair (i, j) such that $i < j_1 < j$. Then, by Definition 4, $\mathrm{height}(x[1, j_1]) > \mathrm{height}(x[1, i-1])$ and also $\mathrm{height}(x[1, j_1]) = \mathrm{height}(x[1, 0]) = 0$. Thus $\mathrm{height}(x[1, i-1]) < 0$, a contradiction to the first part of the proposition. $\square$

Observe that checking that $[1, n]$ is a (possibly ill-formed) matching set, or equivalently that $\mathrm{height}(x) = 0$ and the height of every prefix of x is nonnegative, can be checked deterministically within one pass over stream x, using $\log n$ memory. Our algorithms do not explicitly check this, but nonetheless ensure this property when accepting x. During the computation our algorithms implicitly keep track of the height of the word read so far. They reject when the height of any prefix is negative, so for ease of exposition, we assume that the height of the stream is always non-negative.

Let p be a prime number such that $n^{1+\gamma} \leq p < 2n^{1+\gamma}$, for some fixed constant $\gamma > 0$. The algorithm uses a random function $\mathrm{hash}(\cdot)$ that maps subwords v of x to integers in $[0, p-1]$, as follows: $\mathrm{hash}(x_{i_1} x_{i_2} \ldots x_{i_m}) = \sum_j \mathrm{hash}(x_{i_j})$, with

$$\mathrm{hash}(x_i) = \begin{cases} \alpha^{\mathrm{height}(x[1,i-1])} \bmod p & \text{if } x_i = a, \\ -\alpha^{\mathrm{height}(x[1,i])} \bmod p & \text{if } x_i = \overline{a}, \\ 0 & \text{otherwise,} \end{cases}$$

where α is a uniformly random integer in $[0, p-1]$. Note that the computation of $\mathrm{hash}(v)$ depends not just on v but also on the height of its letters *within* x.

Given x and v, the value of $\mathrm{hash}(v)$ is a polynomial in α of degree bounded by the maximum height of a prefix of x, which is at most n. A polynomial of degree d over $\mathbb{F}_p$ has at most d roots. Therefore, if the polynomial corresponding to $\mathrm{hash}(v)$ is not identically zero, for a uniformly random α, the probability that $\mathrm{hash}(v) = 0$ is at most $n/p \leq n^{-\gamma}$. In particular:

Proposition 5. *Let $x \in \Sigma^n$ be such that every prefix of x has nonnegative height, and let $v = x_{i_1} x_{i_2} \ldots$ be a subword of x. If $v \in$* Dyck(2), *then* $\mathrm{hash}(v) = 0$ *for all α. Moreover, if there is a height d at which v has a single ill-formed pair (and possibly other ill-formed matching pairs at heights $\neq d$), then* $\mathrm{hash}(v) \neq 0$ *with probability at least $1 - n^{-\gamma}$, for a uniformly random integer $\alpha \in [0, p-1]$.*

Proof. If $v \in \textsc{Dyck}(2)$, then by Proposition 4 the set $[1, n]$ is well-formed, and then each well-formed matching pair (i, j) at height d contributes

$$\begin{cases} \alpha^d - \alpha^d = 0, & \text{if } (x_i, x_j) = (a, \overline{a}); \\ 0 - 0 = 0, & \text{if } (x_i, x_j) = (b, \overline{b}). \end{cases}$$

Therefore, we get $\mathrm{hash}(v) = 0$.

Now, assume there is a height d at which v has a single ill-formed pair. Since every prefix of x has nonnegative height, the value $\mathrm{hash}(v)$ is a polynomial $q(z)$ evaluated at $z = \alpha$. Every well-formed pair at height d cancels, and so the coefficient of z^d in q is $+1$ if $(x_i, x_j) = (a, \overline{b})$, and -1 if $(x_i, x_j) = (b, \overline{a})$. Thus q is not identically zero. The claim follows from the uniformly random choice of α. □

For any letter x_i, we may compute $\mathrm{hash}(x_i)$ in time $\mathrm{polylog}\, n$ and space $\mathrm{O}(\log n)$. Moreover, for any word v the value of $\mathrm{hash}(v)$ can be maintained with $\mathrm{O}(\log n)$ space.

3.1 The one-pass algorithm

The algorithm is easiest to understand if $x = uv$, where u has only upsteps and v has only downsteps, in equal numbers. To check whether $uv \in \textsc{Dyck}(2)$, the naive algorithm would grow a stack of size $n/2$. Here is a simple alternative. We read the input in blocks of length q. For simplicity, assume that n is divisible by $2q$. While the algorithm is reading letters of u, the stack stores the values of $\mathrm{hash}(x[iq + 1, (i + 1)q])$, one stack item for each $i \in \{0, \ldots, n/2q - 1\}$ and notes that $\mathrm{height}(x[iq + 1, (i + 1)q]) = q$. While the algorithm is reading letters of v, it adds $\mathrm{hash}(x[jq + 1, (j + 1)q])$ to $\mathrm{hash}(x[iq + 1, (i + 1)q])$ for $j = n/q - i - 1$, and checks whether their sum is 0. The input x is ill-formed if any of the sums is non-zero. Our algorithm is a generalization of this stack compression idea, and the block length is chosen to be $q = \lceil \sqrt{n \log n}\, \rceil$ to minimize the space used.

Algorithm 1 attempts to collect a sequence of $\ell = \lceil \sqrt{n \log n}\, \rceil$ upsteps while doing obvious checks. Using a straightforward stack-based algorithm, any upstep followed by a downstep is checked for well-formedness, and once checked, the pair are discarded. The stack, called S_{temp} in the algorithm, allows us to apply this check for every matching pair that is encountered before reaching the limit of ℓ upsteps. When the stack S_{temp} collects a sequence v of ℓ upsteps, the algorithm hashes v to $\mathrm{hash}(v)$ and empties S_{temp}. The hash value is pushed to a second stack S. The stack S encodes the subword given by the letters seen so far that remain to be checked. Each item of S of the form (h, ℓ) *encodes* a subword v of the stream x, in the sense that $h = \mathrm{hash}(v)$ and $\ell = \mathrm{height}(v)$. The algorithm accesses S to look up information about the blocks previously read.

To process a downstep y, the algorithm either checks for a match in S_{temp} or incorporates it into the topmost stack item of S. More precisely for the second case, given a downstep y and given $(h, \ell) = (\mathrm{hash}(v), \mathrm{height}(v))$, it computes $\mathrm{hash}(vy) = h + \mathrm{hash}(y)$ and $\mathrm{height}(vy) = \ell - 1$, thus encoding vy without explicit knowledge of v. Note that this relies on the linearity of the hash function. When the encoded subword v has height 0, to test whether it is well-formed, the algorithm checks whether $\mathrm{hash}(v) = 0$. If this test succeeds, the entry of the stack encoding v is removed. An example execution of the algorithm is presented in Figure 1.

For the analysis, we start with the following invariants of **Algorithm 1**.

Proposition 6. *Let (h, ℓ) be an item of S that encodes a subword v. Then $v = v_1 v_2$, where v_1 has only upsteps, v_2 has only downsteps, $\ell = |v_1| - |v_2|$, and $\ell > 0$.*

The proof is by a straightforward induction on the number of operations on S, and is omitted.

We say that the pair of stacks (S, S_{temp}) *encodes* v if $v = v_1 v_2 \ldots v_m v_{\text{temp}}$, where $v_1, v_2, \ldots, v_m$ are the subwords encoded by S (in bottom-up order), and v_{temp} is the sequence of upsteps in S_{temp} (in bottom-up order).

Proposition 7. *Let v be the subword encoded by (S, S_{temp}) just before processing x_j, at line 23, assuming that the algorithm has not already rejected x. Then $v \prec x[1, j - 1]$.*

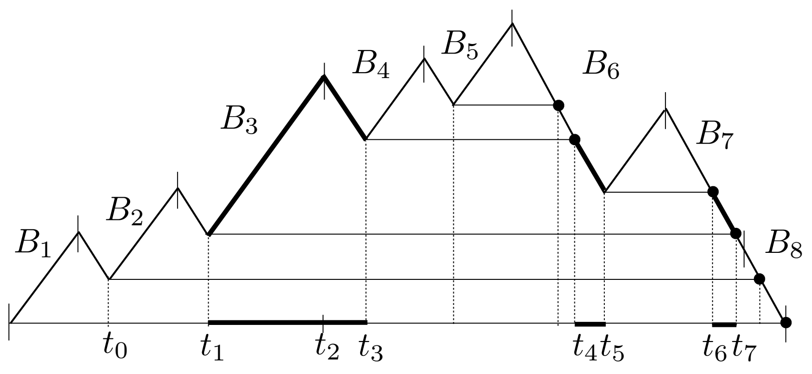

Figure 1: *Example of execution of* ***Algorithm 1****.* Here there are eight blocks, and they are shown after the internal simplifications have already been done. The dotted vertical lines mark times at which the stack changes size, either starting a new stack item (for example, at time t_0) or discarding a stack item (for example, at time t_4). Note that blocks and stack items are staggered: the first item incorporates the first block and the downsteps of the second block, the second item incorporates the upsteps of the second block and the downsteps of the third block, etc. The bullets mark times when the algorithm checks and discards an item, if the hash value is 0. The horizontal lines go from the time when a stack item is created to the time when it is checked and discarded. For example, at time t_7 the algorithm checks and discards an item (h_m, ℓ_m) such that h_m incorporates the upsteps marked in bold on the figure, namely $x(t_1, t_2]$, and incorporates the downsteps marked in bold on the figure, namely $x(t_2, t_3]$, $x(t_4, t_5]$ and $x(t_6, t_7]$.

Algorithm 1 One-pass algorithm (when the length n of the stream is known in advance)

```
1: $S_{\text{temp}} \leftarrow$ empty stack of upsteps; $S \leftarrow$ empty stack of items $(h, \ell)$
2: $(h_{\text{temp}}, \ell_{\text{temp}}) \leftarrow (0, 0)$ {This pair encodes the subword contained in $S_{\text{temp}}$.}
3: Compute a prime $p$ such that $n^{1+\gamma} \le p < 2n^{1+\gamma}$; Pick a uniformly random $\alpha \in [0, p-1]$
   {The pair $(p, \alpha)$ are used in the function hash; $\gamma > 0$ is a constant of our choice.}
4: while stream is not empty do
5:   read next letter $y$ from stream
6:   if $y$ is an upstep then
7:     push $y$ on $S_{\text{temp}}$
8:     update $(h_{\text{temp}}, \ell_{\text{temp}})$ with $y$: $h_{\text{temp}} \leftarrow (h_{\text{temp}} + \text{hash}(y) \mod p)$; $\ell_{\text{temp}} \leftarrow \ell_{\text{temp}} + 1$
9:     if $S_{\text{temp}}$ has size $\lceil \sqrt{n \log n} \rceil$ then
10:      push $(h_{\text{temp}}, \ell_{\text{temp}})$ on to $S$ and reset $S_{\text{temp}}$ to empty; $(h_{\text{temp}}, \ell_{\text{temp}}) \leftarrow (0, 0)$
11:    end if
12:  else {$y$ is a downstep}
13:    if $S_{\text{temp}}$ is not empty then
14:      pop $z$ from $S_{\text{temp}}$
15:      check that $zy$ is well-formed: $zy \in \{a\bar{a}, b\bar{b}\}$ (if not, reject: "mismatch")
16:      update $(h_{\text{temp}}, \ell_{\text{temp}})$ for removal of $z$: $h_{\text{temp}} \leftarrow (h_{\text{temp}} - \text{hash}(z) \mod p)$; $\ell_{\text{temp}} \leftarrow \ell_{\text{temp}} - 1$
17:    else {$S_{\text{temp}}$ is empty}
18:      pop $(h, \ell)$ from $S$ (if empty, reject: "extra closing parenthesis")
19:      update $(h, \ell)$ with $y$: $h \leftarrow (h + \text{hash}(y) \mod p)$; $\ell \leftarrow \ell - 1$
20:      if $\ell = 0$ then check that $h = 0$ (if not, reject: "mismatch")
21:      else push $(h, \ell)$ on $S$
22:    end if
23:  end if
24: end while
25: if $S$ and $S_{\text{temp}}$ are not both empty then reject: "missing closing parenthesis"
26: accept
```

Proof. The proof is by induction on the number of letters $(k-1)$ processed from the stream. Initially, $k=1$ and the statement holds since both stacks are empty. Let $v = v_1v_2\dots v_m v_{\text{temp}}$ be the subword encoded by (S, S_{temp}) just before processing x_k. We assume as our inductive hypothesis that $v \prec x[1,k-1]$, and prove that the analogous statement holds after x_k has been processed.

We have $vx_k \prec x[1,k]$. If x_k is an upstep, then after processing x_k, the stacks are modified such that they encode vx_k. Therefore the propositions still holds just before processing x_{k+1}.

Suppose now that x_k is a downstep. We assume that (S, S_{temp}) are not both empty, otherwise the algorithm rejects at line 18 before processing x_{k+1}. We analyze the processing of x_k in order to complete the induction step.

First, if $v_{\text{temp}} \neq \epsilon$ then the last letter of v is the last letter of v_{temp}, which is an upstep. Therefore $(v_{|v|}, x_{k+1})$ is a matching pair for vx_k, and $v[1,|v|-1] \prec vx_k \prec x[1,k]$. If the algorithm does not reject at this point, the last upstep of v_{temp} is deleted at line 14, and (S, S_{temp}) encodes $v[1,|v|-1]$. So the statement holds in this case.

Second, if $v_{\text{temp}} = \epsilon$ and $\text{height}(v_m x_k) > 0$. The item (h,ℓ) popped from S encodes v_m and satisfies $\text{height}(v_m) = \ell$ from Proposition 6. Therefore $\ell > 1$, and (h,ℓ) is updated in order to encode $v_m x_k$, and then pushed back to S. Now (S, S_{temp}) encodes vx_k which satisfies $vx_k \prec x[1,k]$. So the statement holds in this case as well.

The last case is when $v_{\text{temp}} = \epsilon$ and $\text{height}(v_m x_k) = 0$. By Proposition 6, the item (h,ℓ) popped from S encodes v_m and satisfies $\text{height}(v_m) = \ell = 1$. If the algorithm does not reject after processing x_k, this item is deleted from S and (S, S_{temp}) now encodes $v_1v_2\dots v_{m-1}$. Recall that x_k is a downstep. From Proposition 6, the subword $v_m x_k$ is a sequence of upsteps followed by the same number of downsteps. Therefore $\epsilon \prec v_m x_k$. Since $v_m x_k$ is also the suffix of vx_k, we get that it is a matching set for vx_k, that is $v_1v_2\dots v_{m-1} \prec vx_k \prec x[1,k]$, and the statement holds. □

Lemma 1. ***Algorithm 1*** *satisfies the following invariants:*

1. *At line 15, the pair (z,y) is a matching pair for x.*
2. *At line 20, if $\ell = 0$ then $(h,0)$ encodes a subword v which is a matching set for x.*

Proof. For both properties, let y be the letter x_j that the algorithm is currently processing. Let (S, S_{temp}) be the stacks just before processing x_j, and let v be the subword encoded by (S, S_{temp}). Since we consider properties at line 15 and line 20, x_j is necessarily a downstep.

We start with the first property. Therefore stack S_{temp} is not empty before processing x_j and the upstep z is on its top. Therefore v ends with z, and (z, x_j) is a matching pair for vx_j. By Proposition 7, subword v satisfies $v \prec x[1,j-1]$, so $vx_j \prec x[1,j]$. By Proposition 3, (z,x_j) is also a matching pair for $x[1,j]$, and for x.

For the second property, $S_{\text{temp}} = \emptyset$ and $S \neq \emptyset$. Let v_m be the subword encoded by the topmost element of S. Then $(h,0)$ encodes $v_m x_j$. Moreover, by Proposition 6, $v_m x_j$ is a sequence of upsteps followed by the same number of downsteps. Therefore $v_m x_j$ is a matching set for vx_j. By Proposition 7, subword v satisfies $v \prec x[1,j-1]$, so $vx_j \prec x[1,j]$. By Proposition 3, $v_m x_j$ corresponds to a matching set for $x[1,j]$, and therefore for x. □

Lemma 2. *Let (i,j) be a matching pair for x. Then just before processing x_j, the stacks S and S_{temp} of* ***Algorithm 1*** *satisfy one of the following properties, if the algorithm has not already rejected x:*

1. *S_{temp} is not empty and has x_i on top.*
2. *S_{temp} is empty but not S, and the topmost item of S encodes a subword containing x_i.*

Proof. Fix some matching pair (i,j). Let (S, S_{temp}) be the stacks of the algorithm just before processing x_j. By Proposition 7, the subword encoded by (S, S_{temp}) contains x_i. Therefore the stacks are not both empty.

If $S_{\text{temp}} \neq \emptyset$ just before processing x_j, then, by Lemma 1, x_j matches, possibly as an ill-formed pair, the topmost element of S_{temp}. Since any index (in our case, j) may form a matching pair with at most one other index (in our case, i), the second property is satisfied.

Assume now that $S_{\text{temp}} = \emptyset$ and $S \neq \emptyset$. All upsteps in the stream, including x_i, were first pushed onto S_{temp}, unless the algorithm rejected before reading them. Since S_{temp} is now empty, the upstep x_i was later popped. By Lemma 1, x_i was not popped from S_{temp} at line 15. (Otherwise i would match another index $k \neq j$, which is impossible.) Therefore x_i was encoded into a stack element and pushed on to S at line 10. By Lemma 1 it follows that this stack element was not popped from S at line 20, and therefore is still in S just before x_j is read.

It remains to be proven that the stack element containing x_i is at the top of S. Let $v_1 v_2 \dots v_m$ be the subwords encoded by S. By Propositions 6 and 7, we have $v_1 v_2 \dots v_m x_j \prec x[1, j]$, and $\text{height}(v_k) > 0$, for $k = 1, 2, \dots, m$. Since (x_i, x_j) is a matching pair for both $v_1 v_2 \dots v_m x_j$ and $x[1, j]$, we have that x_i is in v_m. □

We conclude with the correctness of our algorithm.

Theorem 1. ***Algorithm 1*** *is a one-pass randomized streaming algorithm for* Dyck(2) *with space* $\mathrm{O}(\sqrt{n \log n}\,)$ *and time* $\mathrm{polylog}(n)$. *If the stream belongs to* Dyck(2) *then the algorithm accepts it with certainty; otherwise it rejects it with probability at least* $1 - n^{-c}$*, where* $c > 0$ *is a constant.*

Proof. The stack elements of S_{temp} and S take space $\mathrm{O}(1)$ and $\mathrm{O}(\log n)$ bits, respectively. Stack S_{temp} has size bounded by $\lceil \sqrt{n \log n}\, \rceil$, and therefore uses space $\mathrm{O}(\sqrt{n \log n}\,)$. A new element is pushed on to S only when S_{temp} is full (has size $\lceil \sqrt{n \log n}\, \rceil$), after which S_{temp} is emptied. Therefore the algorithm processes at least $\lceil \sqrt{n \log n}\, \rceil$ letters between each increase of the size of S, bounding the number of stack elements of S by $n/\sqrt{n \log n}$. Hence S also uses space $\mathrm{O}(\sqrt{n \log n}\,)$.

Using well-known results in algorithmic number theory [BS96, Sections 8.2 and 9.7], the prime p used for the hash function may be computed probabilistically in time $\mathrm{polylog}(n)$. The probability that the procedure returns a prime is at least $1 - n^{-\gamma}$, for a constant $\gamma > 0$ of our choice. The processing time of any letter in the stream is dominated by the computation of the hash function, specifically by the modular exponentiation. Since the modular exponentiation involves $(\log n)$-bit integers, the time taken is $\mathrm{polylog}(n)$.

With probability $n^{-\gamma}$, the number returned by the prime number generation procedure may be composite. We analyse the algorithm assuming that the number is prime, and then consider the case when it is composite.

To prove correctness, we first argue that the algorithm rejects when $[1, n]$ is not a matching set for x. We prove this property by contraposition. Assume that the algorithm accepts x. Then the stacks S, S_{temp} are both empty after processing x and therefore encode the empty word. By Proposition 7, we have that $\emptyset \prec [1, n]$, i.e., $[1, n]$ is a matching set for x.

We assume in the rest of the proof that $[1, n]$ is a matching set for x. By Proposition 7, S and S_{temp} are never both empty while processing a downstep. The proposition also implies that if the algorithm processes the full stream, then it accepts. Therefore, the algorithm only rejects after processing a downstep either at line 15 or 20. Let x_j be any downstep, and let i be the unique integer such that (i, j) is a matching pair. We prove that the algorithm does not reject after processing x_j if $x \in$ Dyck(2), whereas it rejects with high probability if (i, j) is ill-formed.

Consider first the case $x \in$ Dyck(2). By Lemma 1, the tests at lines 15 and 20 check whether a matching pair or set of x is well-formed. Since x is well-formed, those matching sets are all well-formed. Therefore the tests always succeed (thanks to Proposition 5 for the test at line 20).

We consider the remaining case, where (i, j) is ill-formed and the algorithm has not already rejected before processing x_j. Consider the stacks S and S_{temp} just before x_j is processed. They are not both empty since $[1, n]$ is a matching set. By Lemma 2, the topmost element of S_{temp} is x_i when $S_{\text{temp}} \neq \emptyset$, and the topmost element of S encodes a subword containing x_i when $S_{\text{temp}} = \emptyset$. In the first case, the algorithm checks the well-formedness of (x_i, x_j) at line 15 and rejects with probability 1. In the second case, the algorithm updates the stack element so that it contains both x_i and x_j at line 19. This element is either checked at line 20, or it is pushed back to the stack at line 21. In the latter case, either the algorithm rejects while processing subsequent letters, or eventually checks this stack element at line 20. We consider the first time (if any) that this stack element is checked. Recall that the stack element encodes a subword v that contains x_i and x_j, and that v is a matching set by Lemma 1. The crucial observation is that, by Proposition 6, (i, j)

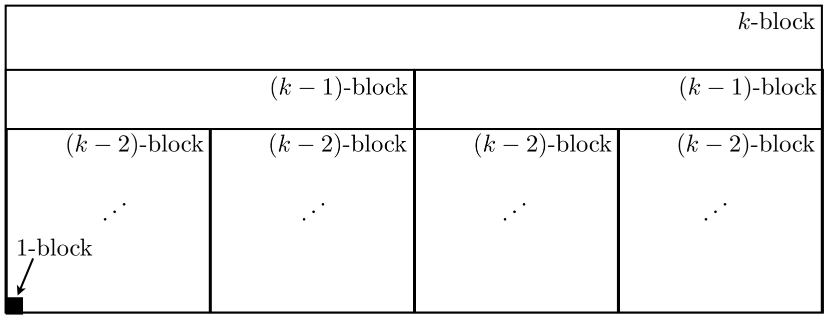

Figure 2: *Block-structure decomposition.* The figure illustrates the binary block decomposition of an input word of length 2^k into all the blocks that will be activated during one full pass pass. They are identical in the left-to-right pass and the right-to-left pass as the input length is a power of 2. At every instant, there is at most one active i-block for any i.

is the only ill-formed matching pair in v at the corresponding height. Therefore Proposition 5 implies that the probability that the algorithm rejects is at least $1 - n^{-\gamma}$.

We point out that the algorithm continues to accept streams $x \in \textsc{Dyck}(2)$ with certainty even if the modulus used in the hash function is composite. When the stream $x \notin \textsc{Dyck}(2)$, the union bound tells us that the probability that the algorithm does not reject is at most $2n^{-\gamma}$. □

3.2 The bidirectional algorithm

The second algorithm uses a (virtual) hierarchical decomposition of the stream x into nested blocks of 2^i letters for $i \le k = \lceil \log n \rceil$ (see Figure 2). We define an *i-block* to be any substring of the form $x[(q-1)2^i + 1, q2^i]$ for $1 \le q \le n/2^i$. We may omit the parameter i when referring to an i-block if its precise value is not important. The algorithm maintains a decomposition of the prefix $x[1,j]$ read so far into $m \le \lceil \log j \rceil$ contiguous blocks of decreasing sizes. The decomposition is given by the binary decomposition of j. Let $0 \le i_1 < \ldots < i_m \le k$ be such that $j = \sum_{t=1}^{m} 2^{i_t}$. Then $x[1,j]$ is partitioned from left to right into adjacent blocks of decreasing lengths $2^{i_m}, 2^{i_{m-1}}, \ldots, 2^{i_1}$. We call such a decomposition the *binary partition* of $x[1,j]$, and the block of size 2^{i_1} the *last block* of the binary partition. We extend the definition and notation related to blocks to intervals $[1,j]$ as well. The binary partition and the last block of an interval $[1,j]$ play an important role in the bidirectional algorithm (see line 13 of **Algorithm 3**).

We assume that $n = 2^k$, for some $k \ge 1$. Thanks to this assumption, the algorithm uses the same hierarchical decomposition whether we read the stream from left to right or from right to left. The assumption is without loss of generality, as we can append to x the word $(a\bar{a})^i$ for a suitable $i \ge 1$. This is only required if $|x|$ is even; otherwise $x \notin \textsc{Dyck}(2)$. At the end of the first pass, we use $\mathrm{O}(\log n)$ bits of memory to store the number of letters added. **Algorithm 2**, the bidirectional algorithm, simply runs **Algorithm 3** twice, once reading the stream in the forward direction, and a second time in reverse. The algorithm accepts if there is no rejection during either pass. During the right to left pass, letters $\bar{a}, \bar{b}$ are interpreted as a, b, respectively (and vice-versa).

Algorithm 2 Bidirectional algorithm (when the length n of the stream is a power of 2, and is known in advance)

```
Compute a prime $p$ such that $n^{1+\gamma} \le p < 2n^{1+\gamma}$; Pick a uniformly random $\alpha \in [0, p-1]$
{The pair $(p, \alpha)$ are used in the function hash; $\gamma > 0$ is a constant of our choice.}
Run **Algorithm 3**, reading the stream from left to right
Run **Algorithm 3**, reading the stream from right to left
{While reading the stream right to left, $\bar{a}, \bar{b}$ are interpreted as $a, b$, respectively (and vice-versa)}
**accept**
```

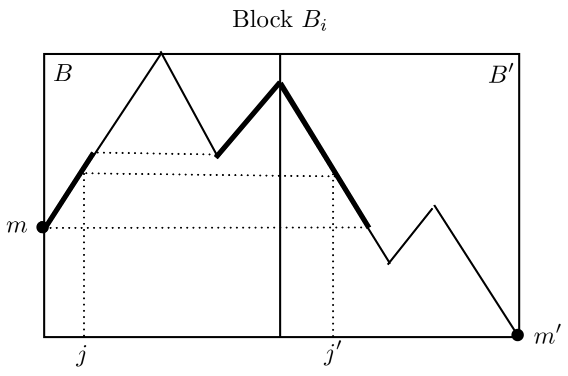

Figure 3: *Asymmetry of the two passes.* The bold-face lines represent matching pairs between the two $(i-1)$-blocks B, B' within the same i-block B_i. In this example, these pairs are checked during the left-to-right pass, since the minimum height m within the left $(i-1)$-block B is larger than the minimum height m' with the right $(i-1)$-block B' (during the right-to-left pass, they are compressed without any checks when B_i is processed).

Algorithm 3 continuously maintains the binary partition of the prefix $x[1, j]$ of the stream that has been read so far. We use a stack data structure to encode the entire prefix $x[1, j]$. Each stack item is now of the form (h, ℓ, f), and encodes a subword v of x, in the sense that $h = \text{hash}(v)$, $\ell = \text{height}(v)$, and f is the position in x of the first letter of v. An item remains in the stack while $\ell > 0$.

The main difference between **Algorithm 3** and **Algorithm 1** is that whenever the algorithm reaches the end of a block, it "compresses" *without checking* the stack items encoding subwords from within the block. This compression is what reduces the stack size from $\sqrt{n/\log n}$ down to $\mathrm{O}(\log n)$, but now Proposition 6 no longer holds for this stack; since hash is commutative, we may lose information. For example, compressing hash($ba\bar{a}$) with hash($b\bar{b}\bar{b}a\bar{a}$) gives hash($ba\bar{a}b\bar{b}\bar{b}a\bar{a}$), which is equal to hash($ba\bar{b}b\bar{a}\bar{b}a\bar{a}$): one word is in Dyck(2), the other one is not, but after compressing we can no longer distinguish between them. In processing the ill-formed word $ba\bar{b}b\bar{a}\bar{b}a\bar{a}$ from left to right, the algorithm compresses the first four letters to hash(ba) and consequently does not detect ill-formedness. The crux of the analysis is that such information loss does not occur both when reading the stream from left to right and when reading it from right to left (see Figure 3). Every matching pair is checked in at least one of the two passes. In the example above, in processing the word $ba\bar{b}b\bar{a}\bar{b}a\bar{a}$ from right to left (with upsteps interpreted as downsteps and vice-versa), a mismatch is detected when the 7th letter is read.

For the analysis of **Algorithm 3**, we first derive the following invariant that is weaker than Proposition 6. The proof follows from induction and is omitted.

Proposition 8. *Let (h, ℓ, f) be an item of S encoding a subword v. Then $\ell = \text{height}(v) > 0$, and every prefix of v has positive height.*

We can adapt Lemmas 1 and 2 to our new algorithm. The proofs are straightforward, and are omitted.

Lemma 3. *At line 10 of **Algorithm 3**, if $\ell = 0$ then $(h, 0, f)$ encodes a subword v that is a matching set for x.*

Lemma 4. *Let (j, j') be a matching pair for x. Then, either **Algorithm 3** rejects before processing $x_{j'}$, or the stack S just before processing $x_{j'}$ is not empty and its topmost item encodes a subword containing x_j.*

We now state a simple observation from the definition of matching pairs. Recall from the convention introduced before Definition 4, that we identify a subword $x_{i_1} x_{i_2} \cdots x_{i_m}$ of x with the set of indices $\{i_1, i_2, \ldots, i_m\}$ corresponding to it.

Proposition 9. *Let $v = uu'$ be a subword of x, and let $d \geq 0$. Then $u \times u'$ has at most one matching pair at height d. In other words, in v there exists at most one matching pair (j, j') at height d such that $j \in u$ and $j' \in u'$.*

Algorithm 3 One pass of the bidirectional algorithm

```
1:  $S \leftarrow$ empty stack of items $(h, \ell, f)$
2:  $j \leftarrow 0$ {This records the length of the stream read so far}
3:  while stream is not empty do
4:      read next letter $y$, and set $j \leftarrow j + 1$
5:      if $y$ is an upstep then
6:          push the item $(\text{hash}(y), 1, j)$ on to $S$ {This encodes the letter $y$}
7:      else {$y$ is a downstep}
8:          pop $(h, \ell, f)$ from $S$ (if empty, reject: "extra closing parenthesis")
9:          update $(h, \ell, f)$ with $h$: $h \leftarrow (h + \text{hash}(y) \mod p)$; $\ell \leftarrow \ell - 1$
10:         if $\ell = 0$ then check that $h = 0$ (if not, reject: "mismatch")
11:         else push $(h, \ell, f)$ on $S$
12:     end if
13:     while the top 2 elements of $S$ both start in the last block of the binary partition of $[1, j]$ do
14:         combine them into one element: pop $(h_2, \ell_2, f_2)$; pop $(h_1, \ell_1, f_1)$; push $(h_1 + h_2, \ell_1 + \ell_2, f_1)$
15:     end while
16: end while
17: if $S$ is not empty then reject: "missing closing parenthesis"
```

Proof. By contradiction, assume that (i, i') and (j, j') are two matching pairs in $u \times u'$ at height d. For simplicity suppose that $i < j$. From the definition of matching pair for (i, i'), we get that $\text{height}(x[1, k]) > \text{height}(x[1, i-1])$, for all $i \leq k < i'$. Since (i, i') and (j, j') are both at height d, indices i, j satisfy $\text{height}(x[1, i-1]) = \text{height}(x[1, j-1]) = d$. Therefore $j > i'$, leading to $j \notin u$, which contradicts that (j, j') is in $u \times u'$. □

We conclude with the correctness of our algorithm.

Theorem 2. *Let $c > 0$.* ***Algorithm 2*** *is a bidirectional two-pass randomized streaming algorithm for* Dyck(2) *with space* $\mathrm{O}((\log n)^2)$ *and time* $\text{polylog}(n)$. *If the input belongs to* Dyck(2) *then the algorithm accepts it with certainty; otherwise it rejects it with probability at least* $1 - n^{-c}$.

Proof. As before, we use well-known results in algorithmic number theory [BS96, Sections 8.2 and 9.7] to compute the prime p for the hash function. This computation is probabilistic, takes time $\text{polylog}(n)$, and space at most $\mathrm{O}(\log^2 n)$. With probability $n^{-\gamma}$, for a constant $\gamma > 0$ of our choice, the number returned may be composite. We first analyze the algorithm assuming the number is prime, and discuss the composite case later.

Each stack element takes space $\mathrm{O}(\log n)$ and the stack has size at most $2k = 2 \log n$, hence space $\mathrm{O}(\log^2 n)$. The processing time is dominated by the computation of the hash function, and the compression of stack elements. Each letter read generates at most one new stack item, after which **Algorithm 3** may combine the elements on top of the stack (at most $\log n$ times). The net time is therefore $\text{polylog}(n)$ per letter.

To analyze the algorithm, observe by induction, and Proposition 4, that the algorithm rejects in either direction with probability 1 if $[1, n]$ is not a matching set. A matching set may be ill-formed, and in the rest of the proof, we focus on proving that the algorithm detects this with high probability.

The above observation implies that we may assume that $[1, n]$ is a matching set. In particular, it implies that S is never empty while processing a downstep. Moreover, if the algorithm processes the full stream in one direction, then the stack is empty at the end and **Algorithm 3** does not reject.

By Lemma 3, each check at line 10 consists of verifying that a matching set is well-formed. Therefore the algorithm always accepts whenever $x \in$ Dyck(2).

Consider now the case when x has an ill-formed matching pair. Let i be minimum number such that some i-block B_i contains an ill-formed matching pair (j, j'). By minimality, x_j and $x_{j'}$ are in different $(i-1)$-blocks B and B'. Let m be the minimum, over upsteps x_l of B, of $\text{height}(x[1, l-1])$. Let m' be the minimum, over downsteps x_l of B', of height($x[1,l]$) (see Figure 3). Up to swapping left-to-right and right-to-left directions, we may assume that $m \geq m'$.

Assume that the algorithm does not reject before processing $x_{j'}$. The stack S is empty since $[1,n]$ is a matching set. Then, by Lemma 4, the topmost element of S encodes a subword containing x_j. Moreover, since all compressions in B involve items with first letter in B, the first letter f of that word is in B, hence starts at height $\geq m$. Since $m \geq m'$, the letter f' matching f is in B_i, and so, from Proposition 8 by the end of reading B' that item is discarded. Let $(h,0,f)$ be that discarded item, encoding a subword v containing both x_j and $x_{j'}$.

Since the first letter f of v is in B, all of the letters of v are in $B \cup B'$. Recall that v is a matching set, and, by Proposition 9, its matching pairs in $B \times B'$ are all at different heights. So, at the height d of pair (j,j'), v only contains (j,j'), which is ill-formed, plus possibly some matching pairs coming from $B \times B$ or from $B' \times B'$, pairs that are all well-formed by minimality of i. Altogether, at height d the word v has exactly one ill-formed matching pair, so by Proposition 5, the probability that v passes the hash test of **Algorithm 3** is at most $n^{-\gamma}$, for a uniformly random choice of α. So the algorithm is correct with probability $1 - n^{-\gamma}$.

The algorithm continues to accept streams $x \in \textsc{Dyck}(2)$ with certainty even if the modulus used in the hash function is composite. When the stream $x \notin \textsc{Dyck}(2)$, the union bound tells us that the probability that the algorithm does not reject is at most $2n^{-\gamma}$. □

4 Lower bounds

In this section, we prove a space lower bound for $\textsc{Dyck}(2)$. We start with a family of hard instances that we embed in a communication problem $\textsc{Ascension}(m)$. A streaming algorithm that uses space σ (a function of m,n) implies a multi-party communication protocol for $\textsc{Ascension}(m)$ with $2m$ players, in which every message has length σ. We then appeal to a direct sum argument to derive a two-party communication protocol for $\textsc{Mountain}$ with "low" information cost. Finally, we show that such a protocol is impossible, unless $\sigma \in \Omega(n)$.

4.1 Reduction from Dyck(2), and an overview

We define the family of hard instances for $\textsc{Dyck}(2)$ as follows. For any word $z \in \{a,b\}^n$, let $\overline{z}$ be the minimal matching word associated with z (so that $z\overline{z}$ is well-formed). For positive integers m,n, consider the following instances of length in $\Theta(mn)$:

$$w \quad = \quad x_1\overline{y}_1\overline{c}_1c_1y_1\ x_2\overline{y}_2\overline{c}_2c_2y_2\ \ldots x_m\overline{y}_m\overline{c}_mc_my_m\ \overline{x}_m\ \ldots\ \overline{x}_2\ \overline{x}_1,$$

where for every i, $x_i \in \{a,b\}^n$, $y_i = x_i[n-k_i+2,n]$ for some $k_i \in \{1,2,\ldots,n\}$, and $c_i \in \{a,b\}$. The word w is in $\textsc{Dyck}(2)$ if and only if, for every i, $c_i = x_i[n-k_i+1]$.

Intuitively, for $m = n/\log n$ recognizing w is difficult with space $o(n)$. After reading x_i, the streaming algorithm does not have enough space to store information about the bit at unknown index $(n-k_i+1)$. When it reads c_i it is therefore unable to decide whether $c_i = x_i[n-k_i+1]$. Moreover, after reading $\overline{y}_m$ it does not have enough space to store information about all indices $k_1,k_2,\ldots,k_m$. When it reads $\overline{x}_m\ \ldots\ \overline{x}_2\ \overline{x}_1$ it therefore misses out on its second chance to check whether $c_i = x_i[n-k_i+1]$ for every i. The formal proof contains several subtleties and is executed in the language of communication complexity.

We define a communication problem $\textsc{Ascension}(m)$ (see Figure 4) associated with the hard instances described above. For convenience, we replace suffixes by prefixes, and identify a with 0 and b with 1. Formally, in the problem $\textsc{Ascension}(m)$ there are $2m$ players $\mathsf{A}_1,\mathsf{A}_2,\ldots,\mathsf{A}_m$ and $\mathsf{B}_1,\mathsf{B}_2,\ldots,\mathsf{B}_m$. Player A_i has $x_i \in \{0,1\}^n$, B_i has $k_i \in [n]$, a bit c_i and the prefix $x_i[1,k_i-1]$ of x_i. Let $\mathbf{x} = (x_1,x_2,\ldots,x_m)$, $\mathbf{k} = (k_1,k_2,\ldots,k_m)$ and $\mathbf{c} = (c_1,c_2,\ldots,c_m)$. The goal is to compute $f_m(\mathbf{x},\mathbf{k},\mathbf{c}) = \bigvee_{i=1}^m f(x_i,k_i,c_i) = \bigvee_{i=1}^m (x_i[k_i] \oplus c_i)$, which is 0 if $x_i[k_i] = c_i$ for all i, and 1 otherwise.

Motivated by the streaming model, we require each message to have length at most σ bits, where the parameter σ is a function of m and n and corresponds to the space used in the streaming algorithm. We also require the communication between the $2m$ participants in a one-pass protocol to be in the following order:

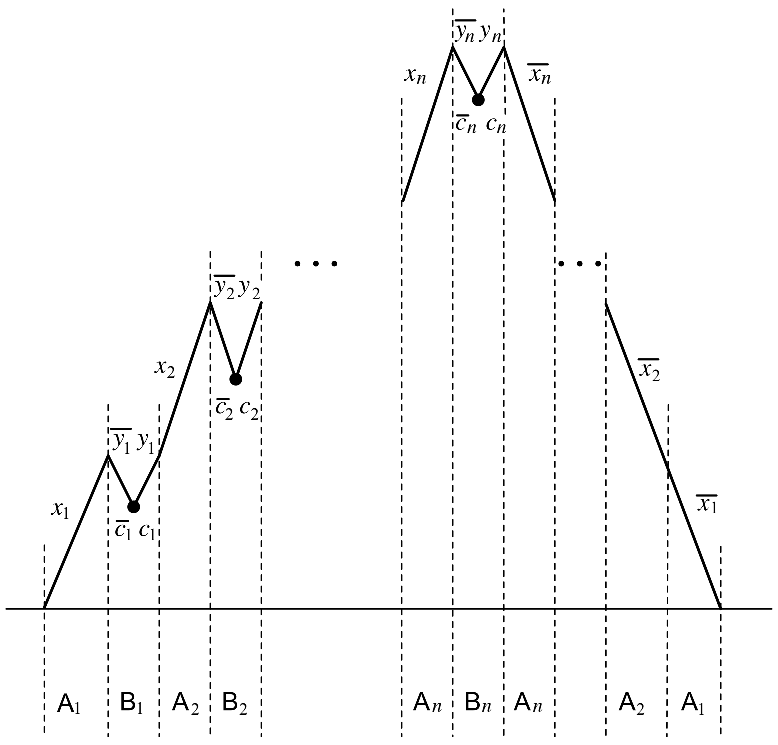

Figure 4: *Problem* ASCENSION(m)*.* The figure presents the m-fold nesting of streams of the form depicted in Figure 5. The stream is divided between $2m$ players. There are m potential mismatches, the ith one caused by the letter c_i in B_i's input. The word is well-formed if and only $c_i = \overline{x}_i[k_i]$, for all i.

Round 1

- For i from 1 to $m-1$, player A_i sends message M_{A_i} to B_i, then B_i sends message M_{B_i} to A_{i+1};
- A_m sends message M_{A_m} to B_m;

Round 2

- B_m sends message M_{B_m} to A_m;
- For i from m down to 2, A_i sends message M'_{A_i} to A_{i-1};
- A_1 computes the output.

A streaming algorithm for DYCK(2) with space 'σ' implies a communication protocol for ASCENSION(m) as described above. So a lower bound on σ follows from a lower bound on the communication complexity of ASCENSION(m).

To establish the hardness of solving ASCENSION(m), we prove a *direct sum* result that captures its relationship to solving m instances of a "primitive" problem MOUNTAIN. In the problem MOUNTAIN (see Figure 5), Alice has an n-bit string $x \in \{0,1\}^n$, and Bob has an integer $k \in [n]$, a bit c and the prefix $x[1, k-1]$ of x. The goal is to compute the Boolean function $f(x,k,c) = (x[k] \oplus c)$ which is 0 if $x[k] = c$, and 1 otherwise. In a one-pass protocol for MOUNTAIN, the communication occurs in the following order: Alice sends a message M_{A} to Bob, Bob sends a message M_{B} to Alice, then Alice outputs $f(x,k,c)$.

As mentioned in Section 1, we follow the "information cost" approach, a method that has been particularly successful in recent works on direct sum results. The method comes in a variety of flavours, each crafted to

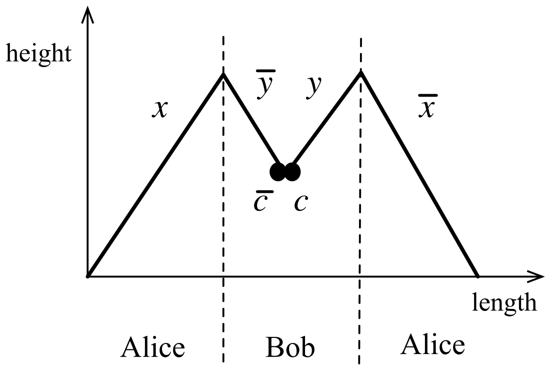

Figure 5: *Problem* MOUNTAIN. The figure presents an input stream with its division between players Alice and Bob. The horizontal axis represents the length of the stream seen so far, and the vertical axis represents the corresponding height. We introduce a potential mismatch denoted by letter c in Bob's input, with $\overline{y}[1,k-1] = \overline{x}[1,k-1]$. Therefore, the word is well-formed if and only if $c = \overline{x}[k]$.

suit the application at hand. We describe the approach as adapted for ASCENSION(m). Information cost is often defined in terms of the entire input and the full transcript of the protocol. We enforce both the nature of streaming algorithms and of our problem, by restricting our attention to only one message M_{B_m} from the transcript. We also split the input in two parts, and measure the information in the message M_{B_m} about one part $(\mathbf{k}, \mathbf{c})$, conditioned on the other part $\mathbf{x}$. In our case, the conditioning corresponds to information that is in the hands of the subsequent players. The closest such measures, of which we are aware, were considered in [JKS03, BBCR13].

The direct sum result is proven using the superadditivity of mutual information for inputs (k_i, c_i) picked independently from a carefully chosen distribution. In the defining information cost, we measure mutual information with respect to a distribution on which the MOUNTAIN function is the constant 0, even though we consider protocols for the problem that are correct with high probability in the worst case (or, equivalently, when the inputs are chosen from a "hard" distribution). The use of this easy distribution collapses the function ASCENSION(m) to an instance of MOUNTAIN in any chosen coordinate. We massage this technique into a form that is better suited to the streaming model and to proving lower bounds for the primitive function MOUNTAIN.

We finish by giving a combinatorial argument that protocols computing MOUNTAIN in the worst case necessarily reveal "a lot" of information even when its inputs are chosen according to the easy distribution. Privacy loss, a measure similar to information cost, has been studied previously in protocols for INDEX (see, e.g., [JRS09, JRS03a] and the references therein). Although this communication problem is closely related to MOUNTAIN, prior works study INDEX under hard distributions, and do not seem to extend directly to our case.

4.2 Information cost

We now implement the program laid out above. We use standard notions from information theory such as Shannon entropy $\mathrm{H}(A)$, mutual information $\mathrm{I}(A : B)$, and their conditional variants $\mathrm{H}(A|C), \mathrm{I}(A : B|C)$, respectively (where A, B, C are jointly distributed random variables). For a primer on these notions and their properties, we refer the reader to the text [CT91].

We measure the *information cost* of a one-pass public-coin randomized protocol P for ASCENSION(m) (of the form described in the previous section), with respect to some distribution ν by $\mathrm{IC}_\nu(P) = \mathrm{I}(\mathbf{K}, \mathbf{C} : M_{\mathsf{B}_m} | \mathbf{X}, R)$, where $(\mathbf{X}, \mathbf{K}, \mathbf{C})$ are inputs drawn from ν, and R denotes the public coins of P. From this we define the *information cost* of the problem ASCENSION(m) itself with respect to a distribution ν and error parameter δ as follows: $\mathrm{IC}^{\mathrm{pub}}_\nu(\text{ASCENSION}(m), \delta) = \min\big(\mathrm{IC}_\nu(P)\big)$, where the minimum is over one-pass public-coin randomized protocols P for the problem, with worst-case error at most δ. Note that the information cost implicitly depends on σ, the length of each message.

For the problem MOUNTAIN we play a subtle game between public and private coins. We consider protocols in which Alice has access only to public coins R, whereas Bob additionally has access to some independent private coins R_{B}. We define $\mathrm{IC}_\nu(P) = \mathrm{I}(K, C : M_{\mathsf{B}} | X, R)$, where R denotes only the public-coins of P. Further, we define $\mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta) = \min\big(\mathrm{IC}_\nu(P)\big)$, where P ranges over "mixed" public and private coin randomized protocols with worst case error at most δ where Alice and Bob share public coins, and only Bob has access to extra private coins.

We also make use of a related measure of complexity for MOUNTAIN when P ranges over protocols where Alice's message is deterministic, and Bob has access to private coins R_{B}: $\mathsf{DIC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \mu, \delta) = \min\big(\mathrm{IC}_\nu(P)\big)$, i.e., the minimum information cost with respect to ν, where P ranges over protocols for MOUNTAIN, in which Alice's message M_{A} is deterministic given her input X, while Bob may use his private coins R_{B} to generate his message. Further, the distributional error of P is at most δ when the inputs are chosen according to μ. Note that in general, and certainly in our application, ν and μ may be different, meaning that we measure the information cost of the protocol with respect to some distribution ν, while we measure its error under a potentially different distribution μ. For later use, we recall that the distributional error under μ is $\mathbb{E}_{(X,K,C)\sim\mu}\big(\Pr(P \text{ fails on } (X,K,C))\big)$, where the probability is over the private coins R_{B} of Bob.

We begin by relating the information cost for protocols in which Alice is deterministic to that of mixed randomized protocols. A similar argument for eliminating public randomness is seen in Ref. [CCM08, Lemma 3.3].

Lemma 5.

$$\mathsf{DIC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \mu, 2\delta) \;\le\; 2 \times \mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta).$$

Proof. Consider a randomized protocol P for MOUNTAIN with worst-case error at most δ such that $\mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta) = \mathrm{IC}_\nu(P)$. We further assume that Alice and Bob have uniformly distributed public coins R, and only Bob has extra private coins R_{B}. Then

$$\mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta) \;=\; \mathop{\mathbb{E}}_r \big(\mathrm{I}(K, C : M_{\mathsf{B}_m} | X, R = r)\big),$$

Since P has worst-case error at most δ, it has distributional error at most δ under μ:

$$\mathop{\mathbb{E}}_r \Big(\mathop{\mathbb{E}}_{(X,K,C)\sim\mu} \big(\Pr(P \text{ fails on } (X,K,C) | R = r)\big)\Big) \;\le\; \delta.$$

Therefore, by the Markov inequality, there is a set $\mathcal{R}$ with $\Pr(R \in \mathcal{R}) \ge \frac{1}{2}$ such that

$$\forall r \in \mathcal{R}, \quad \mathop{\mathbb{E}}_{(X,K,C)\sim\mu} \big(\Pr(P \text{ fails on } (X,K,C) | R = r)\big) \;\le\; 2\delta.$$

Now consider the information cost of P under the distribution ν over inputs. Let $\mathrm{U}(\mathcal{R})$ denote the uniform distribution on $\mathcal{R}$. We have

$$\mathop{\mathbb{E}}_{r\sim \mathrm{U}(\mathcal{R})} \big(\mathrm{I}(k, c : M_{\mathsf{B}_m} | X, R = r)\big) \;\le\; 2 \times \mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta),$$

since the event $\mathcal{R}$ has probability at least $1/2$. Therefore, there exists an $r \in \mathcal{R}$ such that $\mathrm{I}(K, C : M_{\mathsf{B}_m} | X, R = r) \le 2 \times \mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta)$. Let P_r be the protocol obtained by fixing the public coins used in P to r. Then Alice's message M_{A} is deterministic. By definition of $\mathcal{R}$, the protocol P_r has distributional error at most 2δ under μ, and $\mathrm{IC}_\nu(P_r) \le 2 \times \mathrm{IC}^{\mathrm{mix}}_\nu(\textsc{Mountain}, \delta)$. $\square$

4.3 Information cost of Mountain

As explained before, and formally proved in the next section, the information cost approach entails showing that the Mountain problem is "hard" even when we restrict our attention to an easy distribution. We prove such a result here.

Let μ be the distribution over inputs (x, k, c) in which X is a uniformly random n-bit string, K is a uniformly random integer in $[n]$ and C a uniformly random bit. This is a hard distribution for Mountain (as is implicit in [Nay99, ANTV02]). We consider the information cost of Mountain under the distribution μ_0 obtained by conditioning μ on the event that the function value is 0: $\mu_0(x, k, c) = \mu(x, k, c | X[K] = C)$.

Lemma 6. *If $\sigma \leq n/100$, then*

$$\mathsf{DIC}^{\text{mix}}_{\mu_0}(\textsc{Mountain}, \mu, 1/16n^2) \;\in\; \Omega(\log n).$$

Proof. Let P be a randomized protocol for Mountain, where Alice's message M_{A} is deterministic, with distributional error at most $1/16n^2$ under the distribution μ, such that $|M_{\mathsf{A}}| \leq n/100$. We prove that $\mathrm{IC}_{\mu_0}(P) \in \Omega(\log n)$. In the following, all expressions involving mutual information and entropy are with respect to the distribution μ_0.

By the Markov inequality, there are at least 2^{n-1} strings u on which P fails with error at most $1/8n^2$ on average on input (u, K, C), where (K, C) are uniformly distributed. Let $S \subseteq \{0, 1\}^n$ of size at least 2^{n-1} be the set of such strings u. When $u \in S$, the protocol P has error probability at most $1/4n$ on input (u, k, c), for every (k, c).

Let α be a possible message M_{A} from Alice to Bob when her inputs range in S, and let $S_\alpha = \{u \in S : M_{\mathsf{A}}(u) = \alpha\}$. For every string $v \in S_\alpha$, we bound from below the mutual information of K and M_{B}, the randomized message that Bob sends back to Alice. For this we construct a set $J_v \subseteq [n]$ such that the message distributions $M_k \overset{\text{def}}{=} M_{\mathsf{B}}(\alpha, v[1, k-1], k, v[k])$ for $k \in J_v$ are pairwise well-separated in ℓ_1 distance. This is in turn established by exhibiting, for each $k \in J_v$, a string $u_k \in S_\alpha$ such that $u_k[1, k-1] = v[1, k-1]$ and $u_k[k] \neq v[k]$. The details follow.

Associate with S_α its dictionary T, a 2-rank tree (a tree with either 1 or 2 children at any internal node), all of whose nodes except the root are labeled by bits; the root has an empty label. Each string v in S_α is in one-to-one correspondence with a top-down path π in T from the root to one of its leaves, where the label of the $(i+1)$th node in π is $v[i]$. We identify $v \in S_\alpha$ with the path π in T, and refer to this path as v.

The tree T has $|S_\alpha|$ leaves, each at depth n. For a fixed $v \in S_\alpha$, let J_v be the set of integers k such that the $(k+1)$th node in path v has out-degree 2. By construction, for every $k \in J_v$ there exists another string, say, $u_k \in S_\alpha$ such that $u_k[1, k-1] = v[1, k-1]$ and $u_k[k] \neq v[k]$. Set $c_k = v[k]$ for every $k \in [n]$. Then the message distributions satisfy $M_{\mathsf{B}}(\alpha, v[1, k-1], k, c_k) = M_{\mathsf{B}}(\alpha, u_k[1, k-1], k, c_k)$, for all $k \in J_v$. Let $M_k = M_{\mathsf{B}}(\alpha, v[1, k-1], k, c_k)$. Let $k, k' \in J_v$ be distinct indices such that $k < k'$. As $u_{k'}[1, k'-1] = v[1, k'-1]$, the message distribution $M_{\mathsf{B}}(\alpha, u_{k'}[1, k-1], k, c_k)$ on input $(u_{k'}, k, c_k)$ equals M_k, and also $M_{\mathsf{B}}(\alpha, u_{k'}[1, k'-1], k', c_{k'})$ on input $(u_{k'}, k', c_{k'})$ equals $M_{k'}$. However, $u_{k'}[k] = v[k] = c_k$, so the function evaluates to 0 on input $(u_{k'}, k, c_k)$, and $u_{k'}[k'] \neq v[k'] = c_{k'}$, so the function value is 1 on $(u_{k'}, k', c_{k'})$. The protocol P computes its outputs from $M_k, u_{k'}$ and $M_{k'}, u_{k'}$, respectively, on these instances, and errs with probability at most $1/4n$.

We use the above property of the distributions $\{M_k\}$ to bound from below the mutual information of K and the message M_{B}, given v.

Proposition 10.

$$\mathrm{I}(K : M_{\mathsf{B}} | X = v) \;\geq\; \left(\frac{4|J_v| - n}{4n}\right) \log n - 2.$$

(We prove this below.)

Next, we observe from the properties of 2-rank trees that the number of strings $v \in S_\alpha$ for which $|J_v| = l$ is at most 2^l. The number of v for which $|J_v| \leq l - 2$ is therefore at most 2^{l-1}. Now fix $l = \log |S_\alpha|$, and note that the proportion of $v \in S_\alpha$ with $|J_v| \geq l - 1$ is at least $1/2$. Therefore $\mathbb{E}_{v \sim \mathrm{U}(S_\alpha)} |J_v| \quad \geq \quad \frac{l-1}{2}$.

We now concentrate on the messages α such that $\Pr_{X \text{ uniform}}(M_{\mathsf{A}}(X) = \alpha | X \in S) \geq 2^{-n/10}$. Then $l = \log |S_\alpha| \geq n - 1 - n/10 = 0.9n - 1$, and by Proposition 10 for $n \geq 3$,

$$\begin{aligned}
\mathop{\mathbb{E}}_{V \sim \mathrm{U}(S_\alpha)}(\mathrm{I}(K, C : M_{\mathsf{B}} | X = V)) &\geq \left[\frac{1}{n} \mathop{\mathbb{E}}_{V \sim \mathrm{U}(S_\alpha)} |J_V| - \frac{1}{4}\right] \log n - 2 \\
&\geq \left[\frac{l-1}{2n} - \frac{1}{4}\right] \log n - 2 \\
&\geq \left[\frac{0.9n - 2}{2n} - \frac{1}{4}\right] \log n - 2 \geq \frac{1}{10} \log n - 2.
\end{aligned}$$

Consider the set $\mathcal{A}$ of messages α which have probability at most $2^{-n/10}$ given $X \in S$. These messages occur with probability at most $2^{n/100} 2^{-n/10} = 2^{-9n/10}$, which is negligible. Therefore we conclude that $\mathrm{I}(K, C : M_{\mathsf{B}} | X) \in \Omega(\log n)$. □

Proof of Proposition 10. Fix a string v, and the corresponding set of indices J_v. Suppose we are given as input a distribution $M = M_k$, for some $k \in J_v$. We recover k using the following procedure Π:

1. For each $k' \in J_v$, simulate Alice's computation of the output in the protocol P, by setting $M_{\mathsf{B}} = M$, the distribution given as input to Π, and $X = u_{k'}$.
2. Let $(D_{k'})_{k' \in J_v}$ be the sequence of outputs Alice generates from the above simulation. Output the largest k' for which $D_{k'} = 1$. This is our guess for k.

On input M_k, the procedure Π above generates $D_k = 1$, and $D_{k'} = 0$ for $k' > k$, each with probability at least $1 - 1/4n$ for any fixed $k' \geq k$. Therefore, the procedure outputs k with probablity at least $3/4$.

We now argue that the entropy of K is significantly reduced when given $M_{\mathsf{B}}, X = v$, under the distribution μ_0 (i.e., when $c_k = v[k]$). This is equivalent to saying that the mutual information of k and M_{B} is high. When the inputs are picked according to the distribution μ_0, we have

$$\begin{aligned}
\mathrm{I}(K, C : M_{\mathsf{B}} | X = v) &= \mathrm{H}(K | X = v) - \mathrm{H}(K | M_{\mathsf{B}}, X = v) \\
&= \log n - \mathrm{H}(K | M_{\mathsf{B}}, X = v).
\end{aligned}$$

We bound from above the conditional entropy $\mathrm{H}(K | M_{\mathsf{B}}, X = v)$. We first separate the values of $k \notin J_v$ as follows. Let $p = |J_v|/n$, and define the Boolean random variable L as 1 iff $K \in J_v$. We have

$$\begin{aligned}
&\mathrm{H}(K | M_{\mathsf{B}}, X = v) \\
&\quad = \mathrm{H}(KL | M_{\mathsf{B}}, X = v) \\
&\quad = \mathrm{H}(L | M_{\mathsf{B}}, X = v) + \mathrm{H}(K | M_{\mathsf{B}}, X = v, L) \\
&\quad = \mathrm{H}(p) + (1-p)\mathrm{H}(K | M_{\mathsf{B}}, X = v, K \notin J_v) \\
&\qquad\quad + p\,\mathrm{H}(K | M_{\mathsf{B}}, X = v, K \in J_v) \\
&\quad \leq 1 + (1-p)\log n + \mathrm{H}(K | M_{\mathsf{B}}, X = v, K \in J_v) \\
&\quad \leq 1 + (1-p)\log n + \mathrm{H}(K | K_\Pi, X = v, k \in J_v),
\end{aligned}$$

where K_Π is the random variable output by the procedure Π. The second equality follows from the Chain Rule for entropy [CT91, Theorem 2.2.1, p. 16], and the final step follows from the Data Processing Inequality [CT91, Theorem 2.8.1, p. 32]. For any fixed $k \in J_v$, given M_k the procedure Π computes $K_\Pi = k$ with probability at least $3/4$. By the Fano Inequality [CT91, Theorem 2.11.1, p. 39], we have

$$\begin{aligned}
\mathrm{H}(K | K_\Pi, X = v, K \in J_v) &\leq \mathrm{H}\left(\frac{1}{4}\right) + \frac{1}{4}\log(|J_v| - 1) \\
&\leq 1 + \frac{1}{4}\log n.
\end{aligned}$$

□

By combining Lemmas 5 and 6 we get

Theorem 3.

$$\mathrm{IC}^{\mathrm{mix}}_{\mu_0}(\textsc{Mountain}, 1/32n^2) \;\in\; \Omega(\log n).$$

4.4 Reduction from Ascension to Mountain

We now study the information cost of $\textsc{Ascension}(m)$ for the distribution μ_0^m over $(\{0,1\}^n \times [n] \times \{0,1\})^m$ for the inputs $\mathbf{x} = (x_1, x_2, \dots, x_m)$, $\mathbf{k} = (k_1, k_2, \dots, k_m)$ and $\mathbf{c} = (c_1, c_2, \dots, c_m)$. We state a direct sum property that relates this cost to that of one instance of $\textsc{Mountain}$, and then conclude.

Lemma 7.

$$\mathrm{IC}^{\mathrm{pub}}_{\mu_0^m}(\textsc{Ascension}(m), \delta) \;\geq\; m \times \mathrm{IC}^{\mathrm{mix}}_{\mu_0}(\textsc{Mountain}, \delta).$$

Proof. Let P be a public-coin randomized protocol for $\textsc{Ascension}(m)$ with worst-case error δ such that $\mathrm{IC}_{\mu_0^m}(P) = \mathrm{IC}^{\mathrm{pub}}_{\mu_0^m}(\textsc{Ascension}(m), \delta)$.

From P, we construct the following protocol P'_j for $\textsc{Mountain}$, where $j \in [n]$. Let (x, k, c) be the input for $\textsc{Mountain}$.

1. Alice sets A_j's input x_j to her input x.
2. Bob sets B_j's input $(k_j, x_j[1, k_j - 1], c_j)$ to his input $(k, x[1, k-1], c)$.
3. Alice and Bob generate, using public coins, (X_i, K_i, C_i) according to μ_0, independently for all $i < j$, and X_i uniformly independently for $i > j$.
4. Bob generates (K_i) uniformly independently for $i > j$, but using his private coins. Then Bob sets $C_i = X_i[K_i]$ for $i > j$ (so that (X_i, K_i, C_i) are distributed according to μ_0, independently for all $i > j$).
5. Alice and Bob run the protocol P by simulating the players $(\mathsf{A}_i, \mathsf{B}_i)_{i=1}^m$ as follows:
   (a) Alice runs P until she generates the message M_{A_j} from player A_j. She sends this message to Bob.
   (b) Bob continues running P until he generates the message M_{B_m} from player B_m. He sends this message to Alice.
   (c) Alice completes the rest of the protocol P until the end, and produces as output for P'_j, the output of player A_1 in P.

By definition of the distribution μ_0, we have $f(X_i, K_i, C_i) = 0$ for all $i \neq j$. So $f_m(\mathbf{X}, \mathbf{K}, \mathbf{C}) = f(x, k, c)$, and each protocol P'_j computes the function f, i.e., solves $\textsc{Mountain}$, with worst-case error δ.

We prove that $\mathrm{IC}_{\mu_0^m}(P) = \sum_j \mathrm{IC}_{\mu_0}(P'_j)$, which implies the result, since only Bob uses private coins in P'_j.

Let R denote the public coins used in the protocol P. By applying the Chain Rule [CT91, Theorem 2.5.2, p. 22] to $\mathrm{IC}_{\mu_0^m}(P)$, we get

$$\begin{aligned}\mathrm{IC}_{\mu_0^m}(P) &= \mathrm{I}(\mathbf{K}, \mathbf{C} : M_{\mathsf{B}_m} | \mathbf{X}, R) \\ &= \sum_j \mathrm{I}(K_j, C_j : M_{\mathsf{B}_m} | \mathbf{X}, K_1, C_1, \dots, K_{j-1}, C_{j-1}, R)\end{aligned}$$

Let $R_j = (R, (X_i)_{j \neq i}, (K_i, C_i)_{i<j})$. These are all the public random coins used in the protocol P'_j, and any further random coins $(K_i, C_i)_{i>j}$ are used only by Bob. Since for all j

$$\mathrm{IC}_{\mu_0}(P'_j) \quad = \quad \mathrm{I}(K_j, C_j : M_{\mathsf{B}_m} | X_j, R_j) \ ,$$

which is the same as

$$\mathrm{I}(K_j, C_j : M_{\mathsf{B}_m} | \mathbf{X}, K_1, C_1, \dots, K_{j-1}, C_{j-1}, R) \ ,$$

the direct sum result follows. □

We can now conclude a lower bound for $\textsc{Ascension}(m)$.

Theorem 4. *Let P be a public-coin randomized protocol for* $\textsc{Ascension}(n/\log n)$ *with worst-case error probability $1/32n^2$, then $\sigma \in \Omega(n)$.*

Proof. Let $m = n/\log n$ and $\delta = 1/32n^2$, and let P be a public-coin randomized protocol for $\textsc{Ascension}(m)$ with worst-case error probability δ. $\mathrm{IC}_{\mu_0^m}(P)$ is at most σ, and by definition $\mathrm{IC}^{\mathrm{pub}}_{\mu_0^m}(\textsc{Ascension}(m), \delta)$ is less than or equal to $\mathrm{IC}_{\mu_0^m}(P)$. By Lemma 7, we have $\mathrm{IC}^{\mathrm{pub}}_{\mu_0^m}(\textsc{Ascension}(m), \delta) \geq m \times \mathrm{IC}^{\mathrm{mix}}_{\mu_0}(\textsc{Mountain}, \delta)$. By Theorem 3, we get $\mathrm{IC}^{\mathrm{mix}}_{\mu_0}(\textsc{Mountain}, \delta) \in \Omega(\log n)$. Combining yields $\sigma \in \Omega(m \log n) \in \Omega(n)$. $\square$

Corollary 1. *Every one-pass randomized streaming algorithm for* $\textsc{Dyck}(2)$ *with (two-sided) error* $\mathrm{O}(1/n' \log n')$ *uses* $\Omega(\sqrt{n' \log n'})$ *space, where n' is the input length.*

Proof. Assume we have a one-pass randomized streaming algorithm for $\textsc{Dyck}(2)$ with (two-sided) error $\mathrm{O}(1/n' \log n')$ uses space σ, where n' is the input length. Then, by the discussion at the beginning of Section 4, there is a public-coin randomized protocol for $\textsc{Ascension}(n/\log n)$ with $n \in \Theta(\sqrt{n' \log n'})$ and with worst-case error probability $1/32n^2$. By Theorem 4, the messages have length $\Omega(n)$, and therefore, the streaming algorithm has space $\Omega(n) = \Omega(\sqrt{n' \log n'}\,)$. $\square$

5 Concluding remarks

Existing computing infrastructure typically supports unidirectional streams. A question that naturally arises from our work is whether we can achieve the performance of the bidirectional algorithm in Theorem 2 by making multiple passes in the same direction. Two sets of authors, Chakrabarti, Cormode, Kondapally, and McGregor [CCKM10, CCKM13], and Jain and Nayak [JN10] independently and concurrently proved that allowing a larger constant number of passes in the same direction does not help. More precisely, they show that for any $T \geq 1$, any unidirectional randomized T-pass streaming algorithm that recognizes length n instances of $\textsc{Dyck}(2)$ with a constant probability of error uses space $\Omega(\sqrt{n}/T)$. The lower bound in both works goes via an extension of the reduction from $\textsc{Ascension}(m)$ to $\textsc{Mountain}$ (cf. Section 4.4). When specialized to one-pass algorithms, the above gives us a bound that is factor of $\sqrt{\log n}$ better than the one in Corollary 1 for constant error probability. However, it falls short of optimal (by the same factor) for polynomially small error.

A number of later works have explored applications of the fingerprinting technique in streaming algorithms, the relationship of formal language theory to streaming, or the advantage of bidirectional streams over unidirectional ones. Chakrabarti *et al.* [CCKM10, CCKM13] use fingerprinting in a one-pass streaming algorithm for checking priority queues. They also observe that the algorithms in this article extend to checking stacks, queues, and double-ended queues. Konrad and Magniez [KM12] show a qualitatively similar dichotomy between one-pass and bidrectional two-pass algorithms as in this article, for validating XML documents. In addition, they present an algorithm when access to external memory is available. The multipass lower bound for $\textsc{Dyck}(2)$ described above [CCKM10, JN10] extends to the problem of checking priority queues. François and Magniez prove a lower bound of $\Omega(\sqrt{n}/T)$ for this problem even in the presence of timestamps, with T passes in the same direction. They complement this with a polylogarithmic space bidirectional algorithm, thus providing another example of a language for which bidirectional streams are exponentially more powerful than unidirectional ones. Krebs, Limaye, and Srinivasan [KLS11] give streaming algorithms for nearly well-parenthesized "one-turn" expressions and Babu, Limaye, and Varma [BLV10] (see also [BLRV13]) study the streaming complexity of subclasses of context-free languages. We expect the ideas in this article to have further such ramifications.

References

[AKNS01] Noga Alon, Michael Krivelich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[ANTV99] Andris Ambainis, Ashwin Nayak, Amnon Ta-Shma, and Umesh Vazirani. Dense quantum coding and a lower bound for 1-way quantum automata. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 376–383. ACM Press, May 1–4, 1999.

[ANTV02] Andris Ambainis, Ashwin Nayak, Amnon Ta-Shma, and Umesh Vazirani. Dense quantum coding and quantum finite automata. *Journal of the ACM*, 49(4):1–16, July 2002.

[BBCR13] Boaz Barak, Mark Braverman, Xi Chen, and Anup Rao. How to compress interactive communication. *SIAM Journal on Computing*, 42(3):1327–1363, 2013.

[BJKS04] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004. Special issue on FOCS 2002.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[BLR93] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993.

[BLRV13] Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. *Theoretical Computer Science*, 494:13–23, 2013.

[BLV10] Ajesh Babu, Nutan Limaye, and Girish Varma. Streaming algorithms for some problems in log-space. In Jan Kratochvíl, Angsheng Li, Jiří Fiala, and Petr Kolman, editors, *Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010, Proceedings*, volume 6108 of *Lecture Notes in Computer Science*, pages 94–104. Springer Berlin Heidelberg, 2010.

[BS96] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Vol. 1: Efficient Algorithms.* MIT Press, Cambridge, Massachusetts, USA, 1996.

[CCKM10] Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for Augmented Index and streaming language recognition. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 387–396, Washington, DC, USA, 2010. IEEE Computer Society.

[CCKM13] Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM Journal on Computing*, 42(1):61–83, 2013.

[CCM08] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 641–650, New York, NY, USA, 2008. ACM.

[CKM07] Matthew Chu, Sampath Kannan, and Andrew McGregor. Checking and spot-checking the correctness of priority queues. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wrocaw, Poland, July 9-13, 2007. Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 728–739. Springer Berlin Heidelberg, 2007.

[CS63] Noam Chomsky and Marcel-Paul Schützenberger. Computer programming and formal languages. In P. Braffort and D. Hirschberg, editors, *The Algebraic Theory of Context-Free Languages*, pages 118–161, 1963.

[CSWY01] Amit Chakrabarti, Yaoyun Shi, Anthony Wirth, and Andrew C.-C. Yao. Informational complexity and the direct sum problem for simultaneous message complexity. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 270–278, 2001.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA, 1991.

[DBIPW10] Khanh Do Ba, Piotr Indyk, Eric Price, and David P. Woodruff. Lower bounds for sparse recovery. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1190–1197, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[FKSV02] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. Testing and spot-checking of data streams. *Algorithmica*, 34(1):67–80, 2002.

[GGR98] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, July 1998.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[JKS03] T. S. Jayram, Ravi Kumar, and D. Sivakumar. Two applications of information complexity. In *Proceedings of the Thirty-Fifth annual ACM Symposium on Theory of Computing*, pages 673–682. ACM, 2003.

[JN10] Rahul Jain and Ashwin Nayak. The space complexity of recognizing well-parenthesized expressions. Technical Report TR10-071, Electronic Colloquium on Computational Complexity, `http://eccc.hpi-web.de/`, 2010.

[JRS03a] Rahul Jain, Jaikumar Radhakrishnan, and Pranab Sen. A direct sum theorem in communication complexity via message compression. In Jos C.M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of the Thirtieth International Colloquium on Automata Languages and Programming*, volume 2719 of *Lecture notes in Computer Science*, pages 300–315. Springer, Berlin/Heidelberg, 2003.

[JRS03b] Rahul Jain, Jaikumar Radhakrishnan, and Pranab Sen. A lower bound for the bounded round quantum communication complexity of Set Disjointness. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 220–229. IEEE Computer Society Press, Los Alamitos, CA, USA, 2003.

[JRS09] Rahul Jain, Jaikumar Radhakrishnan, and Pranab Sen. A property of quantum relative entropy with an application to privacy in quantum communication. *Journal of the ACM*, 56(6):1–32, 2009.

[KLS11] Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In Filip Murlak and Piotr Sankowski, editors, *Mathematical Foundations of Computer Science 2011, 36th International Symposium, Proceedings*, volume 6907 of *Lecture Notes in Computer Science*, pages 412–423. Springer Berlin Heidelberg, 2011.

[KM12] Christian Konrad and Frédéric Magniez. Validating XML documents in the streaming model with external memory. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 34–45, New York, NY, USA, 2012. ACM.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1161–1178, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[LZ77] Richard J. Lipton and Yechezkel Zalcstein. Word problems solvable in logspace. *Journal of the ACM*, 24(3):522–526, July 1977.

[MMN10] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing*, pages 261–270, New York, NY, June 6–8 2010. ACM Press.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*, volume 1, number 2 of *Foundations and Trends in Theoretical Computer Science*. Now Publishers Inc., Hanover, MA, USA, 2005.

[Nay99] Ashwin Nayak. Optimal lower bounds for quantum automata and random access codes. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 369–376. IEEE Computer Society Press, October 17–19, 1999.

[PRR03] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Structures and Algorithms*, 22(1):98–138, January 2003.

[SS02] Michael Saks and Xiaodong Sun. Space lower bounds for distance approximation in the data stream model. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 360–369. ACM, 2002.

[SS07] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In Thomas Schwentick and Dan Suciu, editors, *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007. Proceedings*, volume 4353 of *Lecture Notes in Computer Science*, pages 299–313. Springer Berlin Heidelberg, 2007.

[SV02] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 53–64, New York, NY, USA, 2002. ACM.