Space Complexity vs. Query Complexity*

Oded Lachish † Ilan Newman ‡ Asaf Shapira §

Abstract

Combinatorial property testing deals with the following relaxation of decision problems: Given a fixed property and an input x, one wants to decide whether x satisfies the property or is "far" from satisfying it. The main focus of property testing is in identifying large families of properties that can be tested with a certain number of queries to the input. Unfortunately, there are nearly no general results connecting standard complexity measures of languages with the hardness of testing them. In this paper we study the relation between the space complexity of a language and its query complexity. Our main result is that for any space complexity $s(n) \leq \log n$ there is a language with space complexity $O(s(n))$ and query complexity $2^{\Omega(s(n))}$. We conjecture that this exponential lower bound is best possible, namely that the query complexity of a languages is at most exponential in its space complexity.

Our result has implications with respect to testing languages accepted by certain restricted machines. Alon et al. [FOCS 1999] have shown that any regular language is testable with a constant number of queries. It is well known that any language in space $o(\log\log n)$ is regular, thus implying that such languages can be so tested. It was previously known that there are languages in space $O(\log n)$ that are not testable with a constant number of queries and Newman [FOCS 2000] raised the question of closing the exponential gap between these two results. A special case of our main result resolves this problem as it implies that there is a language in space $O(\log\log n)$ that is not testable with a constant number of queries, thus showing that the $o(\log\log n)$ bound is best possible. It was also previously known that the class of testable properties cannot be extended to all context-free languages. We further show that one cannot even extend the family of testable languages to the class of languages accepted by single counter machines which is perhaps the weakest (uniform) computational model that is strictly stronger than finite state automata.

1 Introduction

1.1 Basic Definitions

Combinatorial property testing deals with the following relaxation of decision problems: for a fixed property $\mathcal{P}$, given an input x, one wants to decide whether x satisfies $\mathcal{P}$ or is "far" from satisfying

*A preliminary version of this paper appeared in the Proc. of the 10^{th} International Workshop on Randomization and Computation (RANDOM), 2006, 426-437.

†University of Haifa, Haifa, Israel, loded@cs.haifa.ac.il.

‡University of Haifa, Haifa, Israel, ilan@cs.haifa.ac.il. Research was supported by the Israel Science Foundation (grant number 55/03)

§Microsoft Research, Email: asafico@tau.ac.il. Part of this work was done while the author was a PhD student at the School of Computer Science, Tel Aviv University, Tel Aviv, Israel.

the property. This notion was first introduced in the work of Blum, Luby and Rubinfeld [5], and was explicitly formulated for the first time by Rubinfeld and Sudan [18]. Goldreich, Goldwasser and Ron [9] have started a rigorous study of what later became known as "combinatorial property testing". Since then much work has been done, both on designing efficient algorithms for specific properties, and on identifying natural classes of properties that are efficiently testable. For detailed surveys on the subject see [6, 8, 16, 17].

In this paper we focus on testing properties of strings, or equivalently languages [1]. In this case a string of length n is ϵ-far from satisfying a property $\mathcal{P}$ if at least ϵn of the string's entries should be modified in order to get a string satisfying $\mathcal{P}$. An ϵ-tester for $\mathcal{P}$ is a randomized algorithm that given ϵ and the ability to query the entries of an input string, can distinguish with high probability (say 2/3) between strings satisfying $\mathcal{P}$ and those that are ϵ-far from satisfying it. The query complexity $q(\epsilon, n)$ is the maximum number of queries the algorithm makes on any input of length n. Property $\mathcal{P}$ is said to be testable with a *constant number of queries* if $q(\epsilon, n)$ can be bounded from above by a function of ϵ only. For the sake of brevity, we will sometimes say that a language is *easily testable* if it can be tested with a constant number of queries [2].

If a tester accepts with probability 1 inputs satisfying $\mathcal{P}$ then it is said to have a 1-sided error. If it may err in both directions then it is said to have 2-*sided error*. A tester may be *adaptive*, in the sense that its queries may depend on the answers to previous queries, or *non-adaptive*, in the sense that it first makes all the queries, and then makes its final decision using the answers to these queries. All the lower bounds we prove in this paper hold for the most general testers, namely, 2-sided error adaptive testers.

1.2 Background

One of the most important questions in the field of property testing is to prove general testability results, and more ambitiously to classify the languages that are testable with a certain number of queries. While in the case of (dense) graph properties, many general results are known (see [2] and [3]) there are not too many general results for testing languages that can be decided in certain computational models. Our investigation is more related to the connection between certain classical complexity measures of languages and the hardness of testing them, which is measured by their query complexity as defined above. A first result in this direction was obtained by Alon et al. [1] were it was shown that any regular language is easily testable. In fact, it was shown in [1] that any regular language can be tested with an optimal constant number of queries $\Theta(1/\epsilon)$ (the hidden constant depends on the language). It has been long known (see Exercise 2.8.12 in [14]) that any language that can be recognized in space [3] $o(\log\log n)$ is in fact regular. By the result of [1] this means that any such language is easily testable. A natural question is whether it is possible to extend the family of easily testable languages beyond those with space complexity $o(\log\log n)$. It was (implicitly) proved in [1] that there are properties in space $O(\log n)$ that are not easily testable (see Theorem 4

[1] It will sometimes be convenient to refer to properties $\mathcal{P}$ of strings as languages L, as well as the other way around, where the language associated with the property is simply the set of strings that satisfy the property.

[2] We note that some papers use the term *easily testable* to indicate that a language is testable with $poly(1/\epsilon)$ queries.

[3] Throughout this paper we consider only deterministic space complexity. Our model for measuring the space complexity of the algorithm is the standard Turing Machine model, where there is a read only input tape, and a work tape where the machine can write. We only count the space used by the work tape. See [14] for the precise definitions. For concreteness we only consider the alphabet $\{0, 1\}$.

below), and Newman [13] raised the question of closing the exponential gap between the $o(\log\log n)$ space positive result and the $\Omega(\log n)$ space negative result. Another natural question is whether the family of easily testable languages can be extended beyond those of regular languages by considering stronger machines. Newman [13] has considered *non-uniform* extensions of regular languages and showed that any language that can be accepted by read-once branching programs of constant width is easily testable. Fischer and Newman [7] showed that this can not be further extended even to read twice branching programs of constant width. For the case of *uniform* extensions, it has been proved in [1] that there are context-free languages that are not easily testable (see [15] for additional results).

In this paper we study the relation between the space complexity and the query complexity of a language. As a special case of this relation we resolve the open problem of Newman [13] concerning the space complexity of the easily testable languages by showing $O(\log\log n)$ space sets that are not easily testable. We also show that the family of easily testable languages cannot be extended to single-counter automata, which seems like the weakest (natural) generalization of finite state automata.

1.3 Main Results

As we have discussed above there are very few known connections between standard complexity measures and query complexity. Our first and main investigation in this paper is about the relation between the space complexity of a language and the query complexity of testing it. Our main result shows that in some cases the relation between space complexity and query complexity may be at least exponential. As we show in Theorem 2 below, there are languages whose space complexity is $O(\log n)$ and whose query complexity is $\Omega(n)$. Also, as we have previously noted, languages whose space complexity is $o(\log\log n)$ can be tested with $\Theta(1/\epsilon)$ queries. Therefore, the interesting space complexities $s(n)$ that are left to deal with are in the "interval" $[\Omega(\log\log n), O(\log n)]$. For ease of presentation it will be more convenient to assume that $s(n) = f(\log\log n)$ for some integer function $x \leq f(x) \leq 2^x$. As in many cases, we would like to rule out very "strange" complexity functions $s(n)$. We will thus say that $s(n) = f(\log\log n)$ is *space constructible* if the function f is space constructible, that is, if given the *unary* representation of a number x it is possible to generate the *binary* representation of $f(x)$ using space $O(f(x))$. Note that natural functions, such as $s(n) = (\log\log n)^2$ and $s(n) = \sqrt{\log n}$ are space constructible [4]. The following is the main result of this paper.

Theorem 1 (Main Result) *Let $s(n)$ be any (space constructible) space complexity function satisfying $\log\log n \leq s(n) \leq \frac{1}{10}\log n$. Then, there is a language in space $O(s(n))$, whose query complexity is $2^{\Omega(s(n))}$.*

We believe it will be interesting to further study the relation between the space complexity and the query complexity of a language. Specifically, we raise the following conjecture claiming that the lower bound of Theorem 1 is best possible:

Conjecture 1 *Any language in space $s(n)$ can be tested with query complexity $2^{O(s(n))}$.*

[4] We note that the notion of space constructibility (of f) that we use here is the standard one, see e.g. [14]. Observe that when $s(n) = (\log\log n)^2$ we have $s(n) = f(\log\log n)$ where the function f is $f(x) = x^2$ and when $s(n) = \sqrt{\log n}$ we have $s(n) = f(\log\log n)$ where the function f is $f(x) = 2^{x/2}$.

Note that Theorem 1 asserts, as a special case, the existence of languages of logarithmic space complexity and query complexity $n^{\Omega(1)}$. As we have mentioned above, we also prove the following stronger result that may be of independent interest.

Theorem 2 *There is a language in space $O(\log n)$ whose query complexity $\Omega(n)$.*

To the best of our knowledge, the lowest complexity class that was previous known to contain a language whose query complexity is $\Omega(n)$, is **P** (see [9]). If Conjecture 1 is indeed true then Theorems 1 and 2 are essentially best possible.

As an immediate application of Theorem 1 we deduce the following corollary, showing that the class of easily testable languages cannot be extended from the family of regular languages even to the family of languages with space complexity $O(\log \log n)$.

Corollary 1 *For any $k > 0$, there is a language in space $O(\log \log n)$, whose query complexity is $\Omega(\log^k n)$.*

Corollary 1 rules out the possibility of extending the family of easily testable languages from regular languages, to the entire family of languages whose space complexity is $O(\log \log n)$, thus answering the problem raised by Newman in [13] concerning the space complexity of easily testable languages.

We turn to address another result, ruling out another possible extension of regular languages. As we have mentioned before, it has been first shown in [1] (see also [15] for further results) that there are context-free languages that are not easily testable. Hence, a natural question is whether there exists a uniform computational model stronger than finite state machines and weaker than stack machines such that all the languages that are accepted by machines in this model are easily testable. Perhaps the weakest uniform model within the class of context-free languages is that of a *deterministic single-counter automaton* (also known as one-symbol push-down automaton). A deterministic single-counter automaton is a finite state automaton equipped with a counter. The possible counter operations are increment, decrement and do nothing, and the only feedback from the counter is whether it is currently 0 or positive (larger than 0). Such an automaton, running on a string ω reads an input character at a time, and based on its current state and whether the counter is 0, moves to the next state and increments/decrements the counter or leaves it unchanged. Such an automaton accepts a string ω if starting with a counter holding the value 0 it reads all the input characters and ends with the counter holding the value 0. It is quite obvious that such an automaton is equivalent to a deterministic push-down automaton with one symbol stack (and a read-only bottom symbol to indicate empty stack).

Deterministic single-counter automata can recognize a very restricted subset of context free languages. Still, some interesting languages are recognized by such an automaton, e.g. D_1 the first Dyck language, which is the language of balanced parentheses. It was shown in [3] that D_1 is easily testable and that more general parenthesis languages (which cannot be recognized by single counter automata) are not easily testable. It was later shown in [15] that testing the more general parenthesis languages requires $\Omega(n^{1/11})$ queries, but can still be carried out using a sub-linear $O(n^{2/3})$ number of queries. Formal definition and discussion on variants of counter automata can be found in [20].

In this paper we also prove the following theorem showing that the family of easily testable languages cannot be extended even to those accepted by single-counter automata.

Theorem 3 *There is a language that can be accepted by a deterministic single-counter automaton and whose query complexity is* $\Omega(\log\log n)$.

Combining Theorem 3 and Corollary 1 we see that in two natural senses the family of easily testable languages cannot be extended beyond that of the regular languages.

Organization: The rest of the paper is organized as follows. In Section 2 we prove the exponential relation between space complexity and query complexity of Theorem 1. In Section 3 we prove Theorem 2. Section 4 contains the proof of Theorem 3 showing that there are languages accepted by counter machines that are not easily testable. Section 5 contains some concluding remarks and open problems.

2 Space Complexity vs. Query Complexity

In this section we prove that languages in space $s(n)$ may have query complexity exponential in $s(n)$. To this end, we will need a language $\mathcal{L}$ whose space complexity is $O(\log n)$ and whose query complexity is $\Omega(n^\alpha)$ for some positive α. Of course, Theorem 2 supplies such an example, but (though possible) it will be hard to use Theorem 2 as we will need to refer to specific values of n for which it is hard to test $\mathcal{L}$. It will be more convenient to apply a result of Alon et al. [1] who have shown that the query complexity of testing the language vv^Ruu^R is $\Omega(\sqrt{n})$ (the string v^R is the "reverse" of v). Moreover, they showed that the lower bound holds for all inputs whose size is divisible by 6 (see the proof in [3]). As it is easy to see that the language vv^Ruu^R is in space $O(\log n)$, we thus get the following.

Theorem 4 (Implicit in [1]) *There is a language* $\mathcal{L}$*, whose space complexity is* $O(\log n)$*, such that for some* ϵ_0 *and all* n *divisible by 6, the query complexity of an* ϵ_0*-tester of* $\mathcal{L}$ *for inputs of length* n*, is* $\Omega(\sqrt{n})$.

We start with a toy example of the proof of Theorem 1 for the case $s(n) = \log\log n$ and then turn to consider the general case. Consider the following language L: a string $x \in \{0,1,\#\}^n$ is in L if it is composed of $n/\log n$ blocks of size $\log n$ each, separated by the # symbol, such that each block is a word of the language of Theorem 4. It can be shown that the query complexity of testing L is $\Omega(\sqrt{\log n})$. As the language of Theorem 4 can be recognized using logarithmic space it is clear that if the blocks of an input are indeed of length $O(\log n)$, then we can recognize L using space $O(\log\log n)$; we just run the logarithmic space algorithm on each of the blocks, whose length is $O(\log n)$. This "seems" to give an exponential relation between the space complexity and the query complexity of L. Of course, the problem is that the above algorithm is not really an $O(\log\log n)$ space algorithm, as if the blocks are not of the right length then it may be "tricked" into using too much space. We thus have to add to the language some "mechanism" that will allow us to check if the blocks are of the right length. This seems to be impossible as we seem to need to initiate a counter that will hold the value $\log n$, but we need to do so without using more than $O(\log\log n)$ space, and just holding the value n requires $\Theta(\log n)$ bits.

The following language will allow us to "initiate" a counter holding the value $\log n$ while using space $O(\log\log n)$: consider the language $\mathcal{B}$, which is defined over the set of symbols $\{0,1,*\}$ as follows: for every integer $r \geq 1$, the language $\mathcal{B}$ contains the string $s_r = \mathrm{bin}_r(0) * \mathrm{bin}_r(1) * \ldots *$

$\text{bin}_r(2^r-1)*$, where $\text{bin}_r(i)$ is the binary representation of the integer i of length precisely r (that is, with possibly leading 0's). Therefore, for every r there is precisely one string in $\mathcal{B}$ of length $(r+1)2^r$. This language is the standard example for showing that there are languages in space $O(\log\log n)$ that are not regular (see [14] Exercise 2.8.11). To see that $\mathcal{B}$ can indeed be accepted by a machine using space $O(\log\log n)$ note that again the wrong way to do so is to verify that the strings between consecutive $*$ symbols are the binary representations of consecutive integers, because if the string does not belong to $\mathcal{B}$ we may be tricked into using space $\Omega(\log n)$ (for example, if one of the strings is of length more than $poly(\log n)$). In order to "safely" verify that a string belongs to $\mathcal{B}$ we check in the i^{th} step that the last i bits in the blocks form an increasing sequence modulo 2^i and that all blocks are of length i. We also check that in the last step we get an increasing sequence. This way in the i^{th} step we use space $O(\log i)$, and it is easy to see that we never use more than $O(\log\log n)$ space.

Recall that we have previously mentioned that one would like to initiate a counter with value (close to) $\log n$ while using space $O(\log\log n)$. Note that after verifying that a string x of length n belongs to $\mathcal{B}$ (while using space $O(\log\log n)$), we are guaranteed that the number of entries before the first $*$ symbol is close to $\log n$. We will thus want to "incorporate" that language $\mathcal{B}$ into the above language L in order to get a language that is both (relatively) hard to test, and (relatively) easy to recognize, in terms of space complexity. From now on we will deal with general space complexity functions $s(n)$. We remind the reader that we confine ourselves to functions $s(n)$ satisfying $\log\log n \leq s(n) \leq \frac{1}{10}\log n$ that can be written as $s(n) = f(\log\log n)$ for some space constructible function f satisfying $x \leq f(x) \leq 2^x$.

The main idea for the proof of Theorem 1 is to "interleave" the language $\mathcal{B}$ with a language consisting of blocks of length (roughly) $6^{s(n)}$ of strings from the language of Theorem 4 (where $\log\log n \leq s(n) \leq \log n$). This second language will be the obvious generalization of the language L discussed above for the case $s(n) = \log\log n$. For ease of presentation the language we construct to prove Theorem 1 is over the alphabet $\{0,1,\#,*\}$. It can easily be converted into a language over $\{0,1\}$ with the same asymptotic properties by encoding each of the 4 symbols using 2 bits. The details follow.

Let $\mathcal{L}$ be the language of Theorem 4 and let $s(n)$ satisfy $s(n) = f(\log\log n)$ for some space constructible function $n \leq f(n) \leq 2^n$. In what follows, let us set for any integer $r \geq 1$

$$n(r) = 2(r+1)2^r.$$

Given a function f as above, we define a language L^f as the union of families of strings $X_{n(r)}$, consisting strings of length $n(r)$. A string $x \in \{0,1,\#,*\}^{n(r)}$ belongs to $X_{n(r)}$ if it has the following two properties:

1. The odd entries of x form a string from $\mathcal{B}$ (thus the odd entries are over $\{0,1,*\}$).
2. In the even entries of x, substrings between consecutive # symbols [5] form a string from $\mathcal{L}$ whose size is *precisely* k, where $k = 6^{f(\lfloor \log r \rfloor)}$. The only exception is the last block for which the only requirement is that it would be of length at most k (thus these entries of the string are over $\{0,1,\#\}$).

[5] The first # symbol is between the first block and the second block.

Note that the strings from $\mathcal{L}$, which appear in the even entries of strings belonging to $X_{n(r)}$ all have length $6^{f(\lfloor \log r \rfloor)}$. As indicated above, we now define L^f as the union of the sets $X_{n(r)}$

$$L^f = \bigcup_{r=1}^{\infty} X_{n(r)} \ . \tag{1}$$

Let us also define

$$K_f = \{6^{f(\lfloor \log r \rfloor)} : r \in \mathbb{N}\} \ , \tag{2}$$

and observe that the words from $\mathcal{L}$, which appear in the even entries of strings belonging to L^f, all have lengths that belong to the set K_f. Define the language $\mathcal{L}_f$ as the subset of $\mathcal{L}$ consisting of words whose length belongs to the set K_f, that is

$$\mathcal{L}_f = \{x \in \mathcal{L} : |x| \in K_f\}. \tag{3}$$

For future reference, let us recall that Theorem 4 guarantees that $\mathcal{L}$ has query complexity $\Omega(\sqrt{n})$ for all integers n divisible by 6. We thus get the following:

Claim 2.1 *For some $\epsilon_0 > 0$, every ϵ_0-tester of $\mathcal{L}_f$ has query complexity $\Omega(\sqrt{n})$.* ■

We now turn to prove the main claims needed to obtain Theorem 1.

Claim 2.2 *The language L^f has space complexity $O(s(n)) = O(f(\log \log n))$.*

Proof: To show that L^f is in space $O(f(\log \log n)) = O(s(n))$ we consider the following algorithm for deciding if an input x belongs to L^f. We first consider only the odd entries of x and use the $O(\log \log n)$ space algorithm for deciding if these entries form a string from $\mathcal{B}$. If they do not we reject and if they do we move to the second step. Note, that at this step we know that the input's length n is $2(r+1)2^r$ for some $r \leq \log n$. In the second step we initiate a binary counter that stores the number $\lfloor \log r \rfloor \leq \log \log n$. Observe, that the algorithm can obtain r by counting the number of odd entries between consecutive $*$ symbols, and that we need $O(\log \log n)$ bits to hold r. We then initiate a counter that holds the value $k = 6^{f(\lfloor \log r \rfloor)}$, using space $O(f(\lfloor \log r \rfloor))$ by exploiting the fact that f is space constructible [6]. We then verify that the number of even entries between consecutive $\#$ symbols is k, except the last block for which we check that the length is at most k. Finally, we run the logarithmic space algorithm of $\mathcal{L}$ in order to verify that the even entries between consecutive $\#$ symbols form a string from $\mathcal{L}$ (except the last block).

The algorithm clearly accepts a string if and only if it belongs to L^f. Regarding the algorithm's space complexity, recall that we use an $O(\log \log n)$ space algorithm in the first step (this algorithm was sketched at the beginning of this section). Note, that after verifying that the odd entries form a string from the language $\mathcal{B}$, we are guaranteed that $r \leq \log n$. The number of bits needed to store the counter we use in order to hold the number $k = 6^{f(\lfloor \log r \rfloor)}$ is $O(f(\lfloor \log r \rfloor)) = O(f(\log \log n)) = O(s(n))$ as needed. Finally, as each block is guaranteed to be of length $6^{f(\lfloor \log r \rfloor)}$, the logarithmic space algorithm that we run on each of the blocks uses space $O(\log(6^{f(\lfloor \log r \rfloor)})) = O(f(\lfloor \log r \rfloor)) = O(f(\log \log n)) = O(s(n))$, as needed. ■

[6] More precisely, given the binary encoding of $\lfloor \log r \rfloor$ we form an unary representation of $\lfloor \log r \rfloor$. Such a representation requires $O(\log \log n)$ bits. We then use the space constructibility of f to generate a binary representation of $f(\lfloor \log r \rfloor)$ using space $O(f(\lfloor \log r \rfloor))$. Finally, given the binary representation of $f(\lfloor \log r \rfloor)$ it is easy to generate the binary representation of $6^{f(\lfloor \log r \rfloor)}$ using space $O(f(\lfloor \log r \rfloor))$.

Claim 2.3 *The language L^f has query complexity $2^{\Omega(f(\log\log n))} = 2^{\Omega(s(n))}$.*

Proof: By Claim 2.1, for some fixed ϵ_0, every ϵ_0-tester for $\mathcal{L}_f$ has query complexity $\Omega(\sqrt{n})$. We claim that this implies that every $\frac{\epsilon_0}{3}$-tester for L^f has query complexity $2^{\Omega(f(\log\log n))}$. The idea is to take a tester for L^f and use it to devise a tester for $\mathcal{L}_f$, using a simple simulation argument. To this end we will need to take an input x to the tester for $\mathcal{L}_f$ and *implicitly* construct an input x' for the tester of L^f in a way that if $x \in \mathcal{L}_f$ then $x' \in L_f$ and if x is far from $\mathcal{L}_f$ then x' is far from L_f.

Let T be an $\frac{\epsilon_0}{3}$-tester for L^f and consider the following ϵ_0-tester $\mathcal{T}$ for $\mathcal{L}_f$: Given an input x, the tester $\mathcal{T}$ immediately rejects x in case there is no integer r for which $|x| = 6^{f(\lfloor \log r \rfloor)}$ (recall that the strings of $\mathcal{L}_f$ are all taken from K_f as defined in (2)). In case such an integer r exists, set $n = 2(r+1)2^r$. The tester $\mathcal{T}$ now *implicitly* constructs the following string x' of length n. The odd entries of x' will contain the unique string of $\mathcal{B}$ of length $(r+1)2^r$. The even entries of x' will contain repeated copies of x separated by the # symbol (the last block may contain some prefix of x). Note that if $x \in \mathcal{L}_f$ then $x' \in L^f$. Furthermore, observe that if x is ϵ-far from $\mathcal{L}_f$ then x' is $(\frac{\epsilon}{2} - o(1))$-far from L^f, because in the even entries of x', one needs to change an ϵ-fraction of the entries in the substring between consecutive # symbols, in order to get a word from $\mathcal{L}_f$ (the $o(1)$ term is due to the fraction of the string occupied by the # symbols that need not be changed). This means that it is enough for $\mathcal{T}$ to simulate T on x' with error parameter $\frac{\epsilon_0}{3}$ and thus return the correct answer with high probability. Of course, $\mathcal{T}$ cannot construct x' "for free" because to do so $\mathcal{T}$ must query all entries of x. Instead, $\mathcal{T}$ only answers the oracle queries that T makes as follows: given a query of T to entry $2i-1$ of x', the tester $\mathcal{T}$ will supply T with the i^{th} entry of the unique string of $\mathcal{B}$ of length $(r+1)2^r$. Given a query of T to entry $2i$ of x', the tester $\mathcal{T}$ will supply T with the j^{th} entry of x, where $j = i \pmod{|x|+1}$. To this end, $\mathcal{T}$ will have to perform a query to the entries of x.

We thus get that if L^f has an $\frac{\epsilon_0}{3}$-tester making t queries on inputs of length $2(r+1)2^r$, then $\mathcal{L}_f$ has an ϵ_0-tester making t queries on inputs of length $6^{f(\lfloor \log r \rfloor)}$. We know by Claim 2.1 that the query complexity of any ϵ_0-tester of $\mathcal{L}_f$ on inputs of length $6^{f(\lfloor \log r \rfloor)}$ is $\Omega(\sqrt{6^{f(\lfloor \log r \rfloor)}})$. This means that the query complexity of $\mathcal{T}$ on the inputs x' we described must also be $\Omega(\sqrt{6^{f(\lfloor \log r \rfloor)}})$. The lengths of these inputs is $n = 2(r+1)2^r$. This means that $r = \log n - \Theta(\log\log n)$ and therefore the query complexity on these inputs is

$$\Omega(\sqrt{6^{f(\lfloor \log r \rfloor)}}) = 2^{\Omega(f(\log\log n - 2))} = 2^{\Omega(f(\log\log n))},$$

where in the last equality we used the fact that $f(x) \leq 2^x$. ■

Proof of Theorem 1: Take the language L^f and apply Claims 2.2 and 2.3. ■

3 Proof of Theorem 2

The proof of Theorem 2 uses dual-codes of asymptotically good linear codes over $GF(2)$, which are based on Justesen's construction [10]. We begin with some brief background from Coding Theory (see [12] for a comprehensive background). A *linear code* C over $GF(2)$ is just a subset of $\{0,1\}^n$ that forms a linear subspace. The (Hamming) *distance* between two words $x, y \in C$, denoted $d(x,y)$, is the number of indices $i \in [n]$ for which $x_i \neq y_i$. The *distance* of the code, denoted $d(C)$ is the minimum distance over all pairs of distinct words of C, that is $d(C) = \min_{x \neq y \in C} d(x,y)$. The *size* of a code,

denoted $|C|$ is the number of words in C. The *dual-code* of C, denoted $C^{\perp}$ is the linear subspace orthogonal to C, that is $C^{\perp} = \{y : \langle x, y\rangle = 0 \text{ for all } x \in C\}$, where $\langle x, y\rangle = \sum_{i=1}^{n} x_i y_i \pmod 2$ is the dot product of x and y over $GF(2)$. The *generator matrix* of a code C is a matrix G whose rows span the subspace of C. Note, that a code is a family of strings of fixed size n and our interest is in languages containing strings of unbounded size. We will thus have to consider families of codes of increasing size. The following notion will be central in the proof of Theorem 2:

Definition 3.1 (Asymptotically Good Codes) *An infinite family of codes $\mathcal{C} = \{C_{n_1}, C_{n_2}, \ldots\}$, where $C_{n_i} \subseteq \{0,1\}^{n_i}$, is said to be* asymptotically good *if there exist positive reals $0 < d, r < 1$ such that $\liminf_{i\to\infty} \frac{d(C_{n_i})}{n_i} \geq d$ and $\liminf_{i\to\infty} \frac{\log(|C_{n_i}|)}{n_i} \geq r$.*

We note that in the coding literature, the above reals d and r are sometimes referred to as the relative distance and relative rate of a code, respectively. We turn to discuss the main two lemmas needed to prove Theorem 2. The first is the following:

Lemma 3.2 *Suppose $\mathcal{C} = \{C_{n_1}, C_{n_2}, \ldots\}$ is an asymptotically good family of linear codes, and set $L = \bigcup_{i=1}^{\infty} C_{n_i}^{\perp}$. Then, for some ϵ_0, the query complexity of ϵ_0-testing L is $\Omega(n)$.*

Lemma 3.2 is essentially a folklore result. Its (simple) proof relies on the known fact that if C is a linear code with distance t then $C^{\perp}$ is a *t-wise independent* family, that is, if one uniformly samples a string from $C^{\perp}$ then the distribution induced on any t coordinates is the uniform distribution. Such families are sometimes called in the coding literature *orthogonal array of strength t*, see [12]. The fact that the codes in $\mathcal{C}$ satisfy $\frac{\log(|C_{n_i}|)}{n_i} \geq r$ implies that $|C^{\perp}| \leq 2^{(1-r)n_i}$ giving that a random string is with high probability far from belonging to $C_{n_i}^{\perp}$. These two facts allow us to apply Yao's principle to prove that even *adaptive* testers must use at least $\Omega(n)$ queries in order to test L for some fixed ϵ_0. For completeness we sketch below the proof of Lemma 3.2. As pointed to us by Eli Ben-Sasson, Lemma 3.2 can also be proved by applying a general non-trivial result about testers for membership in linear codes (see Theorem 3.3 in [4] for more details).

A well known construction of Justesen [10] gives an asymptotically good family of codes. By exploiting the fact that for appropriate prime powers n, one can perform arithmetic operations over $GF(n)$ in space $O(\log n)$, one can use the main idea of [10] in order to prove the following:

Lemma 3.3 *There is an asymptotically good family of linear codes $\mathcal{C} = \{C_{n_1}, C_{n_2}, \ldots\}$ and an algorithm, with the following property: Given integers n, i and j, the algorithm generates entry i, j of the generator matrix of C_n, while using space $O(\log n)$.*

Apparently this result does not appear in any published paper. However, most details of the construction appear in Madhu Sudan's lecture notes [19]. For the sake of completeness we sketch a self contained proof, which is somewhat simpler than the one based on Justesen's [10] construction as it doesn't use code ensembles. Theorem 2 will follow by a simple application of Lemmas 3.2 and 3.3.

Proof of Lemma 3.2: Consider any integer n_i and let us henceforth refer to it as n. In order to show the lower bound for testing words of length n, we will apply Yao's principle, according to which it is enough to show that there is a distribution of inputs $\mathcal{D}$ of length n, such that any *deterministic*

adaptive ϵ_0-tester, making $o(n)$ queries, will err on the inputs generated by $\mathcal{D}$ with probability larger than 1/3. Consider the distribution $\mathcal{D}$, where with probability $\frac{1}{2}$ we generate a string using a distribution $\mathcal{D}_N$, which consists of strings that are typically far from L, and with probability $\frac{1}{2}$ we generate a string using a distribution $\mathcal{D}_P$, which consists of strings that are typically far from L. The distribution $\mathcal{D}_N$ generates random strings from $\{0,1\}^n$, while $\mathcal{D}_P$ generates random strings from $C_n^\perp$.

Let A be any deterministic $\frac{1}{2}r$-tester making $q < dn$ queries, where d and r are the constants appearing in Definition 3.1. Consider a representation of A as binary decision tree of depth q. Each vertex is labelled with an index $1 \leq i \leq n$, representing the query the algorithm makes. Each leaf of the tree is represented with either "accept" or "reject" (remember that A is deterministic). Note that the probability of reaching any fixed leaf of the tree given that a string is generated by $\mathcal{D}_N$ is *precisely* $(\frac{1}{2})^q$. It is well known (see [12], Chapter 1, Theorem 10) that if $d(C_n) \geq dn$ then $C_n^\perp$ is a dn-wise independent family, namely, if we generate a random string from $C_n^\perp$ then the distribution induced on *any* set of dn coordinates is the uniform one. Therefore, the probability of reaching any fixed leaf of the tree of A given that a string is generated by $\mathcal{D}_P$ is also *precisely* $(\frac{1}{2})^q$. Let r_P be the probability of rejecting a string generated by $\mathcal{D}_P$ and r_N be the probability of rejecting a string generated by $\mathcal{D}_N$. By the above discussion we get that $r_N = r_P$.

As C_n is a linear subspace containing at least 2^{rn} strings, where r is the constant from Definition 3.1, we get that its dual subspace $C_n^\perp$ contains at most $2^{(1-r)n}$ words. By the union bound, this means that the probability that a string generated by $\mathcal{D}_N$ is $\frac{1}{2}r$-close to $C_n^\perp$ is at most

$$2^{-n} \cdot |C_n^\perp| \sum_{i=1}^{\frac{1}{2}rn} \binom{n}{i} \leq 2^{-n} 2^{(1-r)n} 2^{\frac{2}{3}rn} = o(1).$$

Therefore, with probability $1 - o(1)$ a string generated by $\mathcal{D}_N$ is $\frac{r}{2}$-far from L. Let a'_N denote the probability of (incorrectly) accepting an input generated by $\mathcal{D}$ given that it is $\frac{1}{2}r$-far from L. We thus get that $a'_N = 1 - r_N - o(1)$. This, together with the fact that $r_N = r_P$, implies that the probability of A erring on $\mathcal{D}$ is $\frac{1}{2}r_P + \frac{1}{2}a'_N = \frac{1}{2}r_P + \frac{1}{2}(1 - r_N - o(1)) = \frac{1}{2} - o(1) > 1/3$. ■

Proof of Lemma 3.3 (sketch): We use code concatenation (see [12]) in order to reduce the size of the domain of a Reed-Solomon code. We do this repeatedly until the domain/alphabet is small enough to allow us to use logarithmic space while exhaustively looking for an asymptotically good code. We start with an $[n, n/2, n/2]_n$ Reed-Solomon code [7], whose generator matrix can clearly be constructed in space $O(\log n)$. We now concatenate this code with another (appropriate) Reed-Solomon code in order to reduce the domain size to $O(\log n)$ and then concatenate with another Reed-Solomon code in order to reduce the domain size to $O(\log \log n)$. The properties of code concatenation guarantee that the resulting code C_2 has relative distance and relative rate at least 1/8. As the domain size of C_2 is $q = O(\log \log n)$ we can now use exhaustive search to find a code C_3 mapping binary strings of length $\log q = O(\log \log \log n)$ to binary strings of length (say) $4 \log q$ and whose distance is 1/4. Moreover this computation can clearly be carried in space $O(\log n)$. The concatenated code $C_2 \circ C_3$ is therefore over $\{0,1\}$, its distance is 1/32 and its rate is also 1/32. This code satisfies the requirements of Lemma 3.3. ■

[7] An $[n, k, d]_q$ code is a mapping $C : \Sigma^k \to \Sigma^n$, where Σ is a domain of size q with the property that for every $x \neq y$ we have $d(C(x), C(y)) \geq d$. See [12] for more details.

Proof of Theorem 2: Let $\mathcal{L} = \{C_{n_1}, C_{n_2}, \ldots\}$ be the family of codes guaranteed by Lemma 3.3 and as in Lemma 3.2 define $L = \bigcup_i C_{n_i}^{\perp}$. By Lemma 3.2 L has query complexity $\Omega(n_i)$ for all the integers n_i. To show that L can be recognized with space $O(\log n)$ we rely on Lemma 3.3: given an input $x \in \{0,1\}^{n_i}$ we accept if and only if $x \in C_{n_i}^{\perp}$. This can be achieved by accepting if and only if $Mx = 0$, where M is the generator matrix of C_{n_i}. Note, that we can compute each of the entries of Mx sequentially, while reusing space by iteratively multiplying x by the rows of M and verifying that they all equal 0 over $GF(2)$. In order to verify that the inner product of x and the t^{th} row of M is 0, we sequentially generate the entries $M_{t,1}, \ldots, M_{t,n}$ and multiply them by the corresponding entries of x. Note, that at each stage we only have to keep one bit, which is the current sum of $\sum_{k=1}^{j} x_k M_{t,k}$ over $GF(2)$. Apart from that we only have to keep the indices t and j and the space needed to generate the entries on M. The total is space $O(\log n)$. ■

4 Testing Counter Machine Languages May Be Hard

4.1 Proof overview

In this section we define a language $\mathcal{L}$ that can be decided by a deterministic single-counter automaton and prove an $\Omega(\log \log n)$ lower bound on the query complexity of *adaptive 2-sided error* testers for testing membership in $\mathcal{L}$. We start with defining the language $\mathcal{L}$.

Definition 4.1 *$\mathcal{L}$ is the family of strings $s \in \{0,1\}^*$ such that $s = 0^{k_1}1^{k_1}0^{k_2}1^{k_2}\ldots 0^{k_i}1^{k_i}$ for some integer i and some sequence of integers $k_1, \ldots, k_i$. For every integer n we set $\mathcal{L}_n = \mathcal{L} \cap \{0,1\}^n$.*

We proceed with the proof of Theorem 3 using the above language $\mathcal{L}$. First note that one can easily see that $\mathcal{L}$ can be accepted by a deterministic single-counter automaton (as defined in Subsection 1.3). What we are left with is thus to prove the claimed lower bound on testing $\mathcal{L}$. Note that any adaptive tester of a language $L \subseteq \{0,1\}^*$ with query complexity $q(\epsilon, n)$ can be simulated by a non-adaptive tester with query complexity $2^{q(\epsilon,n)}$. Therefore, in order to prove our $\Omega(\log \log n)$ lower bound, we may and will prove an $\Omega(\log n / \log \log n)$ lower bound that holds for *non-adaptive* testers. To this end we apply Yao's minimax principle, which implies that in order to prove a lower bound of $\Omega(\log n / \log \log n)$ for non-adaptive testers it is enough to show that there is a distribution $\mathcal{D}$ over legitimate inputs (e.g., inputs from $\mathcal{L}_n$ and inputs that are $\frac{1}{24}$-far from $\mathcal{L}_n$), such that for any non-adaptive *deterministic* algorithm Alg that makes $o(\log n / \log \log n)$ queries, the probability that Alg errs on inputs generated by $\mathcal{D}$ is at least 1/3.

As in many applications of Yao's principle, the overall strategy for constructing the distribution $\mathcal{D}$ is to define $\mathcal{D}$ as a "mixture" of two distributions $\mathcal{D}_P$, which consists of instances that satisfy $\mathcal{L}$, and $\mathcal{D}_N$, which consists of instances that are far from satisfying $\mathcal{L}$. As in other cases, what we need to show is that a low query-complexity algorithm has a very small probability of distinguishing between instances generated by $\mathcal{D}_P$ and instances generated by $\mathcal{D}_N$. To this end, one usually needs to show that unless the algorithm finds "something" in the input, the two distributions look *exactly* the same.

We start with describing the key "gadget" of the proof. We then give an overview of why this gadget is indeed helpful for obtaining the lower bound. Afterwards we give the formal details. For

any positive integer ℓ let us define the following two pairs of strings [8]:

$$BAD_\ell = \left\{ \begin{array}{l} 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 0^\ell\ 1^\ell, \\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 1^\ell \end{array} \right\}$$

$$GOOD_\ell = \left\{ \begin{array}{l} 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell, \\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell \end{array} \right\}$$

Note, that for any ℓ each of the four strings is of length 12ℓ. We refer to strings selected from these sets as 'phrase strings'. We view the phrase strings as being composed of twelve disjoint intervals of length ℓ, which we refer to as 'phrase segments'. By the definition of the 'phrase strings' each 'phrase segment' is an homogeneous substring (that is, all its symbols are the same).

The reader should note that for any ℓ, the four strings in BAD_ℓ and $GOOD_\ell$ have the following two important properties: (i) The four strings have the same (boolean) value in phrase segments $1, 4, 5, 8, 9$ and 12. (ii) In the other phrase segments, one of the strings in BAD_ℓ has the value 0 and the other has value 1, and the same applies to $GOOD_\ell$. The idea behind the construction of $\mathcal{D}$ and the intuition of the lower bound is that in order to distinguish between a string chosen from BAD_ℓ and a string chosen from $GOOD_\ell$ one must make queries into two distinct phrase segments. The reason is that by the above observation, even if one knows ℓ, if all the queries belong to segment $i \in [12]$, then either the answers are all identical and are known in advance (in case $i \in \{1, 4, 5, 8, 9, 12\}$), or they are identical and have probability $1/2$ to be either 0 or 1, *regardless* of the set from which the string was chosen. Thus, queries into two different phrase segments are needed for distinguishing GOOD from BAD. As we now explain, we add another mechanism that will make it hard for a tester, who does not know ℓ, to make such a pair of queries into two distinct phrase segments (for example 10 and 11).

In the construction of the "hard" distribution $\mathcal{D}$ we select with probability $1/2$ whether the string we choose will be a positive instance (a string in $\mathcal{L}_n$) or a negative instance (a string $\frac{1}{24}$-far from $\mathcal{L}_n$). We select a positive instance by distribution $\mathcal{D}_P$ over $\{0,1\}^n$ that is defined as follows:

1. Uniformly select an integer $s \in [3, \lfloor \log n \rfloor - 3]$ and set $\ell = 2^s$ and $r = \lceil \frac{n}{24\ell} \rceil$.
2. For each $i \in [r]$ set $B_i = 0^{b_i} 1^{b_i} \beta_i$, where b_i is uniformly and independently selected from $[\ell/4, \ldots, \ell/2]$ and β_i is uniformly and independently selected from $Good_\ell$. We refer to B_i as the i^{th} 'block string'. We refer to the substring $0^{b_i} 1^{b_i}$ as the 'buffer string' and β_i as the 'phrase'.
3. Set $\alpha = B_1 \cdots B_r 0^t 1^t$, where $t = (n - \sum_{i=1}^{r} |B_i|)/2$.

We select a negative instance by distribution $\mathcal{D}_N$ that is defined in the same manner as $\mathcal{D}_P$ with the exception that in the second stage we select strings from BAD_ℓ instead of $GOOD_\ell$.

Recall the only way to distinguish whether a string generated by $\mathcal{D}$ is a positive instance or a negative instance is if at least two queries are located in the same phrase string, but in different phrase segments. Obviously, for two queries $q_1 < q_2$ to be in the same phrase string it must be that $q_2 - q_1 < 12\ell$. Moreover if one wants to ensure that with reasonable probability two queries q_1, q_2 are in different phrase segments of the same phrase it must be that $q_2 - q_1 \geq \ell / \log n$. We show this by generalizing the proof of the following specific case.

[8] In the proof ℓ will take values between (roughly) 1 and $\log n$, where n is the length of the string that contains these substrings.

Assume that $q_2 - q_1 < \ell/\log n$ and that a block string B_i starts at $m = q_1 - 3\ell/2 + 1$. Let t be the location of the boundary between the first and the second phrase segment of the phrase string in B_i. By definition $t = m + 2b_i + \ell$, where $2b_i$ is the length of the buffer string of B_i and ℓ is the length of a phrase segment. Observe that in this case the value of t depends only on the choice b_i since m and ℓ are fixed. We also know that t is between q_1 and q_2 if and only if $b_i \in [\ell/4, \ldots, \ell/4 + (q_2 - q_1)/2]$. As b_i is uniformly selected from $[\ell/4, \ldots, \ell/2]$, the probability of this event is $2(q_2 - q_1)/\ell \leq 2/\log n$. One should note that in this case the boundary between q_1 and q_2 is the only one that can fall between q_1 and q_2.

4.2 Proof of Theorem 3

We assume in what follows that $n \geq 2^{16}$ (large enough). We need the following claim showing that $\mathcal{D}_P$ and $\mathcal{D}_N$ are over legitimate inputs.

Claim 4.2 *Every instance drawn from $\mathcal{D}_P$ satisfies $\mathcal{L}_n$. Every instance drawn from $\mathcal{D}_N$ is $1/24$-far from $\mathcal{L}_n$.*

Proof of Claim 4.2 First, note that every string which is generated by $\mathcal{D}_P$, is a concatenation of strings from $\mathcal{L}$ (e.g., strings in $GOOD_\ell$ and of the sort 0^t1^t). Hence, a string drawn from $\mathcal{D}_P$ belongs to $\mathcal{L}_n$ with probability 1.

Let α be a string selected according to $\mathcal{D}_N$. Recall that this means that α contains $\lceil \frac{n}{24\ell} \rceil$ disjoint phrase strings each selected from BAD_ℓ. We claim that any string in $\mathcal{L}_n$ differs from α on at least ℓ entries of each such substring and hence α is $1/24$-*far* from $\mathcal{L}_n$.

Consider any phrase $\beta \in BAD_\ell$ of α and note that it contains the substring $1^\ell 0^\ell 1^{3\ell}$. We set m to be the entry in which this substring begins. Let γ be a string in $\mathcal{L}_n$. Assume first that there is no entry $j \in \{m, \ldots, m + 2\ell - 1\}$ such that γ is 0 at j and 1 at $j - 1$. Consequently the substring of γ whose entries are $\{m, \ldots, m + 2\ell - 1\}$ is of the sort $0^w 1^{2\ell - w}$. Now since the substring of α, whose entries are $\{m, \ldots, m + 2\ell - 1\}$ is $1^\ell 0^\ell$, we get that α differs from γ on at least ℓ entries of the phrase β. Assume that there is an entry $j \in \{m, \ldots, m + 2\ell - 1\}$ such that γ is 0 at j and 1 at $j - 1$. Recall that for every string in $\mathcal{L}_n$ any maximal sequence of 0's must be followed by a maximal sequence of 1's of the same length. As j is the first entry of a maximal sequence of 0's in γ the number of entries in $\{j, \ldots, m + 5\ell - 1\}$ on which γ is 0 is at least the number of such entries on which γ is 1. By definition the number of entries in $\{j, \ldots, m + 5\ell - 1\}$ in which α is 1 is greater by at least 2ℓ from the number of such entries for which α is 0. Therefore, α differs from γ on at least ℓ entries in $\{j, \ldots, m + 5\ell - 1\}$ (e.g., in β). ■

Let Alg be any fixed deterministic algorithm that uses $d = o(\log n/\log\log n)$ queries. We will show that the error Alg has when trying to distinguish between the case that the input is drawn from $\mathcal{D}_P$ and the case it is drawn from the distribution $\mathcal{D}_N$ is at least $1/3$. As the lower bound is proved for the non-adaptive case, we may assume that the queries $Q = \{q_1, \ldots, q_d\}$ are fixed in advance and renumbered such that $q_1 < q_2 \ldots < q_d$. Let $\Delta_i = q_{i+1} - q_i$ be the distances between consecutive queries. Our strategy is to show that conditioned on a certain "good" event $\mathcal{G}$, the error of Alg is actually $1/2$. We will then show that the event $\mathcal{G}$ happens with very high probability.

Let $\mathcal{G}$ be the event that ℓ and $b_1, \ldots, b_r$ (that are selected in Steps 1 and 2 in the definition of D_N and D_P) are such that for every $1 \leq i < j \leq d$, if q_i and q_j are in the same phrase of α then they

are also in the same phrase segment. As a preliminary step towards analyzing event $\mathcal{G}$, let us define event $\mathcal{A}$ as the event that ℓ chosen according to $\mathcal{D}$ is such that each pair of neighboring queries is either extremely close (relative to ℓ), or extremely far (relative to ℓ). Formally,

$$\mathcal{A} \triangleq \forall i \ (\ (\Delta_i \leq \frac{\ell}{\log n}) \vee (\Delta_i \geq 24\ell) \ ). \tag{4}$$

Claim 4.3 $\text{Prob}[\mathcal{A}] = 1 - o(1)$.

Proof: Recall that $\ell = 2^s$ and $\Delta_i = q_{i+1} - q_i$ for every $i \in [d]$. Set $p_i = \text{Prob}\left[\frac{\ell}{\log n} \leq \Delta_i \leq 24\ell\right]$ then

$$p_i = \text{Prob}_s\left[\frac{2^s}{\log n} \leq \Delta_i \leq 24 \cdot 2^s\right].$$

Taking log and rearranging terms we get that equivalently

$$p_i = \text{Prob}_s\left[\log \Delta_i - \log 24 \leq s \leq \log \Delta_i + \log\log n\right].$$

Since s is distributed uniformly in $[3, \lfloor \log n \rfloor - 3]$ we have

$$p_i \leq \frac{\log\log n + \log 24}{\lfloor \log n \rfloor - 6}$$

By the union bound we have for $d = o(\log n / \log\log n)$:

$$\text{Prob}[\mathcal{A}] \geq 1 - \sum_{i=1}^{d} p_i = 1 - o(1).$$

■

Claim 4.4 $\text{Prob}[\mathcal{G}] = 1 - o(1)$.

Proof of Claim 4.4 By Claim 4.3 we know that $\text{Prob}[\mathcal{A}] = 1 - o(1)$. Therefore, it is enough to show that $\text{Prob}\left[\mathcal{G} \mid \mathcal{A}\right] = 1 - o(1)$. Note, that the assumption that event $\mathcal{A}$ holds only carries information about s and no information about the integers $b_1, \ldots, b_r$ that are chosen in the second step. Assume then that event $\mathcal{A}$ holds and let $Q = \{q_1, \ldots, q_d\}$ be the queries of the algorithm. Recall that we numbered the queries such that $q_1 < q_2 < \ldots < q_d$. Consider the following process for partitioning the set Q: put q_1 in a set Q_1, and keep adding queries q_i into Q_1 as long as Δ_i (i.e. the distance between q_i and q_{i-1}) is at most $\ell / \log n$. If the distance is larger then "open" a new set Q_2, put q_i in Q_2 and continue as above. Suppose the resulting partition is $Q_1, \ldots, Q_p$ such that $Q_1 = \{q_{l_1}, \ldots, q_{r_1}\}$, $Q_2 = \{q_{l_2}, \ldots, q_{r_2}\}$, $\ldots$, $Q_p = \{q_{l_p}, \ldots, q_{r_p}\}$ where for every $i \in [p]$ we have that q_{l_i} is the smallest query in Q_i and q_{r_i} is the largest query in Q_i. As we assume that event $\mathcal{A}$ holds and by the definition of the partition we know that the distance between queries that belong to different sets is at least 24ℓ. As the length of each phrase string is 12ℓ such pairs do not belong to the same phrase and we should not worry about them.

Consider now pairs belonging to the same set Q_i. We will show that with high probability the first and last query in each of the sets Q_i will belong to the same phrase segment. This will clearly

imply that event $\mathcal{G}$ holds. Consider for simplicity the set Q_1. Think of generating the strings in $\mathcal{D}_N$ and $\mathcal{D}_P$ as an iterative process, where in each iteration we generate an integer $b_i \in [\ell/4, \ldots, \ell/2]$ and then add to the end of the string $0^{b_i}1^{b_i}$ concatenated with a string β_i from either BAD_ℓ or $GOOD_\ell$. This means that in each iteration the length of the string grows by at least $25\ell/2$ and at most 13ℓ and the $\ell/2$ locations after it are in a buffer string. Assume that the length p of the current string is in $[q_{l_1} - 13\ell, \ldots, q_{r_1} - \ell]$ and that B_i is the block string that starts at p. Note that now only the choice of b_i determines whether a boundary between phrase segments that may fall between q_{l_1} and q_{r_1}. Since the length of the buffer string of B_i is in the range $[\ell/2, \ldots, \ell]$ there is at most one boundary between phrase segments that may fall between q_{l_1} and q_{r_1}. If there is such a boundary let $p + 2b_i + a\ell$ be the location of this boundary. As p,a and ℓ are fixed this location depends only on the value of b_i. Since the value of b_i is selected uniformly, the probability that $p + 2b_i + a \cdot \ell$ is between q_{l_1} and q_{r_1} is at most $2(q_{r_1} - q_{l_1})/\ell$. The above analysis is true for the first and last element of every set Q_i. Hence, again by the union bound,

$$\text{Prob}_{\mathcal{D}}\left[\mathcal{G} \mid \mathcal{A}\right] \geq 1 - \sum_{i=1}^{p} \frac{2(q_{r_i} - q_{l_i})}{\ell}. \tag{5}$$

As we assume the event $\mathcal{A}$ the distance between consecutive queries is at most $\ell/\log n$. This means that for any set Q_i the distance between the first and the last queries is

$$q_{r_i} - q_{l_i} \leq \frac{\ell |Q_i|}{\log n}. \tag{6}$$

Hence, combining (5) and (6) we have that:

$$\begin{aligned} \text{Prob}_{\mathcal{D}}\left[\mathcal{G} \mid \mathcal{A}\right] &\geq 1 - \frac{2}{\log n} \sum_{i=1}^{p} |Q_i| \\ &= 1 - \frac{2d}{\log n}\ . \end{aligned}$$

As $d = o(\log n/\log\log n)$ we conclude that $\text{Prob}_{\mathcal{D}}\left[\mathcal{G} \mid \mathcal{A}\right] = 1 - o(1)$. ■

Proof of Theorem 3: As we have mentioned before, it is enough to prove a lower bound for deterministic non-adaptive testers via Yao's minimax principle. Let Alg be any deterministic non-adaptive $\frac{1}{24}$-tester for $\mathcal{L}_n$ whose query complexity is $o(\log n/\log\log n)$. Consider the distribution $\mathcal{D}$ which generates an element from $\mathcal{D}_P$ with probability $1/2$ and an element from $\mathcal{D}_N$ with probability $1/2$. By Claim 4.2 we know that this distribution generates inputs that are either in $\mathcal{L}_n$ or are $\frac{1}{24}$-far from belonging to this language.

According to the definition of $\mathcal{D}$ the choice whether the string generated is a positive instance or a negative instance does not depend on the selection of the parameters s,$b_1, \ldots, b_r$ and vice versa. Consequently, we can and actually do view the process of generating a string by $\mathcal{D}$ as being done by two consecutive stages: In the first stage the parameters s,$b_1, \ldots, b_r$ are selected; In the second stage it is first determined whether the phrases selected are from BAD_ℓ of from $GOOD_\ell$ (e.g., whether the instance generated is a positive instance of a negative instance) afterwards the individual phrases are selected and the generated string is constructed.

Let $Q = \{q_1, \ldots, q_d\}$, where $q_1 < q_2 < \ldots < q_d$, be the set of queries used by Alg. As Alg is deterministic and non-adaptive we may assume that the set Q is fixed in advance. Recall that $\mathcal{G}$ is the event that given Q the integers ℓ and $b_1, \ldots, b_r$ are such that for every $1 \leq i < j \leq d$, if q_i and q_j are in the same phrase of α then they are also in the same phrase segment. To conclude the claim we show that if the values of ℓ and $b_1, \ldots, b_r$ determined in the first stage of $\mathcal{D}$ satisfy $\mathcal{G}$ then the distribution of the answers to Q is exactly the same if in the second stage of $\mathcal{D}$ either $GOOD_\ell$ is selected or BAD_ℓ is selected. This is sufficient, since in this case Alg errs with probability exactly $1/2$ (the probability of selecting BAD_ℓ). As by Claim 4.4 the values of ℓ and $b_1, \ldots, b_r$ determined in the first stage of $\mathcal{D}$ satisfy $\mathcal{G}$ with probability $1 - o(1)$ we get that overall Alg errs with probability $1/2 - o(1)$.

From here on we assume that the values of s, which determines ℓ, and the b_i's selected by $\mathcal{D}$ are all fixed and satisfy $\mathcal{G}$. This implies that the length of every string used by $\mathcal{D}$ is now fixed and only the choice of the specific phrase strings has not been determined. Thus, the answers to every query that is **not** in one of the phrase segments $\{2, 3, 6, 7, 10, 11\}$ of some phrase string is fixed. Therefore, we assume that each query in Q is in one of the phrase segments $\{2, 3, 6, 7, 10, 11\}$ of some phrase string.

Let $\mathcal{Q}$ be the family of all maximal subsets Q' of Q, such that all queries in Q' are in the same phrase string. We construct a subset Q'' of Q by selecting one arbitrary query from each set in $\mathcal{Q}$. Let Q' be a set in $\mathcal{Q}$. As $\mathcal{G}$ is satisfied all the queries in Q' are in the same phrase segment. Hence, the answer to all the queries in Q' is the same for any choice of phrase string. This implies that if the distribution of the answers to Q'' is exactly the same when $GOOD_\ell$ is selected and when BAD_ℓ is selected, then the same is true for Q. As each query in Q'' is in a different phrase string and the phrase strings are selected independently, the answer to each of the queries in Q'' is independent of all the other answers. Since each query in Q'' is in one of the phrase segments $\{2, 3, 6, 7, 10, 11\}$, the answers to each query in Q'' is 0 or 1 with probability $1/2$. Thus, the distribution of answers to Q'' is the uniform distribution when $GOOD_\ell$ is selected and when BAD_ℓ is selected. ■

5 Concluding Remarks and Open Problems

Our main result in this paper gives a relation between the space complexity and the query complexity of a language, showing that the later may be exponential in the former. We also raise the conjecture that this relation is tight, namely that the query complexity of a language is at most exponential in its space complexity. The results of this paper further show that the family of easily testable languages cannot be extended beyond that of the regular languages in terms of two natural senses; the space complexity of the accepting machine or a minimal natural computational model in which it can be recognized.

An intriguing related question is to understand the testability of languages with sublinear number of queries. In particular, an intriguing open problem is whether all the context free languages can be tested with a sublinear number of queries. Currently, the lower bounds for testing context-free languages are of type $\Omega(n^\alpha)$ for some $0 < \alpha < 1$, see [15]. It seems that as an intermediate step towards understanding the testability of context-free languages, it will be interesting to investigate whether all the languages acceptable by single-counter automata can be tested with $o(n)$ queries. We note that the language we constructed in order to prove Theorem 3 can be tested with $poly(\log n, \epsilon)$ queries. As the proof is rather involved we refer the interested reader to [11].

Acknowledgments: The authors would like to thank Noga Alon, Madhu Sudan and Eli Ben-Sasson for helpful discussions, and Oded Goldreich for his useful comments regarding the presentation of our results.

References

[1] N. Alon, M. Krivelevich, I. Newman and M. Szegedy, Regular languages are testable with a constant number of queries, Proc. 40^{th} FOCS, New York, NY, IEEE (1999), 645–655. Also: SIAM J. on Computing 30 (2001), 1842-1862.

[2] N. Alon and A. Shapira, A characterization of the (natural) graph properties testable with one-sided error, Proc. of FOCS 2005, 429-438.

[3] N. Alon, E, Fischer, I. Newman and A. Shapira, A combinatorial characterization of the testable graph properties: it's all about regularity, Proc. of STOC 2006, 251-260.

[4] E. Ben-Sasson, P. Harsha and S. Raskhodnikova, Some 3-CNF properties are hard to test, Proc. of STOC 2003, 345-354.

[5] M. Blum, M. Luby and R. Rubinfeld, Self-testing/correcting with applications to numerical problems, JCSS 47 (1993), 549-595.

[6] E. Fischer, The art of uninformed decisions: A primer to property testing, The Computational Complexity Column of The Bulletin of the European Association for Theoretical Computer Science 75 (2001), 97-126.

[7] E. Fischer, I. Newman and J. Sgall, Functions that have read-twice constant width branching programs are not necessarily testable, Random Structures and Algorithms, 24 (2004), 175-193.

[8] O. Goldreich, Combinatorial property testing - a survey, In: Randomization Methods in Algorithm Design (P. Pardalos, S. Rajasekaran and J. Rolim eds.), AMS-DIMACS (1998), 45-60.

[9] O. Goldreich, S. Goldwasser and D. Ron, Property testing and its connection to learning and approximation, Proc. of 37^{th} Annual IEEE FOCS, (1996), 339–348. Also: JACM 45(4): 653-750 (1998).

[10] J. Justesen, A class of constructive asymptotically good algebraic codes, IEEE Transcations on Information, 18:652-656, 1972.

[11] O. Lachish and I. Newman, Languages that are Recognized by Simple Counter Automata are not necessarily Testable, ECCC report TR05-152.

[12] F. MacWilliams and N. Sloane, **The Theory of Error-Correcting Codes**, North-Holland, Amsterdam, 1997.

[13] I. Newman, Testing of functions that have small width branching programs, Proc. of 41^{th} FOCS (2000), 251-258.

[14] C. Papadimitriou, **Computational Complexity**, Addison Wesley, 1994.

[15] M. Parnas, D. Ron and R. Rubinfeld, Testing membership in parenthesis languages, Random Structures and Algorithms, 22 (2002), 98-138.

[16] D. Ron, Property testing, in: P. M. Pardalos, S. Rajasekaran, J. Reif and J. D. P. Rolim, editors, *Handbook of Randomized Computing*, Vol. II, Kluwer Academic Publishers, 2001, 597–649.

[17] R. Rubinfeld, Sublinear time algorithms, Proc. of ICM 2006, to appear.

[18] R. Rubinfeld and M. Sudan, Robust characterization of polynomials with applications to program testing, *SIAM J. on Computing* 25 (1996), 252–271.

[19] M. Sudan, Lecture Notes on Algorithmic Introduction to Coding Theory, available at http://theory.lcs.mit.edu/∼madhu/FT01/scribe/lect6.ps.

[20] L.G. Valiant, M. Paterson, Deterministic one-counter automata, Journal of Computer and System Sciences, 10 (1975), 340–350.