RAIRO-Theor. Inf. Appl. 47 (2013) 241–259
DOI: 10.1051/ita/2013038

Available online at:
www.rairo-ita.org

FACTORING AND TESTING PRIMES IN SMALL SPACE *

Viliam Geffert[1] and Dana Pardubská[2]

Abstract. We discuss how much space is sufficient to decide whether a unary given number n is a prime. We show that $O(\log\log n)$ space is sufficient for a deterministic Turing machine, if it is equipped with an additional pebble movable along the input tape, and also for an alternating machine, if the space restriction applies only to its accepting computation subtrees. In other words, the language un-Primes $= \{1^n : n \text{ is a prime}\}$ is in pebble–DSPACE($\log\log n$) and also in accept–ASPACE($\log\log n$). Moreover, if the given n is composite, such machines are able to find a divisor of n. Since $O(\log\log n)$ space is too small to write down a divisor, which might require $\Omega(\log n)$ bits, the witness divisor is indicated by the input head position at the moment when the machine halts.

Mathematics Subject Classification. 11A51, 68Q15, 68Q17.

1. Introduction

Few mathematical ideas are as simple as the concept of a prime number: a prime number can evenly be divided by no number other than 1 and itself. The number 1 is not normally considered to be prime.

Keywords and phrases. Prime numbers, factoring, sublogarithmic space, computational complexity.

* *The first author was supported under contract VEGA 1/0479/12 "Combinatorial Structures and Complexity of Algorithms", the second author partially supported under contract VEGA 1/0671/11 "Computations with Supplementary Information". A preliminary version of this work was presented at* SOFSEM 2009 *[Lect. Notes Comput. Sci., vol. 5404. Springer-Verlag (2009) 291–302].*

[1] Department of Computer Science, P. J. Šafárik University, Jesenná 5, 04001 Košice, Slovakia. viliam.geffert@upjs.sk

[2] Department of Computer Science, Comenius University, Mlynská dolina, 84248 Bratislava, Slovakia. pardubska@dcs.fmph.uniba.sk

Article published by EDP Sciences

The first proof showing the existence of infinite number of primes is due to Euclid. He also taught us that the prime numbers are "bricks" for constructing all other integers. Stated in other words, the Fundamental Theorem of Arithmetic says that each integer $n \geq 1$ can uniquely be factorized into $n = \prod_{m \geq 0} p_m^{\alpha_m}$, where $\alpha_m \geq 0$ is an integer and p_m is the mth prime[3]. However, factoring (together with discrete logarithms) are computationally very hard. The difficulty of these problems has been utilized to design cryptographic systems for secure data transmission over insecure networks, such as internet.

In 2004 [1], it was shown that the binary version of Primes belongs to P. Nevertheless, this algorithm does not exhibit any divisors, if the given n is composite, and hence it does not break security of cryptographic systems. At present, the most promising approach for a fast factoring seems to be the polynomial time quantum algorithm from 1994 [21].

Our paper is devoted to the problem of *how much space* is sufficient to decide whether a *unary* given number n is prime or composite. We show that $O(\log \log n)$ space[4] is sufficient for a deterministic Turing machine, if it is equipped with a single additional pebble movable along the input tape, and also for an alternating Turing machine (no pebble), if the space restriction applies only to its accepting computation subtrees. In what follows, let un-Primes $= \{1^n : n \text{ is a prime}\}$; the corresponding binary language is referred to as Primes. Then

$$\begin{aligned} &\text{un-Primes} \in \text{pebble–DSPACE}(\log \log n), \\ &\text{un-Primes} \in \text{accept–ASPACE}(\log \log n). \end{aligned} \tag{1.1}$$

It should be pointed out that, by an easy combination of known results [1,6], we get that Primes $\in$ P $=$ ASPACE$(\log n)$. It is also well-known that a binary language L is in ASPACE$(s(n))$ if and only if its unary version un-L is in ASPACE$(s(\log n))$. However, this traditional translation works only if $s(\log n) \in \Omega(\log n)$. This follows from the fact that, for a unary given input 1^n, we need $\log n$ bits to store its binary representation. Hence, although Primes $\in$ ASPACE$(\log n)$, accepting its unary version in alternating space $O(\log \log n)$ is *not* a consequence of the previous result.

Actually, by a more sophisticated argument, $\log n$ bits are not necessary for deterministic demon machines (by definition, starting with $\lfloor \log \log n \rfloor$ worktape cells marked off automatically): by the New Translation Lemma [2,3] (see Lem. 5.2 below), a binary language L is in DSPACE$(\log n)$ if and only if its unary version un-L is in demon–DSPACE$(\log \log n)$. Though not published in the literature, the argument for the "$\Rightarrow$" part can be extended to alternating machines. (We do not give a proof here, since this is not required for our results). This gives un-Primes $\in$ demon–ASPACE$(\log \log n)$. However, such result is weaker than (1.1): a demon machine based on results published in [1] does not use its own computational power to keep the space bound, nor does it give any factoring into divisors.

[3]For technical reasons, $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, ... That is, for $m \geq 1$, p_m denotes actually the mth *odd* prime.

[4]Throughout the paper, $\log x$ denotes the *binary* logarithm of x.

Our approach is different: we simply verify divisibility of the input n by values $X = 2, 3, 4, \ldots, n-1$. (Note that a straightforward binary representation of n or X cannot be used, since it requires at least $\log n$ bits on the worktape). The key to our result is a space-efficient arithmetic with integers using the so-called Chinese Residual Representation, already utilized in several other applications [3,9–12,19]. (The reader is referred to [2] for a good survey on such results).

As an additional bonus, our machines can exhibit all the divisors. Since the space of size $O(\log \log n)$ is too small to write down a divisor, which might require $\Omega(\log n)$ bits, a pebble machine can reveal the smallest divisor $X > 1$ by leaving the pebble at the input position X at the moment when it halts. (Alternatively, the machine can be equipped with an additional write-only output tape, to print a binary sequence $d_1 \ldots d_n$, with $d_X \in \{0, 1\}$ indicating whether X divides n).

Similarly, the alternating machine can also be modified to accept composites and detect divisors. The computation tree of this machine is such that an $X \in \{2, \ldots, n-1\}$ divides n if and only if at least one path, in at least one accepting computation subtree, halts with the input head at the position X. All accepting computation subtrees stay within the space bound $O(\log \log n)$. Thus,

$$\text{un-Composites} \in \text{pebble–DSPACE}(\log \log n),$$
$$\text{un-Composites} \in \text{accept–ASPACE}(\log \log n),$$

where un-Composites denotes a complement language for un-Primes. Note that the second result is not a trivial consequence of (1.1), since it is still an open problem whether the alternating space below $\log n$ is closed under complement. (For completeness, deterministic pebble space is closed under complement even below $\log n$ [8]).

2. Basic definitions

Our machines are equipped with a finite-state control, a two-way read-only input tape, and a separate two-way read-write worktape. We assume the reader is familiar with an *alternating Turing machine* [6, 22].

Several different modes of space complexity have been defined in the literature: a deterministic, nondeterministic, or alternating Turing machine works in *(a) strong space* $s(n)$, if every computation on each input of length n uses no more than $s(n)$ cells on the worktape, *(b) accept space* $s(n)$, if every accepting computation subtree on each input of length n uses no more than $s(n)$ cells, *(c) weak space* $s(n)$, if, for each accepted input of length n, there exists at least one accepting computation subtree not using more than $s(n)$ cells, *(d) demon space* $s(n)$, if the machine starts with a worktape consisting of $\lfloor s(n) \rfloor$ cells delimited by endmarkers, *i.e.*, the worktape is not initially blank.

Clearly, these modes are listed in increasing computational power. Notation for the corresponding complexity classes should be obvious. As an example, accept–ASPACE($s(n)$) is the class of languages recognizable by alternating machines

working in $O(s(n))$ accept space, while demon–DSPACE($s(n)$) the class recognizable by deterministic $s(n)$ space bounded demon machines.

A function $s(n)$ is *fully space constructible*, if there exists a deterministic Turing machine that prints the binary representation of $s(n)$ on an extra one-way output tape, not using more than $s(n)$ space on the worktape, for each input of length n. Similarly, a function $f(n)$ is *computable by an alternating machine in accept space* $s(n)$, if there exists an alternating machine such that each input of length n is accepted by at least one computation subtree and, moreover, each path of each accepting computation subtree prints the same value $f(n)$ on the output, not using more than $s(n)$ space. (Some paths may also accept after printing values different from $f(n)$ or using space above $s(n)$, however, such paths can only be a part of a rejecting computation subtree).

For fully space constructible functions, the space complexity classes for all above modes coincide. Since $\log n$ is fully space constructible, we simply write ASPACE($\log n$) without any mode prefix.

The mode difference becomes important for $\log\log n$, since no unbounded monotone function below $\log n$ is fully space constructible [14]. Moreover, the argument of [14] can easily be extended to alternating machines. Nevertheless, as shown in Lemma 4.5, a very tight approximation of $\log\log n$ is computable by an alternating machine in accept space $O(\log\log n)$. Therefore, except for strong–ASPACE($\log\log n$), all remaining classes coincide for alternation in $\log\log n$ space.

We shall also use a modification of the standard model, namely, a *machine with a single additional "pebble"* which can be placed on, detected, and removed from the input tape cell currently scanned by the input head; the corresponding complexity classes are prefixed with "pebble". Since $\log\log n$ is fully space constructible with a help of a pebble [7,8,22], the mode difference need not be distinguished for pebble–DSPACE($\log\log n$).

3. Modular arithmetic

Here we explain how some basic operations on numbers can be performed efficiently in small space. We first introduce two representations of numbers and then, simplifying several details, we present algorithms for some basic arithmetic in these representations.

3.1. Representation of numbers

We first need to introduce some notation. Throughout the paper, $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ denote, respectively, the sets of natural, integer, and real numbers. The set of all real numbers φ satisfying $\alpha \leq \varphi \leq \beta$ is denoted by $\langle\alpha,\beta\rangle$ and of those satisfying $\alpha < \varphi < \beta$ by (α,β). The meaning of $\langle\alpha,\beta)$ should be obvious. Let p_i denote the ith odd prime. The product of the first m odd primes is denoted by M_m,

$$M_m = p_1 \cdot \ldots \cdot p_m. \tag{3.1}$$

Our machines are based on two representations of natural numbers. The first one is referred to as the (Chinese) *residual representation*: each $X \in \{0, \ldots, M_m - 1\}$ can uniquely be represented as $X = (x_1, \ldots, x_m)$, where $x_i = X \bmod p_i$, for $i = 1, \ldots, m$. Clearly, the binary coded numbers x_i and p_i are of length $O(\log p_m)$.

The second one is a *scalar representation*: $X = \alpha_X \cdot M_m$, where $\alpha_X \in \langle 0, 1)$. We use this representation to compare numbers: for each $X = \alpha_X \cdot M_m$ and $Y = \alpha_Y \cdot M_m$, $X \leq Y$ if and only if $\alpha_X \leq \alpha_Y$. The disadvantage of this representation is that α_X is a real number with a long fractional part.

Given a number in the residual representation, *e.g.*, some $X = (x_1, \ldots, x_m)$, the corresponding scalar value α_X can be obtained analytically, as given by (3.2). To keep the notation a little bit simpler, an operation/operator/function ${}_{[p]}\diamond$ denotes the corresponding operator $\diamond$ taken modulo p. As an example,

$$\begin{aligned} x \,{}_{[p]}{+}\, y &\stackrel{\text{df.}}{=} (x+y) \bmod p, \\ {}_{[p]}\prod_{i=1}^{m} x_i &\stackrel{\text{df.}}{=} \left(\prod_{i=1}^{m} x_i\right) \bmod p, \\ x^{[p]^{-1}} &\stackrel{\text{df.}}{=} \min\{y \in \mathbb{N} : (x \cdot y) \bmod p = 1\}. \end{aligned}$$

It is well-known (see, *e.g.*, Cor. 1 in Chapter I/3 in [18]) that the value $x^{[p]^{-1}}$ does exist and, moreover, it is unique, provided that p is a prime and $x \bmod p \neq 0$. (Otherwise, this value might be undefined).

We naturally enlarge the domain for residual arithmetic to real numbers as follows. For each $\alpha \in \mathbb{R}$ and $p \in \mathbb{N}$, $p \neq 0$, let

$${}_{[p]}\alpha \stackrel{\text{df.}}{=} \alpha \bmod p \stackrel{\text{df.}}{=} \beta, \quad \text{where } 0 \leq \beta < p \text{ and } \beta = \alpha + k \cdot p, \text{ for some } k \in \mathbb{Z}.$$

In particular, ${}_{[1]}\alpha$ is the fractional part of α, for each $\alpha \geq 0$.

Now we are ready to give the definition of α_X, for $X = (x_1, \ldots, x_m)$:

$$\alpha_X = {}_{[1]}\sum_{i=1}^{m} \left[\left({}_{[p_i]}\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right)^{[p_i]^{-1}} {}_{[p_i]}\times\, x_i \right] / p_i \tag{3.2}$$

In the next two lemmas, we will show the correctness of this definition, *i.e.*, that $X = \alpha_X \cdot M_m$.

Lemma 3.1. *For each $X \in \{0, \ldots, M_m - 1\}$, $\alpha_X \cdot M_m$ is an integer value in $\{0, \ldots, M_m - 1\}$, where α_X and M_m are defined as above.*

Proof. Obviously, for each $\varphi_1, \ldots, \varphi_m \in \mathbb{R}$, we have ${}_{[1]}\sum_{i=1}^{m} \varphi_i = \sum_{i=1}^{m} \varphi_i - L$, for a suitable integer L. In what follows, L is indexed by X, to point out its connection to the number X. Using (3.2), this gives:

$$\begin{aligned} M_m \times \alpha_X &= M_m \times \left(\sum_{i=1}^{m} \left[\left({}_{[p_i]}\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right)^{[p_i]^{-1}} {}_{[p_i]}\times\, x_i \right] / p_i - L_X \right) \\ &= \sum_{i=1}^{m} M_m \times \left[\left({}_{[p_i]}\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right)^{[p_i]^{-1}} {}_{[p_i]}\times\, x_i \right] / p_i \;-\; M_m \times L_X \\ &= \sum_{i=1}^{m} \left(\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right) \times \left[\left({}_{[p_i]}\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right)^{[p_i]^{-1}} {}_{[p_i]}\times\, x_i \right] \;-\; M_m \times L_X. \end{aligned}$$

It is easy to see that all operations performed in the last formula above produce integer values. Therefore, $M_m \times \alpha_X \in \mathbb{Z}$. Second, by (3.2), α_X is obtained as a sum of reals taken modulo 1, which gives that $0 \leq \alpha_X < 1$, and hence the result follows. □

Lemma 3.2. *For each $X \in \{0, \ldots, M_m - 1\}$ and each $k \in \{1, \ldots, m\}$, we have $(\alpha_X \cdot M_m) \bmod p_k = x_k$.*

Proof. Taken modulo p_k, the formula derived in the proof of Lemma 3.1 gives:

$$
\begin{aligned}
{}_{[p_k]}(M_m \times \alpha_X) &= {}_{[p_k]}\sum_{i=1}^{m} \Bigg(\underbrace{{}_{[p_k]}\prod_{j\neq i}^{m=1} p_j}_{\text{zero, if } i \neq k} \Bigg) {}_{[p_k]}\times \left[\left({}_{[p_i]}\prod_{j\neq i}^{m=1} p_j \right)^{[p_i]^{-1}} {}_{[p_i]}\times x_i \right] \\
&\quad {}_{[p_k]}- \underbrace{M_m}_{\text{multiple of } p_k} {}_{[p_k]}\times L_X \\
&= \left({}_{[p_k]}\prod_{j\neq k}^{m=1} p_j \right) {}_{[p_k]}\times \left[\left({}_{[p_k]}\prod_{j\neq k}^{m=1} p_j \right)^{[p_k]^{-1}} {}_{[p_k]}\times x_k \right] \\
&= \left[\left({}_{[p_k]}\prod_{j\neq k}^{m=1} p_j \right) {}_{[p_k]}\times \left({}_{[p_k]}\prod_{j\neq k}^{m=1} p_j \right)^{[p_k]^{-1}} \right] {}_{[p_k]}\times x_k \\
&= x_k,
\end{aligned}
$$

which completes the argument. □

By combining Lemmas 3.1 and 3.2, we have $X = \alpha_X \cdot M_m$, since two *different* numbers in $\{0, \ldots, M_m - 1\}$ cannot agree in all residues $x_1, \ldots, x_m$.

3.2. Testing divisibility

Using the above number representations, we now give a summary of algorithms for modular arithmetic. Some of them have already been used for other purposes [3, 9–12, 19]. In order to present a complete self-contained construction, we recall their slightly modified versions as Algorithms 1 and 2 (introduced by Lems. 3.3 and 3.4). Based on these we then present our method for testing divisibility, as Algorithm 3, described in Lemma 3.5. Finally, we conclude this section by estimating the total space resources required by this algorithm.

To underline space requirements, we use uppercase literals to denote "large" numbers (not stored on a worktape – usually, due to a space limit); lowercase literals for "small" numbers (stored in space $O(\log \log n)$); and Greek alphabet for "real" numbers (actually, rational values in $\langle 0, 1)$, stored in binary with fractional parts truncated to $O(\log \log n)$ bits). The only exception is n, the length of the input, which is actually a "large" value, for historical reasons.

Lemma 3.3. *Let $X = (x_1, \ldots, x_m)$ be a number in its residual representation. If the value x_i can effectively be computed in $O(\log p_m)$ deterministic space, for each given $i \in \{1, \ldots, m\}$, then the question of whether $X \leq M_m/2$ can also be decided in the same space.*

Proof. Since $X = \alpha_X \cdot M_m$, where $\alpha_X \in \langle 0, 1)$, it is sufficient to decide whether $\alpha_X \leq 1/2$. However, although the scalar α_X can be obtained from the residues $x_1, \ldots, x_m$ by the use of (3.2), the real numbers used in the formula (3.2) cannot be stored in $O(\log p_m)$ space.

The way out is to truncate fractional parts in the binary representation of all reals in $\langle 0, 1)$ to some ℓ bits, with ℓ bounded by $O(\log p_m)$, and to guarantee the correctness of obtained results. (The value of ℓ will be determined below). That is, a binary written real number $\beta = 0\,.\,b_1 b_2 b_3 \ldots$ is truncated to $\beta_{\mathrm{o}} = 0\,.\,b_1 \ldots b_\ell$. Here $b_i \in \{0, 1\}$ represents a single bit. Clearly, this involves an error of size $\varepsilon = \sum_{i=\ell+1}^{\infty} b_i \cdot 2^{-i} \leq 1/2^\ell$.

Thus, to decide whether $\alpha_X \leq 1/2$, we compute some "lower" bound α_{o} and "upper" bound α^{o}, both of length ℓ bits. α_{o} is computed according to the following mutation of (3.2):

$$\alpha_{\mathrm{o}} = {}_{[1]}\!\sum_{i=1}^{m} \left[\left({}_{[p_i]}\!\prod_{\substack{j=1 \\ j \neq i}}^{m} p_j \right)^{[p_i]^{-1}}_{[p_i]} \times\, x_i \right] \not\downarrow_\ell\, p_i \tag{3.3}$$

Here $u \not\downarrow_\ell v$, for $u, v \in \mathbb{N}$, denotes $u/v \in \mathbb{R}$, truncated to ℓ bits. Clearly, if u and v can effectively be computed in $O(\log p_m)$ space and $\ell \in O(\log p_m)$, the value of $u \not\downarrow_\ell v$ can also be computed and stored in the same space, using a "binary version" of an elementary division algorithm taught in grade schools. The same amount of space is sufficient for all remaining operations in (3.3), not producing any other numerical errors. The sum of m real numbers with errors bounded by $\varepsilon \leq 1/2^\ell$ thus gives an approximate result[5] satisfying ${}_{[1]}(\alpha_X - \alpha_{\mathrm{o}}) \leq m/2^\ell$. The upper bound is computed as follows:

$$\alpha^{\mathrm{o}} = \alpha_{\mathrm{o}} \,{}_{[1]}\!+ \tfrac{m}{2^\ell} \tag{3.4}$$

Clearly, ${}_{[1]}(\alpha^{\mathrm{o}} - \alpha_X) \leq m/2^\ell$. To keep α_{o} and α^{o} close enough, we take

$$\ell = \lceil \log(m \cdot p_m) \rceil + 1,$$

thus satisfying

$$\tfrac{m}{2^\ell} < \tfrac{1}{2 \cdot p_m}. \tag{3.5}$$

Since $m \leq p_m$, we get $\ell \in O(\log p_m)$, and hence both α_{o} and α^{o} can be computed in $O(\log p_m)$ space, by the use of (3.3) and (3.4). (See lines 2–7 in Algorithm 1).

Now we are ready to explain how the approximations α_{o} and α^{o} can be used to decide whether $X = \alpha_X \cdot M_m \leq M_m/2$. Note that if α_X is far enough from the "borders", *i.e.*, if $\alpha_X \in \langle 0 + m/2^\ell, 1 - m/2^\ell)$, then $\alpha_{\mathrm{o}} \leq \alpha_X \leq \alpha^{\mathrm{o}}$. Otherwise, for α_X very close to zero or one, $\alpha^{\mathrm{o}} < \alpha_{\mathrm{o}}$ might also be true. There are several cases to consider.

If both α_{o} and α^{o} are smaller than or equal to $1/2$, we can safely declare that $\alpha_X \leq 1/2$, and hence $X \leq M_m/2$. Similarly, if both α_{o} and α^{o} are above $1/2$, we

[5] Using ${}_{[1]}(\alpha_X - \alpha_{\mathrm{o}})$ instead of $\alpha_X - \alpha_{\mathrm{o}}$ follows from the fact that the summation in (3.3) is computed modulo 1.

1: $\ell := \lceil \log(m \cdot p_m) \rceil + 1$; ▷ Numeric precision for real arithmetic
2: **loop** $\alpha_o := 0$; $\varepsilon^o := m/2^\ell$;
3: **for** $i := 1, \ldots, m$ **do** $c := 1$; $\tilde{p} := p_i$;
4: **for** $j := 1, \ldots, m$ **do**
5: **if** $j \neq i$ **then** $c := c \,{}_{[\tilde{p}]}{\times}\, p_j$
end;
6: $c := c^{[\tilde{p}]^{-1}}$; $c := c \,{}_{[\tilde{p}]}{\times}\, x_i$; $\varphi := c \,{}_{\swarrow_\ell}\, \tilde{p}$; $\alpha_o := \alpha_o \,{}_{[1]}{+}\, \varphi$
end;
7: $\alpha^o := \alpha_o \,{}_{[1]}{+}\, \varepsilon^o$;
8: **if** $\alpha_o \leq 1/2$ and $\alpha^o \leq 1/2$ **then return** "$X \leq M_m/2$";
9: **if** $\alpha_o > 1/2$ and $\alpha^o > 1/2$ **then return** "$X > M_m/2$";
10: $m := m - 1$ ▷ Tail recursion: decide whether $\tilde{X} = (x_1, \ldots, x_{m-1}) \leq M_{m-1}/2$
end

Algorithm 1: Decide whether $X = (x_1, \ldots, x_m) \leq M_m/2$.

have $X > M_m/2$. (Lines 8 and 9 in Algorithm 1). The only problems to be decided are $\alpha_o \leq 1/2$ together with $\alpha^o > 1/2$, or $\alpha_o > 1/2$ with $\alpha^o \leq 1/2$.

First, consider $0 \leq \alpha_o \leq 1/2 < \alpha^o < 1$. Divide the segment $\langle 0, M_m)$ of the length M_m into p_m subsegments, each of equal length M_{m-1}. Recall that $M_m = p_1 \cdot \ldots \cdot p_m$, by (3.1). It is not too hard to see that both $\alpha_o \cdot M_m$ and $\alpha^o \cdot M_m$ belong to the same, namely middle, subsegment: Since $0 \leq \alpha_o \leq 1/2 < \alpha^o < 1$, we get $\alpha^o - \alpha_o = {}_{[1]}(\alpha^o - \alpha_o) = m/2^\ell < 1/(2 \cdot p_m)$, by (3.4) and (3.5). The inequality $\alpha^o \cdot M_m - \alpha_o \cdot M_m < M_{m-1}/2$ easily follows which, together with $\alpha_o \cdot M_m \leq M_m/2 < \alpha^o \cdot M_m$, proves that both $\alpha_o \cdot M_m$ and $\alpha^o \cdot M_m$ (hence, $X = \alpha_X \cdot M_m$ as well) lie inside the middle subsegment of length M_{m-1}, between $M_m/2 - M_{m-1}/2$ and $M_m/2 + M_{m-1}/2$. Therefore, we can use a tail recursion to solve the problem:

$$X = (x_1, \ldots, x_m) \leq M_m/2 \text{ iff } \tilde{X} = (x_1, \ldots, x_{m-1}) \leq M_{m-1}/2. \tag{3.6}$$

Second, if $0 \leq \alpha^o \leq 1/2 < \alpha_o < 1$, then α_X belongs to one of the intervals $\langle \alpha_o, 1)$ or $\langle 0, \alpha^o \rangle$. Also here $\langle 0, M_m)$ can be divided into p_m subsegments of equal length M_{m-1}. Since ${}_{[1]}(\alpha^o - \alpha_o) = m/2^\ell < 1/(2 \cdot p_m)$, by (3.4) and (3.5), we get that $1 - 1/(2 \cdot p_m) < \alpha_o < 1$, and also $0 \leq \alpha^o < 0 + 1/(2 \cdot p_m)$. Therefore, $\alpha_o \cdot M_m$ lies in $(M_m - M_{m-1}/2, M_m)$, while $\alpha^o \cdot M_m$ is in $\langle 0, 0 + M_{m-1}/2)$. Since $X = \alpha_X \cdot M_m$ lies "in between" $\alpha_o \cdot M_m$ and $\alpha^o \cdot M_m$ (modulo 1), we have that *(i)* if $X > M_m/2$, then X lies in the second half of the last subsegment of length M_{m-1}, and *(ii)* if $X \leq M_m/2$, then it is in the first half of the first subsegment of length M_{m-1}. Therefore, (3.6) holds true even in this case, and hence we can use the same tail recursion to solve the problem.

This kind of recursion does not increase space requirements: the value of m is decremented by one and the procedure restarts from the beginning[6]. (Line 10 in Algorithm 1).

[6]However, once we have allocated an $O(\ell)$ space for arithmetic with reals, it is fruitless to reduce this space for smaller m. On the contrary, utilizing all this space increases arithmetic precision, which results in a shorter chain of tail-recursion calls.

1: **let** $W \stackrel{\text{df.}}{=} (y_{1\ [p_1]}{-}\, x_1, \ldots, y_{m\ [p_m]}{-}\, x_m)$;
2: **if** $X > M_m/2$ **then return** "$X > Y$" ▷ Comparisons with $M_m/2$ by Algorithm 1
3: **else if** $W \le M_m/2$ **then return** "$X \le Y$"
4: **else return** "$X > Y$"

Algorithm 2: Decide whether $X \le Y$ (assuming $Y \le M_m/2$).

Quite atypically, the procedure does not need any special handling for $m = 1$. (At the "bottom level", no tail recursion can be used). This follows from the fact that, for $m = 1$, Algorithm 1 halts in one of the lines 8 or 9, never trying to execute line 10: Assume $m = 1$. Then, by (3.1), (3.2), and (3.3), we get that $M_1 = p_1 = 3$, $\alpha_X = x_1/3$, and $\alpha_{\mathrm{o}} = x_1 \not\lessapprox_\ell 3$. Moreover, by (3.4), ${}_{[1]}(\alpha^{\mathrm{o}}{-}\alpha_{\mathrm{o}}) = m/2^\ell = 1/2^\ell \le 1/8$, using also the fact that $\ell = \lceil \log(m{\cdot}p_m) \rceil + 1 \ge \lceil \log(3) \rceil + 1 = 3$. Thus, the numerical error does not exceed 1/8. (This holds even if, initially, the value of m was larger than one. Note that Algorithm 1 computes ℓ only once, using the initial value of m). Since $x_1 \in \{0, 1, 2\}$, the case analysis shows:

- if $x_1 = 0$, then $\alpha_X = 0$, $\alpha_{\mathrm{o}} = 0 \not\lessapprox_\ell 3 = 0$, and $\alpha^{\mathrm{o}} \in \langle 0, 0{+}\frac{1}{8} \rangle$;
- if $x_1 = 1$, then $\alpha_X = \frac{1}{3}$, $\alpha_{\mathrm{o}} \in \langle \frac{1}{3}{-}\frac{1}{8}, \frac{1}{3} \rangle$, and $\alpha^{\mathrm{o}} \in \langle \frac{1}{3}, \frac{1}{3}{+}\frac{1}{8} \rangle$;
- if $x_1 = 2$, then $\alpha_X = \frac{2}{3}$, $\alpha_{\mathrm{o}} \in \langle \frac{2}{3}{-}\frac{1}{8}, \frac{2}{3} \rangle$, and $\alpha^{\mathrm{o}} \in \langle \frac{2}{3}, \frac{2}{3}{+}\frac{1}{8} \rangle$.

Thus, for each x_1, either both α_{o} and α^{o} are below 1/2, or they are both above 1/2. □

Lemma 3.4. *Let $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_m)$ be two numbers in the residual representation, $Y \le M_m/2$. If the values x_i and y_i can effectively be computed in $O(\log p_m)$ deterministic space, for each given $i \in \{1, \ldots, m\}$, then the question of whether $X \le Y$ can also be decided in the same space.*

Proof. Let us begin with a small technical detail. Note that $M_m/2 \notin \mathbb{Z}$, since M_m is a product of *odd* primes, but $X \in \mathbb{Z}$. Therefore, $X \ne M_m/2$, and hence $X \le M_m/2$ if and only if $X < M_m/2$.

Now, to compare X with Y, we first compare X with $M_m/2$, using Algorithm 1 described in Lemma 3.3. If $X > M_m/2$, we are done; $X > Y$.

Conversely, for $X \le M_m/2$, thus $X < M_m/2$, it is easy to see that $X \le Y$ if and only if $W = {}_{[M_m]}(Y{-}X) \le M_m/2$. Clearly, the residual representation of this value is $W = (y_{1\ [p_1]}{-}\, x_1, \ldots, y_{m\ [p_m]}{-}\, x_m)$. By assumption, the values x_i and y_i can be computed by some deterministic procedures in $O(\log p_m)$ space. This allows us to devise a procedure computing, on demand, the value $w_i = y_{i\ [p_i]}{-}\, x_i$, for each given $i \in \{1, \ldots, m\}$, in the same space. Thus, to decide whether $X \le Y$, we compare W with $M_m/2$ by the use of Algorithm 1. (These ideas are summarized in Algorithm 2). □

Lemma 3.5. *Let $X = (x_1, \ldots, x_m)$ and $Z = (z_1, \ldots, z_m)$ be two numbers in the residual representation, $X \le Z < \sqrt{M_m}$. Moreover, $z_i \ne 0$, for each*

1: **let** $Y \stackrel{\text{df.}}{=} ({x_1}^{[p_1]^{-1}}_{[p_1]}\times z_1, \ldots, {x_m}^{[p_m]^{-1}}_{[p_m]}\times z_m)$;
2: **for** $i := 1, \ldots, m$ **do**
3: **if** $x_i = 0$ **then return** "X does not divide Z"
end;
4: **if** $Y \leq Z$ **then return** "X divides Z" ▷ $Y \leq Z$ by Algorithm 2
5: **else return** "X does not divide Z"

Algorithm 3: Decide whether X divides Z (assuming $z_i \neq 0$ and $X \leq Z < \sqrt{M_m}$).

$i \in \{1, \ldots, m\}$. *If the values x_i and z_i can effectively be computed in $O(\log p_m)$ deterministic space, for each given $i \in \{1, \ldots, m\}$, then the question of whether X divides Z can also be decided in the same space.*

Proof. Details for the procedure performing this task are shown by Algorithm 3. The algorithm first verifies whether $x_i \neq 0$, for each $i \in \{1, \ldots, m\}$. If $x_i = 0$ for some i, then X does not divide Z, since X is an integer multiple of p_i, but $Z \bmod p_i = z_i \neq 0$, by assumption.

Assume now that $x_i \neq 0$, for each i. For this case, we can easily show that X divides Z if and only if ${X^{[M_m]^{-1}}}_{[M_m]}\times Z \leq Z$:

First, if X divides Z, then there exists a $Y \in \mathbb{N}$ such that $X \times Y = Z$. Clearly, $Y \leq Z$. Moreover, $X {}_{[M_m]}\times Y = {}_{[M_m]}Z = Z$. This gives that $Y = {X^{[M_m]^{-1}}}_{[M_m]}\times Z \leq Z$. (Since p_i is a prime and $x_i \neq 0$, for each i, the value $Y = ({x_1}^{[p_1]^{-1}}_{[p_1]}\times z_1, \ldots, {x_m}^{[p_m]^{-1}}_{[p_m]}\times z_m)$ does exist and, moreover, Y is unique, for each given X and Z).

Conversely, assume that $Y = {X^{[M_m]^{-1}}}_{[M_m]}\times Z \leq Z$. This gives $X {}_{[M_m]}\times Y = Z$. Moreover, since $Y \leq Z$ and $X \leq Z$, we have $0 \leq X \times Y \leq Z^2 < M_m$. But then $X \times Y = X {}_{[M_m]}\times Y = Z$, and hence X divides Z.

Thus, to decide whether X divides Z, Algorithm 3 simply compares $Y = {X^{[M_m]^{-1}}}_{[M_m]}\times Z$ with Z, solving the problem by the use of Algorithm 2 introduced by Lemma 3.4. Since, by assumption, the values x_i and z_i can be computed by some deterministic procedures in $O(\log p_m)$ space, we can devise a subprogram computing, on demand, the value $y_i = {x_i^{[p_i]^{-1}}}_{[p_i]}\times z_i$, for each given $i \in \{1, \ldots, m\}$. Clearly, this subprogram works also in $O(\log p_m)$ space. □

We conclude this section by estimating space requirements for Algorithm 3, introduced by the above lemma. Recall that, for each given $Z = (z_1, \ldots, z_m)$, the algorithm assumes, among others, that $Z < \sqrt{M_m}$. That is, the value of m, stored initially in a global variable, must be sufficiently large so that $Z^2 < M_m = p_1 \cdot \ldots \cdot p_m$. The most natural choice would be the smallest $m \in \mathbb{N}$ satisfying $M_m > Z^2$. However, this value is hard to compute by a machine with sublogarithmic space. Therefore, we shall use a little bit larger m. Let

$$\begin{aligned} m' &= \min\{k \in \mathbb{N} : M_k > Z\}, \\ m &= 2 \cdot m'. \end{aligned} \tag{3.7}$$

Defined this way, m can be computed easily, which will be shown later.

Note that m is "sufficiently large", satisfying the initial assumption $M_m > Z^2$: since $M_{m'} = p_1 \cdot \ldots \cdot p_{m'} > Z$, we get $M_m = p_1 \cdot \ldots \cdot p_{m'} \cdot p_{m'+1} \cdot \ldots \cdot p_{2m'} > p_1^2 \cdot \ldots \cdot p_{m'}^2 > Z^2$.

On the other hand, m is not "too large", so that Algorithm 3 decides whether Z is divisible by $X \leq Z$ in space $O(\log p_m) \leq O(\log \log Z)$. To see this, note that $M_{m'-1} = p_1 \cdot \ldots \cdot p_{m'-1} \leq Z$. This gives that $2^{m'-1} \leq Z$, and hence also $m' \leq 1 + \log Z$. By the Prime Number Theorem (see, *e.g.*, Chapters 2 and 3 in [13], or [5, 23]), we have $p_m \leq O(m \cdot \log m)$, which gives $\log p_m \leq O(\log m) \leq O(\log m') \leq O(\log \log Z)$. Summing up,

$$O(\log p_m) \leq O(\log \log Z). \tag{3.8}$$

Note also that

$$p_m < Z, \text{ for each } Z \geq 18. \tag{3.9}$$

This follows from the fact that $p_m \leq m^2 = 4 \cdot m'^2 \leq 4 \cdot (1 + \log Z)^2 < Z$, for each $Z \geq 361$ (for which we have also $m \geq 8$). Analyzing the finite number of remaining cases, namely, $Z \in \{1, \ldots, 360\}$, we get that (3.9) actually holds for each $Z \geq 18$.

4. Testing primes and factoring in small space

Now we are ready to test a unary given number n for primality with $O(\log \log n)$ space. We first present a conceptually simpler deterministic Turing machine equipped with a single pebble. After that, using the power of alternation, we present a modified machine not using any pebble. This is based on the observation that, in the alternating machine, the residual representation of each X is "known" by each process with the input head at the position X, and hence such process is capable to imitate the actual position of a pebble.

4.1. Testing primes and factoring with a help of a pebble

The main idea is simple: using the residual and scalar representations together with Algorithms 1, 2, and 3, we test divisibility of the unary given input number $n = Z = (z_1, \ldots, z_m)$ by candidates $X = 2, \ldots, Z-1$, one after another. The value X is not stored on the worktape, but rather represented by a position of the movable pebble on the input tape. Recall that Algorithm 3, introduced by Lemma 3.5, needs the following prerequisites for each given X and Z satisfying $X \leq Z$:

- A value of m, sufficiently large so that $Z^2 < M_m = p_1 \cdot \ldots \cdot p_m$, must initially be stored in a global variable.
- For each $i \in \{1, \ldots, m\}$, the condition $Z \bmod p_i = z_i \neq 0$ must be satisfied.
- Finally, Algorithm 3 must be able to call two deterministic subprograms computing the values $x_i = X \bmod p_i$ and $z_i = Z \bmod p_i$, for each given parameter $i \in \{1, \ldots, m\}$, whenever the computation demands. These two subprograms do not use space above $O(\log p_m)$.

With a help of the pebble, it is not hard to fulfill these prerequisites. First, let

$$\begin{aligned} f(X) &= \text{the smallest } \textit{odd} \text{ prime not dividing } X, \\ g(Z) &= \max\{f(1), f(2), \ldots, f(Z)\}. \end{aligned} \tag{4.1}$$

We point out that the function $\log g(n)$ is a very good approximation of $\log\log n$. For each $n \geq 4$, we have $-1 \leq \lfloor \log g(n) \rfloor - \lfloor \log\log n \rfloor \leq +1$. (For argument, see[7] the proofs of Thm. 4.3, Lem. 3.1, and Thm. 6.3 in [8]. See also [7]).

Note that m' used in the next two lemmas coincides with m' defined by (3.7).

Lemma 4.1. *For each $Z \geq 1$, $g(Z) = p_{m'}$, where $m' = \min\{k \in \mathbb{N} : M_k > Z\}$.*

Proof. Clearly, by (4.1), we have that $g(Z) = p_i$, for some odd prime p_i. By definition of m', we get $M_{m'} = p_1 \cdot \ldots \cdot p_{m'} > Z \geq p_1 \cdot \ldots \cdot p_{m'-1} = M_{m'-1}$. Thus, there exists an $X \in \{1, 2, \ldots, Z\}$, namely, $X = p_1 \cdot \ldots \cdot p_{m'-1}$, such that $f(X) = p_{m'}$. Therefore, $g(Z) = \max\{f(1), f(2), \ldots, f(Z)\} \geq p_{m'}$.

On the other hand, $f(Y) \leq p_{m'}$ for each $Y \in \{1, 2, \ldots, Z\}$. Supposing the contrary, we could obtain a Y that is an integer multiple of $p_1, \ldots, p_{m'}$, and hence $Y \geq p_1 \cdot \ldots \cdot p_{m'} > Z$, which is a contraction. Therefore, $g(Z) \leq p_{m'}$, which gives $g(Z) = p_{m'}$. □

Lemma 4.2. *The values $g(Z) = p_{m'}$ and m', and hence also p_m and m, are computable by a deterministic pebble machine in space $O(\log g(Z)) = O(\log p_{m'}) \leq O(\log p_m) \leq O(\log\log Z)$.*

Proof. The basic idea has already been used, *e.g.*, in [7, 8], for a function slightly different from $g(Z)$: To compute $g(Z) = p_{m'}$, we use a machine M that systematically computes $f(X)$, for $X = 1, 2, \ldots, Z$, one by one, keeping the actual values of m'' and $p_{m''} = g = \max\{f(1), f(2), \ldots, f(X)\}$. M does not have to store the value X on the worktape, it is represented by the distance of the pebble from the left endmarker. To compute $f(X)$, M repeatedly checks if X is divisible by odd primes $p_1 = 3$, $p_2 = 5$, ... until it finds the first prime p_i not dividing X. In order to check if p_i divides X, M forms a counter on the worktape and, traversing between the left endmarker and the position indicated by the pebble, counts X modulo p_i.

Obviously, as $g = g(Z) = p_{m'}$ at the end of the computation, the machine M uses $O(\log g(Z)) = O(\log p_{m'})$ cells on its worktape. By (3.7) and (3.8), we then have $O(\log p_{m'}) \leq O(\log p_m) \leq O(\log\log Z)$. □

Now we are ready to show:

Theorem 4.3. *un-Primes $\in$ pebble–DSPACE$(\log\log n)$.*

Proof. The main idea of the procedure testing for primality is illustrated by Algorithm 4. Let Z be a number whose unary representation is given as input.

[7] Actually, the proofs presented in [7, 8] concern functions slightly different from our $g(Z)$: in [8], $f(X)$ is defined as the first prime not dividing X, thus taking also into account divisibility by the number 2. This difference needs a minor modification in the argument, which is left to the reader.

1: **if** $Z \in \{2,3,5,7,11,13,17\}$ **then return** "Z is a prime";
2: **if** $Z < 18$ **then return** "Z is not a prime";
3: **let** $f(X) \stackrel{\text{df.}}{=}$ the smallest odd prime not dividing X;
4: $g := \max\{f(1), f(2), \ldots, f(Z)\}$; ▷ $g = p_{m'}$, for some odd prime $p_{m'}$
5: $m := 2 \cdot m'$, where $p_{m'} = g$;
6: **for** $i := 0, \ldots, m$ **do**
7: **if** $Z \bmod p_i = 0$ **then return** "Z is a composite"
end;
8: **for** $X := p_m + 2, p_m + 4, p_m + 6, \ldots, Z - 2$ **do**
9: **if** X divides Z **then return** "Z is a composite" ▷ Divisibility by Algorithm 3
end;
10: **return** "Z is a prime"

Algorithm 4: Decide whether $Z \in \mathbb{N}$ is a prime.

First, our machine M' resolves the problem of a finite number of "short" inputs: at the very beginning, in an initial phase consisting of a single left-to-right traversal followed by a single right-to-left traversal, it accepts or rejects, if the input is of length $Z < 18$. (Lines 1–2 in Algorithm 4). From now on, assume that $Z \geq 18$.

For "long" inputs, M' computes $p_{m'}$ and m' by using M described in the proof of Lemma 4.2. This is needed to obtain $m := 2 \cdot m'$. (In a more sophisticated implementation, the value of ℓ can also be computed once and for all at the beginning, instead of recomputing it over and over again, in line 1 of Algorithm 1). This marks off a space of size $\Theta(\log \log Z)$ on the worktape. (Lines 3–5 in Algorithm 4).

After that, for $i = 0, 1, 2, \ldots, m$, M' repeatedly traverses the input and checks whether the prime p_i does not divide Z. If, for some i, Z is an integer multiple of p_i, then either Z is not a prime or $Z = p_i$. However, by (3.9), $Z > p_m \geq p_i$ for each $Z \geq 18$. Therefore, M' rejects Z as composite, after moving the pebble to the p_ith input tape cell.

Therefore, if Algorithm 4 does not halt in the loop nested between lines 6–7, we have $Z \bmod p_i = z_i \neq 0$, for each $i \in \{0, 1, \ldots, m\}$.

But then M' can execute Algorithm 3, introduced by Lemma 3.5, to test divisibility of Z by candidates $X = 2, \ldots, Z - 1$. The value X is represented as a distance of the pebble from the left endmarker. Whenever Algorithm 3 requires, for some parameter $i \in \{1, \ldots, m\}$, the values z_i or x_i, they are recomputed again, in space $O(\log p_m)$: M' temporarily interrupts the execution of the "main" program, computes p_i, forms a counter on the worktape and, by traversing from the left endmarker to the right endmarker (or, respectively, to the position indicated by the pebble), counts Z (or X) modulo p_i.

It should be pointed out that we can save most of the input tape traversals, which speeds up the computation: the first-level speed-up utilizes the fact that, in the loop nested between lines 6–7, we have already verified that $Z \bmod p_0 = Z \bmod 2 \neq 0$. Thus, Z is odd, and hence we can skip testing

divisibility of Z by even values[8] of X and also by values $X \leq p_m$. (Lines 8–9 in Algorithm 4).

The correctness of the described pebble machine M' follows from the above discussion, the space complexity of $O(\log \log n)$ follows from (3.8). □

It is easy to see that the pebble machine M', described above, can be modified to accept un-Composites in the same space:

Corollary 4.4. *un-Composites* $\in$ *pebble–DSPACE*$(\log \log n)$.

As follows from the construction in the Proof of Theorem 4.3, M' always halts with the pebble placed at the position X representing the smallest divisor of Z greater than one. (If $X = Z$, then Z is a prime). By equipping this pebble machine with an additional write-only output tape, it can be modified to output a binary sequence $\mathbb{D} = d_1 \dots d_Z$, with $d_x \in \{0, 1\}$ indicating the divisibility of Z by the number X.

4.2. Testing primes and factoring with a help of alternation

With the power of alternation, the role of the pebble can be imitated by input head positions of processes running in parallel. The computation tree is such that an $X \in \{2, \dots, n-1\}$ divides $n = Z$ if and only if there exists at least one accepting computation subtree with at least one of its parallel paths halting at the input tape position X.

Now we are ready for alternating counterparts of Lemma 4.2 and Theorem 4.3.

Lemma 4.5. *The values* $g(Z) = p_{m'}$ *and* m'*, and hence also* p_m *and* m*, are computable by an alternating machine in accept space* $O(\log g(Z)) = O(\log p_{m'}) \leq O(\log p_m) \leq O(\log \log Z)$.

Proof. Analogously to the pebble machine in Lemma 4.2, the construction of $g(Z)$, for the unary given input Z, is based on (4.1).

Guessing phase: Our machine N first nondeterministically picks a position $g \in \{1, \dots, Z\}$ along the input tape. After that, N moves deterministically back to the left endmarker and counts the value of g in binary on the worktape. Thus, N never allocates more space than $O(\log Z)$. However, for the right guess, the guessed value is exactly $g = g(Z) = p_{m'}$, marking off the space $1 + \lfloor \log g(Z) \rfloor \leq O(\log \log Z)$, by (3.7) and (3.8).

Verifying phase: Now N verifies the correctness of the guessed value g. This is based on the fact that $g(Z) = \max\{f(1), f(2), \dots, f(Z)\}$, and hence, for some $X \in \{1, 2, \dots, Z\}$, we have $g(Z) = f(X)$. Therefore, N moves along the input tape

[8] A second-level speed-up can also utilize the fact that $Z \bmod 3 \neq 0$, thus testing divisibility by numbers satisfying $X \bmod 6 \in \{1, 5\}$. An even more advanced version can also skip integer multiples of 5, and so on.

again, this time picking out nondeterministically the position X. To check that $g = g(Z)$, it is sufficient to verify that

- $f(X) = g$, *i.e.*, the smallest odd prime not dividing X is g; and
- for each $Y \neq X$ in $\{1, \ldots, Z\}$, $f(Y) \leq g$, *i.e.*, the smallest odd prime not dividing Y does not exceed g.

With the input head positioned at X, N first *universally* splits into 3 processes, namely, P_X, $P'_<$, and $P'_>$. The process P_X checks whether $f(X) = g$, while $P'_<$ and $P'_>$ verify if $f(Y) \leq g$, for each $Y < X$ and $Y > X$, respectively. Therefore, the process $P'_<$ moves to the left of the position X and, branching *universally* at each input tape cell, it splits into parallel processes $P'_1, \ldots, P'_{X-1}$. The process P'_Y, for $Y \neq X$, will check whether $f(Y) \leq g$. For the same reasons, $P'_>$ moves to the right of X along the input and splits universally into copies $P'_{X+1}, \ldots, P'_Z$. Note that none of the processes $P'_1, \ldots, P'_{X-1}, P_X, P'_{X+1}, \ldots, P'_Z$ stores its index on the worktape, it is merely represented by the position of the input head.

To verify if $f(X) = g$, the process P_X first checks whether g is an odd prime. If the answer is "no", P_X halts and rejects. Otherwise, branching *universally*, it writes $j \in \{1, \ldots, g\}$ in a separate track on the worktape. This splits P_X into copies $P_{X,1}, \ldots, P_{X,g}$ running in parallel. The process $P_{X,j}$, for each $j < g$, first checks whether j is an odd prime. If the answer is "no", $P_{X,j}$ halts and accepts. Otherwise, it deterministically verifies that X is divisible by j, by moving from the input position X to the left endmarker and counting X modulo j. Similarly, for $j = g$, the process $P_{X,j}$ verifies that X *is not* divisible by j.

To verify if $f(Y) \leq g$, for each $Y \neq X$, the process P'_Y *nondeterministically* chooses a $j \in \{1, \ldots, g\}$, checks whether j is an odd prime and, after that, it deterministically verifies that Y is not divisible by j, by moving from the input position Y to the left endmarker and counting Y modulo j.

Note that all parallel paths in each accepting computation subtree halt with the correct value $g = g(Z) = p_{m'}$ written on their worktapes, using $O(\log \log Z)$ space. Moreover, no rejecting computation subtree will ever try to use space above $O(\log Z)$. □

Theorem 4.6. *un-Primes* $\in$ *accept–ASPACE*$(\log \log n)$.

Proof. Let Z be the unary input. Analogously to the pebble machine of Theorem 4.3, our alternating machine N' follows the main ideas of Algorithm 4. First, N' resolves the problem of short inputs, of length $Z < 18$, by a deterministic left-to-right traversal along the input tape followed by a right-to-left traversal. (Lines 1–2 in Algorithm 4).

For $Z \geq 18$, N' computes the values $p_{m'}$ and m' by simulation of N described in Lemma 4.5. However, before switching from the *guessing phase* to the *verifying phase* (see the proof of Lem. 4.5), N' universally splits into two processes V and R. The process V verifies the correctness of the guessed value $g = g(Z) = p_{m'}$ as described in the proof of Lemma 4.5, while R follows the steps of Algorithm 4

assuming the guess is correct. That is, R starts by computing $m := 2m'$. (Lines 3–5 in Algorithm 4).

Now, branching *universally*, R spawns new processes $R'_0, R'_1, \ldots, R'_m$ running in parallel and verifying that Z is not divisible by $p_0, p_1, \ldots, p_m$. If, for some $i \in \{0, 1, \ldots, m\}$, R'_i finds that Z is an integer multiple of p_i, R'_i rejects Z as composite, after parking the input head at the p_ith cell. Otherwise, R'_i accepts with the input head at the position 1. This only requires traversing the entire input and counting modulo p_i. (Lines 6–7 in Algorithm 4).

Assuming that none of the processes $R'_0, R'_1, \ldots, R'_m$ rejects, *i.e.*, that $Z \bmod p_i = z_i \neq 0$ for each $i \in \{1, \ldots, m\}$, R examines divisibility of Z by candidates $X = 2, \ldots, Z-1$, using Algorithm 3. (Lines 8–9 in Algorithm 4). Therefore, R moves along the input tape and, branching universally at each tape cell, it splits into parallel processes $R_2, \ldots, R_{Z-1}$.

From this point forward, the process R_X keeps its input head parked at the Xth cell. With the value m stored on the worktape, R_X executes Algorithm 3 and verifies that Z cannot be divided by X. The computation of R_X is deterministic until the moment Algorithm 3 requires, for some parameter $i \in \{1, \ldots, m\}$, the values x_i or z_i. Whenever this happens, the process R_X computes p_i and nondeterministically guesses the required value $x_i \in \{0, \ldots, p_i - 1\}$ (or z_i, respectively). Then, branching universally, R_X spawns a new process X_{p_i,x_i} (or Z_{p_i,z_i}, respectively) to verify the guess, with the values p_i and x_i (or z_i) written on the worktape. After that, the process R_X resumes the execution of Algorithm 3 assuming the guessed value is correct.

The process $X_{p,x}$, starting with some values p and x on the worktape and the input head at a position X, accepts or rejects depending on whether $X \bmod p = x$. This only requires traversing from the current input tape position to the left and counting modulo p.

Similarly, the process $Z_{p,z}$, starting with p and z on the worktape, verifies if $Z \bmod p = z$. That is, after moving the input head to the right endmarker, it traverses the entire input to the left and counts modulo p.

Finally, if the process R_X, executing Algorithm 3, finds out that Z is *not* divisible by X, it accepts the input, after parking the input head at the position 1. Otherwise, it rejects, leaving the input head at the position X.

Clearly, N' does not use more space than does the machine N of Lemma 4.5, since N' uses N as its initial subprogram to allocate space on the worktape. All parallel paths in each accepting computation subtree use the same correct value of $g = g(Z) = p_{m'}$, and hence also the same worktape space of size $O(\log \log Z)$. Moreover, even a rejecting computation subtree never uses space above $O(\log Z)$. □

It is easy to modify N' so that it accepts composites: recall that N' rejects a composite Z after identifying some X as a divisor of Z, *still staying within the space bound* $O(\log \log Z)$. More precisely, a path in N' can halt in four possible states. First, it can halt in *"correct guess"* or *"wrong guess"* states, which means acceptance/rejection, entered by offsprings of V verifying whether the guessed

value $g = g(Z) = p_{m'}$ is correct. Second, the process R_X (or R'_i) halts in *"reject"* or *"accept"* states, depending on whether the number X (or p_i, respectively) divides Z. (This applies also to all spawned offsprings $X_{p,x}$ and $Z_{p,z}$). Thus, the machine accepting un-Composites can be obtained by swapping the roles of *"accept/reject"* states, but keeping the roles of *"correct guess/wrong guess"*, and by choosing one R_X or R'_i from among the processes $R_2, \ldots, R_{Z-1}$ or $R'_0, R'_1, \ldots, R'_m$ existentially rather than universally[9]. This gives:

Corollary 4.7. *un-Composites* $\in$ *accept–ASPACE*$(\log\log n)$.

5. Concluding remarks

By an easy modification, branching existentially at the very beginning, we can choose whether to simulate the machine presented by Theorem 4.6 or the one presented by Corollary 4.7. This gives a single self-verifying machine with two accepting states, namely, *"prime"* and *"composite"*, detecting whether the unary given number is prime or composite and revealing divisors, such that no accepting computation subtree uses space above $O(\log\log n)$. (Due to wrong nondeterministic guesses, some computation subtrees are not accepting. Even in this case, no path will try to use space above $O(\log n)$).

A more careful analysis reveals that, in the alternating machines above, no computation path changes the direction of input head movement more than $O(1)$ times. The product of space by input head reversals was studied in a line of research towards the simplest possible complexity classes still containing nonregular languages [4,15,20]. The class of languages accepted by alternating machines working simultaneously in accept $s(n)$ space and $i(n)$ input head reversals satisfying $s(n)\cdot i(n) \in O(h(n))$ is usually denoted by accept–ASPACE×REVERSALS$(h(n))$. Thus, we have actually shown:

Corollary 5.1. *Both un-Primes and un-Composites are contained in the class accept–ASPACE*×*REVERSALS*$(\log\log n)$.

The upper bounds presented in Theorems 4.3 and 4.6 as well as in Corollaries 4.4, 4.7, and 5.1 cannot be improved since they match the corresponding lower bounds for accepting nonregular languages [7,17,20]. We conjecture that the above alternating machines cannot be improved so that they work in strong $O(\log\log n)$ space and that, for strong–ASPACE, the lower bound for accepting un-Primes is $\Omega(\log n)$. The argument is an open problem.

However, the most challenging open problem in this area is whether we can factorize a *binary* given number deterministically in polynomial time. Since P = ASPACE$(\log n)$ [6] and we have shown here a factoring of a *unary* given number

[9] A plain swapping of all existential/universal decisions and all accepting/rejecting halting states is *not* sufficient: this gives only a machine corresponding to ASPACE$(\log n)$, since the original machine may reject in the *"wrong guess"* state after using $\log Z$ space.

working in ASPACE($\log\log n$), such factoring could, at first glance, be obtained by using some modification of the New Translation Lemma [2,3]:

Lemma 5.2. *Let $s(n) \in \Omega(\log n)$ be a fully space constructible function. Then $L \in DSPACE(s(n))$ if and only if un-$L \in$ demon–$DSPACE(\log\log n + s(\log n))$.*

In particular, for $s(n) = \log n$, a binary language L is in DSPACE($\log n$) if and only if its unary version un-L is in demon–DSPACE($\log\log n$). Though not published in the literature, it is not very difficult to show that the argument for the "$\Rightarrow$" part can be extended to alternating (and also to nondeterministic) machines. However, quite recently [16], it was shown that the "$\Leftarrow$" part does not hold for alternating machines with small space, unless P = NP.

For this reason, we cannot turn the unary machine of Corollary 4.7 into a binary ASPACE($\log n$) machine for Composites exhibiting divisors which, in turn, would have given us a deterministic polynomial time algorithm for factoring, with drastic consequences for security of cryptographic systems.

However (assuming P $\neq$ NP), this negative answer gave a unary language un-L belonging to demon–ASPACE($\log\log n$) and hence, by Lemma 4.5, to accept–ASPACE($\log\log n$), such that its binary version L is not in ASPACE($\log n$) = P. Consequently, un-L is not in demon–NSPACE($\log\log n$). This should be compared with observation made in [2, 3]: in order to show that NP is not contained in DSPACE($\log n$) = L, it suffices to present a language $L \in$ NP such that un-$L \notin$ demon–DSPACE($\log\log n$).

References

[1] M. Agrawal, N. Kayal and N. Saxena, Primes is in P. *Ann. Math.* **160** (2004) 781–93.

[2] E. Allender, The division breakthroughs. *Bull. Eur. Assoc. Theoret. Comput. Sci.* **74** (2001) 61–77.

[3] E. Allender, D.A. Mix Barrington and W. Hesse, Uniform circuits for division: Consequences and problems, in *Proc. of IEEE Conf. Comput. Complexity* (2001) 150–59.

[4] A. Bertoni, C. Mereghetti and G. Pighizzini, Strong optimal lower bounds for Turing machines that accept nonregular languages, in *Proc. of Math. Found. Comput. Sci.*, *Lect. Notes Comput. Sci.,* vol. 969. Springer-Verlag (1995) 309–18.

[5] C. Boyer, *A History of Mathematics*. John Wiley & Sons (1968).

[6] A. K. Chandra, D. C. Kozen and L. J. Stockmeyer. Alternation. *J. Assoc. Comput. Mach.* **28** (1981) 114–33.

[7] J.H. Chang, O. H. Ibarra, M.A. Palis and B. Ravikumar, On pebble automata. *Theoret. Comput. Sci.* **44** (1986) 111–21.

[8] R. Chang, J. Hartmanis and D. Ranjan. Space bounded computations: Review and new separation results. *Theoret. Comput. Sci.* **80** (1991) 289–302.

[9] A. Chiu, *Complexity of Parallel Arithmetic Using The Chinese Remainder Representation.* Master's thesis, University Wisconsin-Milwaukee (1995). (G. Davida, supervisor).

[10] A. Chiu, G. Davida and B. Litow, Division in logspace-uniform NC^1. *RAIRO: ITA* **35** (2001) 259–75.

[11] G.I. Davida and B. Litow, Fast parallel arithmetic *via* modular representation. *SIAM J. Comput.* **20** (1991) 756–65.

[12] P.F. Dietz, I.I. Macarie and J.I. Seiferas, Bits and relative order from residues, space efficiently. *Inform. Process. Lett.* **50** (1994) 123–27.

[13] W. Ellison and F. Ellison, *Prime Numbers*. John Wiley & Sons (1985).
[14] V. Geffert, Nondeterministic computations in sublogarithmic space and space constructibility. *SIAM J. Comput.* **20** (1991) 484–98.
[15] V. Geffert, C. Mereghetti and G. Pighizzini, Sublogarithmic bounds on space and reversals. *SIAM J. Comput.* **28** (1999) 325–40.
[16] V. Geffert and D. Pardubská, Unary coded NP-complete languages in ASPACE($\log\log n$), in *Proc. of Develop. Lang. Theory*, *Lect. Notes Comput. Sci.,* vol. 7410. Springer-Verlag (2012) 166–77.
[17] K. Iwama, ASPACE($o(\log\log n)$) is regular. *SIAM J. Comput.* **22** (1993) 136–46.
[18] N. Koblitz, *A Course in Number Theory and Cryptography*, *Graduate Texts in Math.*, vol. 114. Springer-Verlag (1994).
[19] I. I. Macarie, Space-efficient deterministic simulation of probabilistic automata, in *Proc. of Symp. Theoret. Aspects Comput. Sci.*, *Lect. Notes Comput. Sci.,* vol. 775. Springer-Verlag (1994) 109–22.
[20] C. Mereghetti, The descriptional power of sublogarithmic resource bounded Turing machines. In *Proc. of Descr. Compl. Formal Syst.* IFIP (2007) 12–26. (To appear in J. Automat. Lang. Combin).
[21] P. Shor, Algorithms for quantum computation: Discrete logarithms and factoring, in *Proc. of IEEE Symp. Found. Comput. Sci.* (1994) 124–34.
[22] A. Szepietowski, Turing Machines with Sublogarithmic Space, *Lect. Notes Comput. Sci.*, vol. 843. Springer-Verlag (1994).
[23] http://en.wikipedia.org/wiki/Prime_number_theorem.

Communicated by A. Bertoni.
Received December 3, 2011. Accepted June 12, 2013.