The I/O Complexity of Computing Prime Tables

Michael A. Bender[1], Rezaul Chowdhury[1], Alexander Conway[2(✉)], Martín Farach-Colton[2], Pramod Ganapathi[1], Rob Johnson[1], Samuel McCauley[1], Bertrand Simon[3], and Shikha Singh[1]

[1] Stony Brook University, Stony Brook, NY 11794-2424, USA
{bender,rezaul,pganapathi,rob,smccauley,shiksingh}@cs.stonybrook.edu
[2] Rutgers University, Piscataway, NJ 08854, USA
{alexander.conway,farach}@cs.rutgers.edu
[3] LIP, ENS de Lyon, 46 allée d'Italie, Lyon, France
bertrand.simon@ens-lyon.fr

Abstract. We revisit classical sieves for computing primes and analyze their performance in the external-memory model. Most prior sieves are analyzed in the RAM model, where the focus is on minimizing both the total number of operations and the size of the working set. The hope is that if the working set fits in RAM, then the sieve will have good I/O performance, though such an outcome is by no means guaranteed by a small working-set size.

We analyze our algorithms directly in terms of I/Os and operations. In the external-memory model, permutation can be the most expensive aspect of sieving, in contrast to the RAM model, where permutations are trivial. We show how to implement classical sieves so that they have both good I/O performance and good RAM performance, even when the problem size N becomes huge—even superpolynomially larger than RAM. Towards this goal, we give two I/O-efficient priority queues that are optimized for the operations incurred by these sieves.

Keywords: External-memory algorithms · Prime tables · Sorting · Priority queues

1 Introduction

According to Fox News [21], "Prime numbers, which are divisible only by themselves and one, have little mathematical importance. Yet the oddities have long fascinated amateur and professional mathematicians." Indeed, finding prime numbers has been the subject of intensive study for millennia.

Prime-number-computation problems come in many forms, and in this paper we revisit the classical (and Classical) problem of computing prime tables: how efficiently can we compute the table $P[a, b]$ of all primes from a to b and the table

This research was supported by NSF grants CCF 1217708, IIS 1247726, IIS 1251137, CNS 1408695, CCF 1439084, CNS-1408782, IIS-1247750, and Sandia National Laboratories.

© Springer-Verlag Berlin Heidelberg 2016
E. Kranakis et al. (Eds.): LATIN 2016, LNCS 9644, pp. 192–206, 2016.
DOI: 10.1007/978-3-662-49529-2_15

$P[N] = P[2, N]$. Such prime-table-computation problems have a rich history, dating back 23 centuries to the sieve of Eratosthenes [17,30].

Until recently, all efficient prime-table algorithms were ***sieves***, which use a partial (and expanding) list of primes to find and disqualify composites [6,7,15,30]. For example, the sieve of Eratosthenes maintains an array representing $2, \ldots, N$ and works by crossing off all multiples of each prime up to $\sqrt{N}$ starting with 2. The surviving numbers, those that have not been crossed off, comprise the prime numbers up to N.

Polynomial-time primality testing [2,18] makes another approach possible: independently test each $i \in \{2, \ldots, N\}$ (or any subrange $\{a, \ldots, b\}$) for primality. The approaches can be combined; sieving steps can be used to eliminate many candidates cheaply before relatively expensive primality tests are performed. This is a feature of the sieve of Sorenson [31] (discussed in Sect. 6) and can also be used to improve the efficiency of AKS [2] when implemented over a range.

Prime-table algorithms are generally compared according to two criteria [6,25,27,30,31]. One is the standard run-time complexity, that is, the number of RAM operations. However, when computing very large prime tables that do not fit in RAM, such a measure may be a poor predictor of performance. Therefore, there has been a push to reduce the ***working-set size***, that is, the size of memory used other than the output itself [6,11,31].[1] The hope is that if the working-set size is small enough to fit in memory for larger N, larger prime tables will be computable efficiently, though there is no direct connection between working-set size and input-output (I/O) efficiency.

Sieves and primality testing offer a trade-off between the number of operations and the working-set size of prime-table algorithms. For example, the sieve of Eratosthenes performs $O(N \log \log N)$ operations on a RAM but has a working-set size of $O(N)$. The fastest primality tests take polylogarithmic time in N, and so run in $O(N\text{polylog}N)$ time for a table but enjoy polylogarithmic working space.[2] This run-time versus working-set-size analysis has lead to a proliferation of prime-table algorithms that are hard to compare.

A small working set does not guarantee a fast algorithm for two reasons. First, eventually even slowly growing working sets will be too big for RAM. But more importantly, even if a working set is small, an algorithm can still be slow if the output table is accessed with little locality of reference.

In this paper, we analyze a variety of sieving algorithms in terms of the number of ***block transfers*** they induce, in addition to the number of operations. For out-of-core computations, these block transfers are page faults, and for smaller computations, they are cache misses. Directly counting such I/Os are often more predictive of the efficiency of an algorithm than the working set size or the instruction count.

[1] In our analyses, we model each sieving algorithm as if it writes the list of primes to an append-only output tape (i.e., the algorithm cannot read from this tape). All other memory used by the algorithm counts towards its working set size.

[2] Sieves are also less effective at computing $P[a, b]$. For primality-test algorithms, one simply checks the $b - a + 1$ candidate primes, whereas sieves generally require computing many primes smaller than a.

1.1 Computational Model

In this paper, we are interested in both the I/O complexity $\mathcal{C}_{\mathrm{I/O}}$ and the RAM complexity $\mathcal{C}_{\mathrm{RAM}}$. We indicate an algorithm's performance using the notation $\langle \mathcal{C}_{\mathrm{I/O}},\ \mathcal{C}_{\mathrm{RAM}} \rangle$.

We use the standard ***external memory*** or ***disk-access machine*** (***DAM***) model of Aggarwal and Vitter [1] to analyze the I/O complexity. The DAM model allows block transfers between any two levels of the memory hierarchy. In this paper, we denote the smaller level by ***RAM*** or ***main memory*** and the larger level by ***disk*** or ***external memory***.

In the DAM model, main memory is divided into M words, and the disk is modeled as arbitrarily large. Data is transferred between RAM and disk in blocks of B words. The I/O cost of an algorithm is the number of block transfers it induces [1,33].

We use the RAM model for counting operations. It costs $O(1)$ to compare, multiply, or add machine words. As in the standard RAM, a machine word has $\Omega(\log N)$ bits.

The prime table $P[N]$ is represented as a bit array that is stored on disk. We set $P[i] = 1$ when we determine that i is prime and set $P[i] = 0$ when we determine that i is composite. The prime table fills $O(N/\log N)$ words.[3] We are interested in values of N such that $P[N]$ is too large to fit in RAM.

1.2 Sieving to Optimize both I/Os and Operations

Let's begin by analyzing the sieve of Eratosthenes. Each prime is used in turn to eliminate composites, so the ith prime p_i touches all multiples of p_i in the array. If $p_i < B$, every block is touched. As p_i gets larger, every $\lceil p_i/B \rceil$th block is touched. We bound the I/Os by $\sum_{i=2}^{\sqrt{N}} N/(B\lceil p_i/B \rceil) \leq N \log\log N$. In short, this algorithm exhibits essentially no locality of reference, and for large N, most instructions induce I/Os. Thus, the naïve implementation of the sieve of Eratosthenes runs in $\langle \Theta(N \log\log N),\ \Theta(N \log\log N) \rangle$.

Section 2 gives descriptions of other sieves. For large N (e.g., $N = \Omega(M^2)$), most of these sieves also have poor I/O performance. For example, the segmented sieve of Eratosthenes [7] also requires $\langle \Theta(N \log\log N),\ \Theta(N \log\log N) \rangle$. The sieve of Atkin [6] requires $\langle O(N/\log\log N),\ O(N/\log\log N) \rangle$. On the other hand, the primality-checking sieve based on AKS has good I/O performance but worse RAM performance, running in $\langle \Theta\left(N/(B \log N)\right),\ \Theta(N \log^c N) \rangle$, as long as $M = \Omega\left(\log^c N\right)$.[4]

[3] It is possible to compress this table using known prime-density theorems, decreasing the space usage further.

[4] Here the representation of $P[N]$ matters most, because the I/O complexity depends on the size (and cost to scan) $P[N]$. For most other sieves in this paper, $P[N]$ is represented as a bit array and the I/O cost to scan $P[N]$ is a lower-order term.

As a lead-in to our approach given in Sect. 3, we show how to improve the I/O complexity of the naïve sieve of Eratosthenes (based on Schöhage et al.'s algorithm on Turing Machines [12,28]) as follows. Compute the primes up to $\sqrt{N}$ recursively. Then for each prime, make a list of all its multiples. The total number of elements in all lists is $O(N \log\log N)$. Sort using an I/O-optimal sorting algorithm, and remove duplicates: this is the list of all composites. Take the complement of this list. The total I/O-complexity is dominated by the sorting step, that is, $O(\frac{N}{B}(\log\log N)(\log_{M/B} \frac{N}{B}))$. Although this is a considerable improvement in the number of I/Os, the number of operations grows by a log factor to $O(N \log N \log\log N)$. Thus, this implementation of the sieve of Eratosthenes runs in $\left\langle O(\frac{N}{B}(\log\log N)(\log_{M/B} \frac{N}{B})),\ O(N \log N \log\log N)\right\rangle$.

In our analysis of the I/O complexity of diverse prime-table algorithms, one thing becomes clear. All known fast algorithms that produce prime numbers, or equivalently composite numbers, do so out of order. Indeed, sublinear sieves seem to require the careful representation of integers according to some order other than by value.

Consequently, the resulting primes or composites need to be permuted. In RAM, permuting values (or equivalently, sorting small integers) is trivial. In external memory, permuting values is essentially as slow as sorting [1]. Therefore, our results will involve sorting bounds. Until an in-order sieve is produced, all fast external-memory algorithms are likely to involve sorting.

1.3 Our Contributions

The results in this paper comprise a collection of data structures based on buffer trees [3] and external-memory priority queues [3–5] that allow prime tables to be computed quickly, with less computation than sorting implies.

We present data structures for efficient implementation of the sieve of Eratosthenes [17], the linear sieve of Gries and Misra [15] (henceforth called the GM linear sieve), the sieve of Atkin [6], and the sieve of Sorenson [31]. Our algorithms work even when $N \gg M$.

Table 1 summarizes our main results. Throughout, we use the notation $\textsc{sort}(x) = O(\frac{x}{B} \log_{M/B} \frac{x}{B})$. Thus, the I/O lower bound of permuting x elements can be written as $\min(\textsc{sort}(x), x)$ [1].

The GM linear sieve and the sieve of Atkin both slightly outperform the classical sieve of Eratosthenes. The sieve of Sorenson on the other hand induces far fewer I/O operations, but the RAM complexity is dependent on some number-theoretic unknowns, and may be far higher.

Note that the sieves of Eratosthenes and Atkins use $O(\sqrt{N})$ working space, whereas the GM Linear sieve and the sieve of Sorenson use $O(N)$ working space, which is consistent with our observation that working space is not predictive of the I/O complexity of an algorithm.

Table 1. Complexities of the main results of the paper, simplified under the assumption that N is large relative to M and B (see the corresponding theorems for the full complexities and exact requirements on N, M, and B). Note that $\text{SORT}(x) = O(\frac{x}{B}\log_{M/B}\frac{x}{B})$ is used as a unitless function, when specifying the number of I/Os in the I/O column and the number of operations in the RAM column. It is denoted by "SORT" because it matches the number of I/Os necessary for sorting in the DAM model. Here $p(N)$ is the smallest prime such that the pseudosquare $L_{p(N)} > N/(\pi(p)\log^2 N)$, and π is the prime counting function (see Sect. 6). Sorensen [31] conjectures, and the extended Riemann hypothesis implies, that $\pi(p(N))$ is polylogarithmic in N.

Sieve	I/O operations	RAM operations
Eratosthenes Sect. 3	$\text{SORT}(N)$	$B\text{SORT}(N)$
GM Linear Sect. 4	$\text{SORT}\left(\frac{N}{\log\log N}\right)$	$B\text{SORT}\left(\frac{N}{\log\log N}\right)$
Atkin Sect. 5	$\text{SORT}\left(\frac{N}{\log\log N}\right)$	$B\text{SORT}\left(\frac{N}{\log\log N}\right)$
Sorenson Sect. 6	$O(N/B)$	$O(N\pi(p(N)))$ s

2 Background and Related Work

In this Section we discuss some previous work on prime sieves. For a more extensive survey on prime sieves, we refer readers to [30].

Much of the previous work on sieving has focused on optimizing the sieve of Eratosthenes. Recall that the original sieve has an $O(N)$ working set size and performs $O(N \log\log N)$ operations. The notion of chopping up the input into intervals and sieving on each of them, referred to as the ***segmented sieve of Eratosthenes*** [7], is used frequently [6,9,11,29,30]. Segmenting results in the same number of operations as the original but with only $O(N^{1/2})$ working space. On the other hand, linear variants of the sieve [8,15,19,27] improve the operation count by a $\Theta(\log\log N)$ factor to $O(N)$, but also require a working set size of about $\Theta(N)$; see Sect. 4.

Recent advances in sieving achieve better performance. The sieve of Atkin [6] improves the operation count by an additional $\Theta(\log\log N)$ factor to $\Theta(N/\log\log N)$, with a working set of $N^{1/2}$ words [6] or even $N^{1/3}$ [6,14]; see Sect. 5.

Alternatively, a primality testing algorithm such as AKS [2] can be used to test the primality of each number directly. Using AKS leads to a very small working set size but a large RAM complexity. The sieve of Sorenson uses a hybrid sieving approach, combining both sieving and direct primality testing. This results in polylogarithmic working space, but a smaller RAM complexity if certain number-theoretic conjectures hold; see Sect. 6.

A common technique to increase sieve efficiency is preprocessing by a ***wheel sieve***, which was introduced by Pritchard [25,26]. A wheel sieve preprocesses a large set of potential primes, quickly eliminating composites with small divisors. Specifically, a wheel sieve begins with a number $W = \prod_{i=1}^{\ell} p_i$, the product of the first ℓ primes (for some ℓ). It then marks all $x < W$ that have at least

one p_i as a factor by simply testing x for divisibility by each p_i. This requires $O(\ell W)$ operations and $O(W/B \log N)$ I/Os, because marks are stored in a bit vector and the machine has a word size of $\Omega(\log N)$. The wheel sieve then uses the observation that a composite $x > W$ has a prime divisor among the first ℓ primes iff $x \bmod W$ is also divisible by that prime. Thus, the wheel iterates through each interval of W consecutive potential primes, marking off a number x iff $x \bmod W$ is marked off. When using a bit vector to store these marks, this can be accomplished by copying the first W bits into each subsequent chunk of W bits. On a machine with word size $\Omega(\log N)$, the total operations for these copies is $O(N/\log N)$, and the I/O complexity is $O(N/B \log N)$, so these costs will not affect the overall complexities of our algorithms. Typically, $\ell = \sqrt{\log N}$, so $W = N^{o(1)}$. Thus, marking off the composites less than W can be done in $N^{o(1)}$ time and $N^{o(1)}/B$ I/Os using $O(\sqrt{\log N})$ space, which will not contribute to the overall complexity of the main sieving algorithm. By Mertens' Theorem [20,32], there will be $\Theta(N/\log\log N)$ potential composites left after this pre-sieving step, which can often translate into a $\Theta(\log\log n)$ speedup to the remaining steps in the sieving algorithm.

An important component of some of the data structures presented in this paper is the priority queue of Arge and Thorup [5], which is simultaneously efficient in RAM and in external memory. In particular, their priority queue can handle inserts with $O(\frac{1}{B}\log_{M/B} N/B)$ amortized I/Os and $O(\log_{M/B} N/B)$ amortized RAM operations. Delete-min requires $O(\frac{1}{B}\log_{M/B} N/B)$ amortized I/Os and $O(\log_{M/B} N/B + \log\log M)$ amortized RAM operations. They assume that each element fits in a machine word and use integer sorting techniques to achieve this low RAM cost while retaining optimal I/O complexity.

3 Sieve of Eratosthenes

In the introduction we showed that due to the lack of locality of reference, the naïve implementation of the sieve of Eratosthenes used $\langle O(N \log\log N),\ O(N \log\log N)\rangle$. A more sophisticated approach—creating lists of the multiples of each prime, and then sorting them together—improved the locality at the cost of additional computation, leading to a cost of $\langle \textsc{sort}(N \log\log N),\ O(N \log N \log\log N)\rangle$. We can sharpen this approach by using a (general) efficient data structure instead of the sorting step, and then further by introducing a data structure designed specifically for this problem.

Using Priority Queues. The sieve of Eratosthenes can be implemented using only priority-queue operations: *insert* and *delete-min*. In this version, instead of crossing off all multiples of a discovered prime consecutively, we perform lazy inserts of these multiples into the priority queue.

The priority queue Q stores $\langle k, v\rangle$ pairs, where v is a prime and k is a multiple of v. That is, the composites are the ***keys*** in the priority queue and the corresponding prime-factor is its ***value***.[5] We start off by inserting the first pair

[5] Note that the delete-min operations of the priority queue are on the keys, i.e., the composites.

$\langle 4, 2\rangle$ into Q, and at each step, we extract (and delete) the minimum composite $\langle k, v\rangle$ pair in Q. Any number less than k which has never been inserted into Q must be prime. We keep track of the last deleted composite k', and check if $k > k' + 1$. If so, we declare $p = k' + 1$ as prime, and insert $\langle p^2, p\rangle$ into Q. In each of these iterations, we always insert the next multiple $\langle k + v, v\rangle$ into Q.

We implement this algorithm using the RAM-efficient priority queue of Arge and Thorup [5].

Lemma 1. *The sieve of Eratosthenes implemented using a RAM-efficient external-memory priority queue* [5] *has complexity* $\left\langle O(\text{SORT}(N \log \log N)), O\left(N \log \log N \left(\log_{M/B} N + \log \log M \right)\right)\right\rangle$ *and uses* $O\left(\sqrt{N}\right)$ *space for sieving primes in* $[1, N]$.

Proof. This follows from the observation that the sieve performs $\Theta\left(\sum_{\text{prime} p \in [1,\sqrt{N}]} \frac{N}{p}\right) = \Theta(N \log \log N)$ operations on Q costing $\left\langle O\left(\frac{1}{B} \log_{M/B} N\right), O\left(\log_{M/B} N + \log \log M\right)\right\rangle$ each. □

Using a Value-sensitive Priority Queue. In the above algorithm, the key-value pairs corresponding to smaller values are accessed more frequently because smaller primes have more multiples in a given range. Therefore, a structure that prioritizes the efficiency of operations on smaller primes (values) outperforms a generic priority queue. We introduce a ***value-sensitive priority queue***, in which the amortized access cost of an operation with value v depends on v instead of the size of the data structure.

A value-sensitive priority queue Q has two parts—the *top part* consisting of a single internal-memory priority queue Q' and the *bottom part* consisting of $\lceil \log \log N \rceil$ external-memory priority queues $Q_1, Q_2, \ldots, Q_{\lceil \log \log N \rceil}$.

Each Q_i in the bottom-part of Q is a RAM-efficient external-memory priority queue [5] that stores $\langle k, v\rangle$ pairs, for $v \in [2^{2^i}, 2^{2^{i+1}})$. Hence, each Q_i contains fewer than $N_i = 2^{2^{i+1}}$ items. With a cache of size M, Q_i supports insert and delete-min operations in $\left\langle O((\log_{M/B} N_i)/B),\ O(\log_{M/B} N_i + \log \log M)\right\rangle$ amortized cost [5]. Moreover, in each Q_i we have $\log v = \Theta(\log N_i)$. Thus, the cost reduces to $\left\langle O((\log_{M/B} v)/B),\ O(\log_{M/B} v + \log \log M)\right\rangle$ for an item with value v. Though we divide the cache equally among all Q_i's, the asymptotic cost per operation remains unchanged assuming $M > B(\log \log N)^{1+\varepsilon}$ for some constant $\varepsilon > 0$.

The queue Q' in the top part only contains the minimum composite (key) item from each Q_i, and so the size of Q' will be $\Theta(\log \log N)$. We use the dynamic integer set data structure [22] to implement Q' which supports insert and delete-min operations on Q' in $O(1)$ time using only $O(\log n)$ space. We also maintain an array $A[1 : \lceil \log \log N \rceil]$ such that $A[i]$ stores Q_i's contributed item to Q'; thus we can access it in constant time.

To perform a delete-min, we extract the minimum key item from Q', check its value to find the Q_i it came from, extract the minimum key item from that

Q_i and insert it into Q'. To insert an item , we first check its value to determine the destination Q_i, compare it with the item in $A[i]$, and depending on the result of the comparison we either insert the new item directly into Q_i or move Q_i's current item in Q' to Q_i and insert the new item into Q'. The following lemma summarizes the performance of these operations.

Lemma 2. *Using a value-sensitive priority queue Q as defined above, inserting an item with value v takes $\left\langle O((\log_{M/B} v)/B), O(\log_{M/B} v)\right\rangle$, and a delete-min that returns an item with value v takes $\left\langle O((\log_{M/B} v)/B),\ O(\log_{M/B} v + \log\log M)\right\rangle$, assuming $M > \log N + B(\log\log N)^{1+\varepsilon}$ for some constant $\varepsilon > 0$.*

We now use this value-sensitive priority queue to efficiently implement the sieve of Eratosthenes. Each prime p is involved in $\Theta(N/p)$ priority queue operations, and by the Prime Number Theorem [16], there are $O(\sqrt{N}/\log N)$ prime numbers in $[1, \sqrt{N}]$, and the ith prime number is approximately $i \ln i$. Theorem 1 now follows.

Theorem 1. *Using a value-sensitive priority queue, the sieve of Eratosthenes runs in $\left\langle \text{SORT}(N),\ O(N(\log_{M/B} N + \log\log M \log\log N))\right\rangle$ and uses $O(\sqrt{N})$ space, provided $M > \log N + B(\log\log N)^{1+\varepsilon}$ for some constant $\varepsilon > 0$.*

We can simplify this to $\langle \text{SORT}(N),\ B\text{SORT}(N)\rangle$ if $\log N/\log\log N = \Omega(\log(M/B)\log\log M)$ and $\log(N/B) = \Omega(\log N)$.

4 Linear Sieve of Gries and Misra

There are several variants of the sieve of Eratosthenes [8,13,15,19] that perform $O(N)$ operations by only marking each composite exactly once; see [27] for a survey. We will focus on one of the linear variants, the GM linear sieve [15]. Other linear-sieve variants, such as [8,13,19] share the same underlying data-structural operations, and much of the basic analysis below carries over.

The GM linear sieve is based on the following basic property of composite numbers: each composite C can be represented uniquely as $C = p^r q$ where p is the smallest prime factor of C, and either $q = p$ or p does not divide q [15].

Thus, each composite has a unique normal form based on p, q and r. Crossing off the composites in a lexicographical order based on these (p, q, r) ensures that each composite is marked exactly once. Thus the RAM complexity is $O(N)$.

Algorithm 1 describes the linear sieve in terms of subroutines. It builds a set $\mathcal{C}$ of composite numbers, then returns its complement.

The subroutine `Insert` $(x, \mathcal{C})$ inserts x in $\mathcal{C}$. Inverse successor (`InvSucc`$(x, \mathcal{C})$ returns the smallest element larger than x that is not in $\mathcal{C}$.

```
C ← {1};  p ← 1;
while p ≤ √N do
    p ← InvSucc(p,C );  q ← p;
        while q ≤ N/p do
            for r = 1, 2, . . . , log_p(N/q)
            do
            Insert(p^r q, C );
            q ← InvSucc(q, C );
    return [1; N] \ C
```

Algorithm 1. GM Linear Sieve

While the RAM complexity is an improvement by a factor of $\log \log N$ over the classic sieve of Eratosthenes, the algorithm (thematically) performs poorly in the DAM model. Even though each composite is marked exactly once, resulting in $O(N)$ operations, the overall complexity of this algorithm is $\langle O(N), O(N) \rangle$, as a result poor data locality. In the rest of the section we improve the locality using a "buffer-tree-like" data structure, while also taking advantage of the bit-complexity of words to improve the performance further.

Using a Buffer Tree. We first introduce the classical buffer tree of Arge [3], and then modify the structure to improve the bounds of the GM linear sieve. We give a high-level overview of the data structure here.

The classical buffer tree has branching factor M/B, with a buffer of size M at each node. We assume a complete tree for simplicity, so its height is $\lceil \log_{M/B} N/M \rceil = O(\log_{M/B} N/B)$. Newly-inserted elements are placed into the root buffer. If the root buffer is full, all of its elements are flushed: first sorted, and then placed in their respective children. This takes $\langle O(M/B), O(M \log M) \rangle$. This process is then repeated recursively as necessary for the buffer of each child. Since each element is only flushed to one node at each level, and the amortized cost of a flush is $\langle O(1/B), O(\log M) \rangle$, the cost to flush all elements is $\left\langle O(N/B \log_{M/B} N/B), O(N \log N) \right\rangle$.

Inverse successor can be performed by searching within the tree. However, these searches are very expensive, as we must search every level of the tree—it may be that a recently-inserted element changed the inverse successor. Thus it costs at least $\left\langle O(M/B \log_{M/B} N/B), O(M \log_{M/B} N/B) \right\rangle$ for a single inverse successor query.

Using a Buffer-tree-like Structure. In order to achieve better bounds, we will need to improve the inverse successor time to match the insert time. It turns out that this will also improve the computation time considerably; we will only do $O(B)$ computations per I/O, the best possible for a given I/O bound.

As an initial optimization, we perform a wheel sieve using the primes up to $\sqrt{\log N}$. By an analogue of Merten's Theorem, this leaves only $N/\log \log N$ candidate primes. This reduces the number of insertions into the buffer tree.

To avoid the I/Os along the search path for the inverse successor queries, we adjust the branching factor to $\sqrt{M/B}$ rather than M/B, which doubles the height, and partition each buffer into $\sqrt{M/B}$ subarrays of size $\sqrt{MB}$: one for each child. Then as we scan the array, we can store the path from the root to the current leaf in $\sqrt{MB} \log_{M/B} N/B$ words. If $\sqrt{M/B} > \log_{M/B} N/B$ this path fits in memory. Thus, the inverse successor queries can avoid the path-searching I/O cost without affecting the amortized insert cost.

Next, since the elements of the leaves are consecutive integers, each can be encoded using a single bit, rather than an entire word. Recall that we can read $\Omega(B \log N)$ of these bits in a single block transfer. This could potentially speed up queries, but only if we can guarantee that the inverse successor can always be found by scanning *only* the bit array. However, during an inverse successor scan, we already maintain the path in memory; thus, we can flush all elements

along the path without any I/O cost. Therefore we can in fact get the correct inverse successor by scanning the array.

As an bonus, we can improve the RAM complexity during a flush. Since our array is static and the leaves divide the array evenly, we can calculate the child being flushed to using modular arithmetic.

In total, we insert $N/\log\log N$ elements into the buffer tree. Each must be flushed through $O(\log_{M/B} N/B)$ levels, where a flush takes $\langle O(1/B),\ O(1)\rangle$ amortized. The inverse successor queries must scan through $N \log\log N$ elements (by the analysis of the sieve of Eratostheses), but due to our bit array representation this only takes $\langle O(N \log\log N/B \log N),\ O(N \log\log N/\log N)\rangle$, a lower-order term.

Theorem 2. *The GM linear sieve implemented using our modified buffer tree structure, assuming* $M > B^2$, $\sqrt{M/B} > \log_{M/B}(N/B)$, *and* $\sqrt{M/B} > \log^2_{M/B}(N/B)/\log\log N$, *uses* $O(N)$ *space and has a complexity of* $\langle \textsc{sort}(N/\log\log N),\ B\,\textsc{sort}(N/\log\log N)\rangle$.

Using Priority Queues. The GM linear sieve can also be implemented using a standard priority queue API. While any priority-queue of choice can be used, the RAM- and I/O-efficient priority queue of Arge and Thorup [5] in particular achieves the same bounds as the modified buffer tree implementation.

The two data structures presented to implement the GM linear sieve offer a nice contrast. The buffer tree approach is self-contained and designed specifically for sieving, while the PQ based approach offers flexibility to use a PQ of your choice. The RAM-efficient PQ [5], in particular, is based on integer sorting techniques, while the buffer tree avoids such heavy machinery. We sketch the PQ-based version here for completeness.

The basic algorithm is the same (Algorithm 1), that is, enumerate composites in their unique normal form $p^r q$. However, in this variant, `InvSucc` is implemented using only insert and delete-min operations.

In contrast to the buffer tree approach where we build the entire set of composites $\mathcal{C}$ and eventually return its complement, we maintain a running list of potential primes as a priority queue $\mathcal{P}$. As the primes are discovered, we extract them from $\mathcal{P}$ and output. The composites $p^r q$ generated by the GM linear sieve algorithm are temporarily stored in another priority queue $\mathcal{C}$. We ensure locality of reference by lazily deleting the discovered composites in $\mathcal{C}$ from $\mathcal{P}$. In particular, we update $\mathcal{P}$ every time `InvSucc` is called, just as much as is required to find the next candidate for p or q, by using delete-min operations on $\mathcal{P}$ and $\mathcal{C}$.

Theorem 3. *The GM linear sieve implemented using RAM-efficient priority queues* [5], *assuming* $N > 2M$ *and* $M > 2B$, *uses* $O(N)$ *space and has a complexity of* $\left\langle \textsc{sort}\left(\frac{N}{\log\log N}\right),\ \frac{N}{\log\log N}\left(\log_{\frac{M}{B}} \frac{N}{B} + \log\log M\right)\right\rangle$.

We can simplify this to $\left\langle \textsc{sort}\left(\frac{N}{\log\log N}\right),\ B\ \textsc{sort}\left(\frac{N}{\log\log N}\right)\right\rangle$ *if* $\log N > \log M \log\log M$.

5 Sieve of Atkin

The sieve of Atkin [6,12] is one of the most efficient known sieves in terms of RAM computations. It can compute all the primes up to N in $O(N/\log\log N)$ time using $O(\sqrt{N})$ memory. We first describe the original algorithm from [6] and then use various priority queues to improve its I/O efficiency.

The algorithm works by exploiting the following characterization of primes using binary quadratic forms. Note that every number that is not trivially composite (divisible by 2 or 3) must satisfy one of the three congruences. For an excellent introduction to the underlying number theoretic concepts, see [10].

Theorem 4 [6]. *Let k be a square-free integer with $k \equiv 1 \pmod 4$ (resp. $k \equiv 1 \pmod 6$, $k \equiv 11 \pmod{12}$) . Then k is prime if and only if the number of positive solutions to $x^2 + 4y^2 = k$ (resp. $3x^2 + y^2 = k$, $3x^2 - y^2 = k$ $(x > y)$) is odd.*

For each quadratic form $f(x, y)$, the number of solutions can be computed by brute force in $O(N)$ operations by iterating over the set $L = \{(x, y) \mid 0 < f(x, y) \leq N\}$. This can be done with a working set size of $O(\sqrt{N})$ by "tracing" the level curves of f. Then, the number of solutions that occur an even number of times are removed, and by precomputing the primes less than $\sqrt{N}$, the numbers that are not square-free can be sieved out leaving only the primes as a result of Theorem 4.

The algorithm as described above requires $O(N)$ operations, as it must iterate through the entire domain L. This can be made more efficient by first performing a wheel sieve. If we choose $W = 12 \cdot \prod_{p^2 \leq \log N} p$, then by an analog of Mertens' theorem, the proportion of (x, y) pairs with $0 \leq x, y < W$ such that $f(x, y)$ is a unit mod W is $1/\log\log N$. By only considering the W-translations of these pairs we obtain $L' \subseteq L$, with $|L'| = O(N/\log\log N)$ and $f(x, y)$ composite on $L \setminus L'$. The algorithm can then proceed as above.

Using Priority Queues. The above algorithm and its variants require that $M = \Omega(\sqrt{N})$. By utilizing a priority queue to store the multiplicities of the values of f over L, as well as one to implement the square-free sieve, we can trade this memory requirement for I/O operations. In what follows we use an analog of the wheel sieve optimization described above, however we note that the algorithm and analysis can be adapted to omit this.

Having performed the wheel sieve as described above, we insert the values of each quadratic form f over each domain L into an I/O- and RAM-efficient priority queue Q [5]. This requires $|L|$ such operations (and their subsequent extractions), and so this takes $\langle \text{SORT}(|L|), O(|L| \log_{M/B} |L| + |L| \log\log M/\log\log N) \rangle$. Because we have used a wheel sieve, $|L| = O(N/\log\log N)$, and so this reduces to

$$\left\langle \text{SORT}\left(\frac{N}{\log\log N}\right),\ O\left(\frac{N \log_{M/B} N}{\log\log N} + \frac{N \log\log M}{\log\log N}\right)\right\rangle. \tag{1}$$

The remaining entries in Q are now either primes or squareful numbers. In order to remove the squareful numbers, we sieve the numbers in Q as follows.

We maintain a separate I/O- and RAM-efficient priority queue Q' of pairs $\langle v, p\rangle$, where $p \leq \sqrt{N}$ is a previously discovered prime and v is a multiple of p^2. For each value v we pull from Q, we repeatedly extract the min value $\langle w, p\rangle$ from Q' and insert $\langle w + p^2, p\rangle$ until either v is found, in which case v is not square-free and thus not a prime, or exceeded, in which case v is prime. If v is a prime, then we insert $\langle v^2, v\rangle$ into Q'.

Each prime $p \leq \sqrt{N}$ will be involved in at most N/p^2 operations on Q', and so will contribute $\left\langle O(\frac{N \log_{M/B} N}{p^2 B}),\ O(\frac{N}{p^2}(\log_{M/B} N + \log\log M)\right\rangle$ operations. Summing over p, the total number of operations in this phase of the algorithm is less than $\langle O\left(\text{SORT}\left(N\right)/(B \log N)\right),\ O\left((\text{SORT}\left(N\right) + \log\log M)/\log N\right)\rangle$.

As described above, the priority queue Q may contain up to N items. We can reduce the max size of Q to $O(\sqrt{N})$ by tracing the level curves much like the sieve of Atkin.

Theorem 5. *The sieve of Atkin implemented with a wheel sieve, as well as I/O and RAM efficient priority queues runs in* $\left\langle \text{SORT}\left(N/\log\log N\right),\ O((N \log_{M/B} N)/\log\log N + N \log\log M/\log\log N)\right\rangle$, *using* $O(\sqrt{N})$ *space.*

We can simplify this to $\langle \text{SORT}\left(N/\log\log N\right),\ B\ \text{SORT}\left(N/\log\log N\right)\rangle$ *if* $\log N = \Omega(\log(M/B)\log\log M)$ *and* $\log N/B = \Omega(\log N)$.

6 Sieve of Sorenson

The sieve of Sorenson [31] uses a hybrid approach. It first uses a wheel sieve to remove multiples of small primes. Then, it eliminates non-primes using a test based on so called pseudosquares. Finally it removes composite prime powers with another sieve.

The ***pseudosquare*** L_p is the smallest non-square integer with $L_p \equiv 1 \pmod 8$ that is a quadratic residue modulo every odd prime $q \leq p$. The sieve of Sorenson is based on the following theorem in that its steps satisfy each requirement of the theorem explicitly.

Theorem 6 [31]. *Let x and s be positive integers. If the following hold:*

(i) *All prime divisors of x exceed s,*
(ii) *$x/s < L_p$, the p-th pseudosquare for some prime p,*
(iii) *$p_i^{(x-1)/2} \equiv \pm 1 \pmod x$ for all primes $p_i \leq p$,*
(iv) *$2^{(x-1)/2} \equiv -1 \pmod x$ when $x \equiv 5 \pmod 8$,*
(v) *$p_i^{(x-1)/2} \equiv -1 \pmod x$ for some prime $p_i \leq p$ when $x \equiv 1 \pmod 8$,*

then x is a prime or a prime power.

The algorithm first sets $s = \lceil\sqrt{\log N}\rceil$. It then chooses $p(N)$ so that $L_{p(N)}$ is the smallest pseudosquare satisfying $L_{p(N)} > N/s$. Thus, the algorithm must calculate $L_{p(N)}$. We omit this calculation; see [31] for an $o(N)$ algorithm to do so.

A table of the first 73 pseudosquares is sufficient for any $N < 2.9 \times 10^{24}$.[6] Next, the algorithm calculates the first s primes. We assume that $M \gg \pi(p(N))$.

The algorithm proceeds in three phases. Sorenson's original algorithm segments the range in order to fit in cache, but this step is omitted here:

1. Perform a (linear) wheel sieve to eliminate multiples of the first s primes.[7] All remaining numbers satisfy the first requirement of Theorem 6.
2. For each remaining k:
   - It verifies that $2^{(k-1)/2} \equiv \pm 1 \pmod{k}$ and is -1 if $k \equiv 5 \bmod 8$.
   - If k passes the above test, then it verifies that $p_i^{(k-1)/2} \equiv \pm 1 \pmod{k}$ for all odd primes $p_i \leq p(N)$, and that $p_i^{(k-1)/2} \equiv -1 \pmod{k}$ for at least one p_i if $k \equiv 1 \pmod 8$.

   Note that this second test determines if the remaining requirements of Theorem 6 are met.
3. Remove all prime powers, as follows. If $N \leq 6.4 \times 10^{37}$, only primes remain and this phase is unnecessary [31,34]. Otherwise construct a list of all the perfect powers less than N by repeatedly exponentiating every element of the set $\{2, \ldots, \lfloor \sqrt{N} \rfloor\}$ until it is greater than N. Sort these $O(\sqrt{N} \log N)$ elements and remove them from the prime candidate list.

The complexity of this algorithm is dominated by step 2. To analyze the RAM complexity, first note that only $O(N/\log\log N)$ elements remain after the wheel sieve. Performing each base 2 pseudoprime test takes $O(\log N)$ time, so the cumulative total is $O(N \log N/\log\log N)$. Now, only $O(N/\log N)$ numbers up to N pass the base-2 pseudoprime test (see e.g. [23,31]). For each of the remaining integers, we must do $\pi(p(N))$ modular exponentiations (to a power less than N), which requires a total of $O(N\pi(p(N)))$ operations. Thus we get a total cost of $O(N\pi(p(N)) + N \log N/\log\log N))$

We can remove the second term using recent bounds on pseudoprimes. Pomerance and Shparlinski [24] have shown that $L_p(N) \leq \exp(3p(N)/\log\log p(N))$. Thus, $N \log N/\log\log N = O(N\pi(p(N))/\log\log p(N))$, and so the running time simplifies to $O(N\pi(p(N)))$.

Theorem 7. *The sieve of Sorenson runs in* $\langle O\left(\frac{N}{B}\right),\ O\left(N\pi(p(N))\right)\rangle$.

We can phrase the complexity in terms of N alone by bounding p. The best known bound for p leads to a running time of roughly $O(N^{1.1516})$. On the other hand, the Extended Riemann Hypothesis implies $p < 2\log^2 N$, and Sorenson conjectures that $p \sim \frac{1}{\log 2} \log N \log\log N$ [31]; under these conjectures the RAM complexity is $O(N \log^2 N/\log\log N)$ and $O(N \log N)$ respectively.

Sieving an Interval. Note that a similar analysis shows we can efficiently sieve an interval with the sieve of Sorenson as well.

[6] These tables are available online. For example, see https://oeis.org/A002189/b002189.txt.

[7] Sorenson's exposition removes multiples of the small primes one by one on each segment in order to retain small working space. From an external memory point of view, building the whole wheel of size $N^{o(1)}$ is also effective.

Acknowledgments. We thank Oleksii Starov for suggesting this problem to us.

References

1. Aggarwal, A., Vitter, S.: Jeffrey: the input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
2. Agrawal, M., Kayal, N., Saxena, N.: Primes is in P. Ann. Math. **50**, 781–793 (2004)
3. Arge, L.: The buffer tree: a technique for designing batched external data structures. Algorithmica **37**(1), 1–24 (2003)
4. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: Cache-oblivious priority queue and graph algorithm applications. In: Proceedings of the 34th Annual Symposium on Theory of Computing, pp. 268–276 (2002)
5. Arge, L., Thorup, M.: RAM-efficient external memory sorting. In: Cai, L., Cheng, S.-W., Lam, T.-W. (eds.) Algorithms and Computation. LNCS, vol. 8283, pp. 491–501. Springer, Heidelberg (2013)
6. Atkin, A., Bernstein, D.: Prime sieves using binary quadratic forms. Math. Comput. **73**(246), 1023–1030 (2004)
7. Bays, C., Hudson, R.H.: The segmented sieve of Eratosthenes and primes in arithmetic progressions to 1012. BIT Numer. Math. **17**(2), 121–127 (1977)
8. Bengelloun, S.: An incremental primal sieve. Acta Informatica **23**(2), 119–125 (1986)
9. Brent, R.P.: The first occurrence of large gaps between successive primes. Math. Comput. **27**(124), 959–963 (1973)
10. Cox, D.A.: Primes of the Form x^2+ny^2: Fermat, Class Field Theory, and Complex Multiplication. Wiley, New York (1989)
11. Dunten, B., Jones, J., Sorenson, J.: A space-efficient fast prime number sieve. IPL **59**(2), 79–84 (1996)
12. Farach-Colton, M., Tsai, M.-T.: On the complexity of computing prime tables. In: Elbassioni, K., Makino, K. (eds.) ISAAC 2015. LNCS, vol. 9472, pp. 677–688. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48971-0_57
13. Gale, R., Pratt, V.: CGOL-an Algebraic Notation for MACLISP Users. MIT Artificial Intelligence Library, Cambridge (1977)
14. Galway, W.F.: Dissecting a sieve to cut its need for space. In: Bosma, W. (ed.) ANTS-IV. LNCS, vol. 1838, pp. 297–312. Springer, Heidelberg (2000)
15. Gries, D., Misra, J.: A linear sieve algorithm for finding prime numbers. Commun. ACM **21**(12), 999–1003 (1978)
16. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers. Oxford University Press, Oxford (1979)
17. Horsley, S.: ΚΟΣΚΙΝΟΝ ΕΡΑΤΟΣΘΕΝΟΥΣ. or, the sieve of eratosthenes. being an account of his method of finding all the prime numbers, by the Rev. Samuel Horsley, FRS. Philos. Trans. **62**, 327–347 (1772)
18. Lenstra Jr., H.W.: Primality testing with gaussian periods. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 1–1. Springer, Heidelberg (2002)
19. Mairson, H.G.: Some new upper bounds on the generation of prime numbers. Commun. ACM **20**(9), 664–669 (1977)
20. Mertens, F.: Ein beitrag zur analytischen zahlentheorie. J. fr die reine und angewandte Mathematik **78**, 46–62 (1874)
21. News, F.: World's largest prime number discovered - all 17 million digits, February 2013. https://web.archive.org/web/20130205223234/, http://www.foxnews.com/science/2013/02/05/worlds-largest-prime-number-discovered/

22. Patrascu, M., Thorup, M., Dynamic integer sets with optimal rank, select, predecessor search. In: FOCS, pp. 166–175 (2014)
23. Pomerance, C., Selfridge, J.L., Wagstaff, S.S.: The pseudoprimes to $25 \cdot 10^9$. Math. Comput. **35**(151), 1003–1026 (1980)
24. Pomerance, C., Shparlinski, I.E.: On pseudosquares and pseudopowers. Comb. Number Theor., 171–184 (2009)
25. Pritchard, P.: A sublinear additive sieve for finding prime number. Commun. ACM **24**(1), 18–23 (1981)
26. Pritchard, P.: Explaining the wheel sieve. Acta Informatica **17**(4), 477–485 (1982)
27. Pritchard, P.: Linear prime-number sieves: a family tree. Sci. Comput. Program. **9**(1), 17–35 (1987)
28. Schönhage, A., Grotefeld, A., Vetter, E.: Fast algorithms: a multitape turing machine implementation. Wissenschaftsverlag, B.I (1994)
29. Singleton, R.C.: Algorithm 357: an efficient prime number generator. Commun. ACM **12**, 563–564 (1969)
30. Sorenson, J.: An introduction to prime number sieves. Technical report 909, Computer Sciences Department, University of Wisconsin-Madison (1990)
31. Sorenson, J.P.: The pseudosquares prime sieve. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 193–207. Springer, Heidelberg (2006)
32. Villarino, M.B.: Mertens' proof of mertens' theorem. arXiv:math/0504289 (2005)
33. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. (CsUR) **33**(2), 209–271 (2001)
34. Williams, H.C.: Edouard Lucas and primality testing. Canadian Mathematics Society Series of Monographs and Advanced Texts, 22 (1998)