The I/O Complexity of Computing Prime Tables

Michael A. Bender[1], Rezaul Chowdhury[1], Alex 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.
{farach,alexander.conway}@cs.rutgers.edu
[3] LIP, ENS de Lyon, 46 allée d'Italie, Lyon, France.
bertrand.simon@ens-lyon.fr

Abstract. We revisit classical primes sieves 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. One reason for parameterizing by working-set size is that if the working set fits in RAM, then there is a better chance that the sieve has good I/O performance.

We analyze our algorithms directly in terms of I/Os and operations. Unlike in the RAM model, where permutation is trivial, in the external-memory model, permutation can be the most expensive aspect of sieving. 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—superpolynomially larger than RAM. Towards this goal, we give two I/O-efficient priority queues that are optimized for the number of operations incurred by these sieves.

Keywords: External Memory Algorithms, Prime Tables, Sorting, Priority Queues

1 Introduction

According to Fox News [20], "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 $P[N] = P[2, N]$. Such prime-table-computation problems have a rich history, dating back 23 centuries to the sieve of Eratosthenes [17, 27].

Until recently, all efficient prime-table algorithms were ***sieves***, which use a partial (and expanding) list of primes to find and disqualify composites [6, 8, 15, 27]. 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 haven't 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. Nevertheless, sieving steps can be used to cheaply eliminate many candidates before the relatively

expensive tests are performed, thus improving their performance. This is a feature of the sieve of Sorenson [28] (discussed in Section 5), 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, 23, 24, 27, 28]. One is the standard run-time complexity, that is, the number of operations such algorithms take in RAM. 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 table itself [6, 12, 28]. The idea is that if the working-set size is smaller, it will fit in memory for larger N, thus allowing larger prime tables to be computed efficiently.

Sieves and primality testing offer a tradeoff 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 uses a working space of size $O(N)$. The fastest primality tests take polylogarithmic time in N, and so run in $O(N \text{polylog} N)$ time, but enjoy polylogarithmic working space. Sieves are also less effective at computing $T[a, b]$. For primality-test algorithms, one simply checks the $b - a + 1$ candidate primes, whereas sieves generally require computing many primes smaller than a.

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. This run-time versus working-set-size analysis has lead to a proliferation of prime-table algorithms that are hard to compare.

In this paper, we analyze a variety of algorithms in terms of the number of block transfers they induce, in addition to the number of operations. We use the standard ***disk access machine*** (***DAM***) model [1] (also called the ***external-memory model*** or ***I/O model***). For out-of-core computations, these block transfers are page faults, and for smaller computations, they are cache misses. The DAM model is often more predictive of the efficiency of an algorithm than the size of the working set or of the instruction count, since it directly counts all I/Os, both on the working set and the output array.

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.

As a lead-in to our work in Section 2, we can improve the I/O complexity of the sieve of Eratosthenes 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, so the time 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, it represents a slowdown in the number of operations, which increases by a log factor to $O(N \log N \log\log N)$.

In our analysis of the I/O complexity of diverse prime-table algorithms in this paper, one thing becomes clear. All known fast algorithms produce prime numbers, or equivalently composite numbers, out of order. Indeed, it seems to be the careful

representation of integers according to some order other than by value that allows for sublinear sieves.

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.

Our main result is 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.

1.1 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 [27].

Much of previous work on sieving has focused on optimizing the sieve of Eratosthenes. Recall that the original sieve uses $O(N)$ working space 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*** [8], is frequently used in practice [6, 10, 12, 26, 27]. It performs the same number of operations as the original but with only $O(\sqrt{N})$ working space. On the other hand, linear variants of the sieve [9, 15, 19, 25] improve the operation count by a $\Theta(\log\log N)$ factor to $O(N)$, but also require a working set of $\Theta(N)$; see Section 3.

Recent advances in sieving use new approaches to achieve better performance. We discuss the sieves of Atkin and Sorenson in Sections 4 and 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 very small working set size but a large computation cost. On the other hand, the sieve of Sorenson uses a hybrid sieving approach, including elements of both sieving and direct primality testing. This results in polylogarithmic working space, but a larger RAM complexity (see Section 5 for details).

A common technique to increase sieve efficiency is a ***wheel sieve***. 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, which is a product of the first p primes (for some p). All multiples less than W of the first p primes are marked. Note that if $x < W$ is composite, then $x + W$ is composite as well (since x and W must share a divisor). Thus we iterate through each interval of W consecutive potential primes, marking off certain composites. Since this is just a scan, it takes at most linear work and I/Os, and marks off all composites divisible by one of the first p primes. We will use this technique in Sections 3 and 4. See, for example, [6] for more details.

Previously, Arge and Thorup created a priority queue that is simultaneously efficient in RAM and external memory [5]. We use this data structure as a black box in Sections 2 and 4. Their results also provide an alternative to our priority queue in Section 3.

Specifically, the bounds in Theorem 3 can be achieved by both Arge and Thorup's priority queue, and the priority queue presented in Section 3; however, there are several distinctions. The data structure in [5] requires $M < N/2$ (an upper bound on M) whereas ours requires $\sqrt{M/B} > \log_{M/B} N/B$ (a lower bound on M). Thus, the approaches are complimentary, covering different ranges of M while achieving the same bounds.

Arge and Thorup's priority queue also differs substantially in structure. Their priority queue is based on integer sorting techniques to lower the RAM complexity, whereas ours uses properties of our specific sequence of inserts. Thus our priority queue avoids the heavy machinery of integer sorting, but is only applicable in this specific context. It would be interesting to further explore the relationship between these techniques.

1.2 External-Memory Model and Prime Tables

We analyze our sieves using the ***external memory*** or ***disk-access machine*** (***DAM***) model of Aggarwal and Vitter [1]. The DAM model focuses on the 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***.

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 $\Theta(\log N)$ bits.

The prime table $P[N]$ is represented as a bit array that is stored on disk and needs to be filled in. We say that $P[i] = 1$ means that i is prime and $P[i] = 0$ means that i is composite. We are interested in values of N, such that $P[N]$ is too large to fit in main memory. Thus, the prime table fills $\Theta(N/\log N)$ words.

RAM is divided into M *words*. Disk is modeled as arbitrarily large. Data is transferred between RAM and Disk in blocks of size B words ($\Theta(B \log N)$ bits). On a DAM (rather than a RAM), performance is measured in terms of block transfers, and computation is modeled as free [1, 29].

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 $\langle \mathcal{C}_{\mathrm{I/O}}, \mathcal{C}_{\mathrm{RAM}} \rangle$. For example, the array implementation of the sieve of Eratosthenes can be shown to run in $\langle \Theta(N), \Theta(N \log\log N) \rangle$.

If the problem size is large ($N = \Omega(M^2)$), other sieves perform poorly as well. In this case, segmenting the sieve of Eratosthenes does not lead to any improvements, and the sieve of Atkin requires $\langle O(N/\log\log N), O(N/\log\log N) \rangle$.

In contrast, a primality-checking sieve based on AKS runs in $\langle \Theta\left(N/(B \log N)\right), \Theta(N \log^c N) \rangle$, as long as $M = \Omega\left(\log^c N\right)$, a factor of $B \log N$ better in memory transfers but nearly $\log^c N$ worse in terms of operations.[4]

1.3 Our Contributions

We present data structures for four main sieves with the objective of optimizing both the number of I/Os and the operation count. Our algorithms work even when $M \ll N$. We consider the sieve of Eratosthenes [17], the linear sieve of Eratosthenes [15], the sieve of Atkin [6], and the sieve of Sorenson [28].

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].

We summarize our main results below.

1. *Sieve of Eratosthenes.* We show that the standard sieve of Eratosthenes can be implemented to run in $\langle \textsc{sort}(N), O(N \log_{M/B} N + N \log\log N \log\log M) \rangle$

[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.

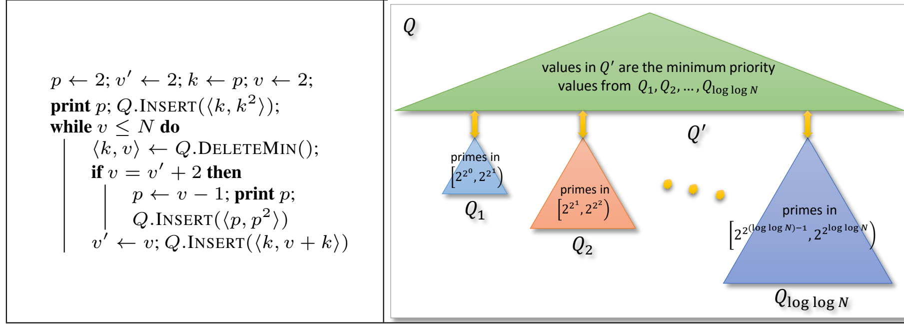

Fig. 1. (a) Original sieve of Eratosthenes using a priority queue, (b) A key-sensitive priority queue.

cost, under the assumption that $M = \Omega\left(B \log^{1+\varepsilon} \log N\right)$ for any constant $\varepsilon > 0$. We achieve these bounds using a new priority queue data structure in which the cost of any operation on an item with key k depends only on k instead of the total number of items in the data structure (as in standard priority queues).

2. *Linear sieve of Eratosthenes.* We implement the linear sieve of Eratosthenes using a buffer-tree-like data structure in $\langle \text{SORT}\,(N/\log\log N)\,,\ \ O((N/\log\log N)\log_{M/B}(N)) \rangle$ under the assumption that $\sqrt{M/B} = \Omega(\max\{\log_{M/B}(\frac{N}{B})\,,\ \log^2_{M/B}(\frac{N}{B})/\log\log N\})$.
3. *Sublinear sieve of Atkin.* We show that the sublinear sieve of Atkin can be implemented using I/O- and RAM- efficient priority queues [5] to run in $\left\langle \text{SORT}\,(N/\log\log N)\,,\ O\Big(\big(\frac{N}{\log\log N}\big)(\log_{M/B}(N) + \log\log M)\Big)\right\rangle$.
4. *Sieve of Sorenson.* We analyze the sieve of Sorenson in external memory and show that it runs in $\langle O(N/B),\ O(N\pi(p))\rangle$, where $\pi(p)$ denotes the smallest i such that the pseudosquare $L_{p_i} > N/(i\log^2 N)$, where p_i is the ith prime. We also show that given the availability of pseudosquare tables, this sieve can be adapted to sieve the interval $[a, b]$ in $\langle O(1 + (b - a + \pi(p)\log^2 b)/B),\ O((b-a)\pi(p) + \pi(p)\log^2 b)\rangle$.

2 Sieve of Eratosthenes

In this section we show that the sieve of Eratosthenes can be implemented to achieve I/O- and RAM-efficiency simultaneously using sublinear space. We start with a standard priority queue based implementation of the sieve (shown in Figure $1(a)$), and show that by using a new data structure which we call a *key-sensitive priority queue*, we can achieve sorting bound in I/Os without sacrificing RAM performance.

We start by analyzing the performance of the sieve using the recently proposed and only known RAM-efficient external-memory priority queue from [5]. We then observe that the smaller the prime the larger the number of priority queue operations performed on it, and so we can potentially improve the performance of the algorithm by reducing the cost of operations on smaller primes.

Sieve of Eratosthenes using a RAM-efficient external-memory priority queue. The sieve of Eratosthenes can be implemented efficiently using the priority queue of Arge and Thorup as a black box [5]. We describe this in detail in Appendix C.1. This achieves a performance of $\langle \text{SORT}\,(N\log\log N)\,,\ O(N\log\log N(\log_{M/B} N + \log\log M))\rangle$. However, we can shave off the $\log\log N$ factor using a new type of priority queue.

Sieve of Eratosthenes using a key-sensitive priority queue. In a ***key-sensitive priority queue*** the amortized access cost of an operation on an item with key k depends on k instead of the size of the data structure. This property is useful in improving the performance of the folklore priority-queue-based implementation of sieve of Eratosthenes (given in Figure $1(a)$). In that implementation, the number of priority queue operations performed on items with a given prime k as the key varies inversely with k. Thus, a reduction in the cost of operations on smaller primes has the potential of reducing the total cost of all operations. Indeed, we use such a priority queue to achieve sorting bound on I/Os in the sieve of Eratosthenes.

A key-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 $Q_1, Q_2, \ldots, Q_{\lceil \log\log N \rceil}$. Priority queues store $\langle key, value \rangle$ pairs where key is an integer in $[1, N]$ and $value$ is the priority of the item. For our sieving application, key will be a prime in $[1, \sqrt{N}]$ and $value$ will be a multiple of that prime. For any given key there will be at most one $\langle key, value \rangle$ pair in the entire data structure.

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 such that k is a prime in $[2^{2^i}, 2^{2^{i+1}})$. Hence, Q_i will contain fewer than $N_i = 2^{2^{i+1}}$ items. Then with a cache of size M, Q_i will support insert and delete-min operations in $\langle O((\log_{M/B} N_i)/B),\ O(\log_{M/B} N_i + \log\log M) \rangle$ amortized cost [5]. But note that in Q_i, each key satisfies $\log k = \Theta(\log N_i)$. Thus the cost reduces to $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k + \log\log M) \rangle$ for an item with key k. 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 will include only the item with the smallest value from each Q_i. So the size of Q' will be $\Theta(\log\log N)$. We use the dynamic integer set data structure from [21] to implement Q' so that insert, delete and delete-min operations on Q' can be supported in $O(1)$ time each using only $O(\log n)$ space (in words). We also maintain an array $A[1 : \lceil \log\log N \rceil]$ such that $A[i]$ stores Q_i's contributed item to Q' so that we can access it constant time.

The priority queue Q only needs to support insert and delete-min operations. To perform an delete-min we extract the smallest item from Q', check its key to find the Q_i it came from, extract the smallest item from that Q_i and insert it into Q'. To insert an item we first check its key to determine its 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 bounds of the operations on Q.

Lemma 1. *Let each Q_i be a RAM-efficient external-memory PQ as described in [5], and let Q' be a priority queue based on the dynamic integer set data structure given in [21]. Then in the resulting data structure, the amortized cost of insert on an item with key k is $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k) \rangle$ and delete-min is $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k + \log\log M) \rangle$, assuming $M > \log N + B(\log\log N)^{1+\varepsilon}$ for any constant $\varepsilon > 0$.*

We use this key-sensitive priority queue to efficiently implement the sieve of Eratosthenes. The following theorem follows from the observation that a prime p will be

involved in $\Theta(N/p)$ priority queue operations in Q, and because it is known that there are approximately $\sqrt{N}/(\ln(\sqrt{N})-1)$ prime numbers in $[1,\sqrt{N}]$ [7], and the i-th such prime number is approximately $i \ln i$ [16].

Theorem 1. *Using the priority queue from Lemma 1, the sieve of Eratosthenes costs* $\langle \text{SORT}(N),\ O(N(\log_{M/B} N + \log\log M \log\log N))\rangle$ *and uses* $O(\sqrt{N})$ *space, provided* $M > \log N + B(\log\log N)^{1+\varepsilon}$ *for some constant* $\varepsilon > 0$.

3 Linear Sieve of Eratosthenes

There are several variants of the sieve of Eratosthenes [9, 14, 15, 19] that perform $O(N)$ operations by only marking each composite exactly once. See [25] for a survey.

Even though each composite is marked exactly once, resulting in $O(N)$ operations, many of these algorithms have poor data locality. The marking requires large jumps around the array, leading to $O(N)$ I/Os—very poor locality.

In this section, we improve the locality of such linear sieves, while also taking advantage of the bit-complexity of words to improve the performance further. We use a buffer-tree-like data structure (adapted from Arge [3]) to improve the locality, resulting in a cost of $\langle \text{SORT}(N/\log\log N),\ O((N \log_{M/B} N/\log\log N)\rangle$.

We focus on one of the linear variants, the linear sieve algorithm by Gries and Misra [15], henceforth referred to as *the linear sieve of Eratosthenes*.[5] The linear sieve of Eratosthenes is based on the following fundamental property of composite numbers.

Theorem 2 ([15]). *Each composite* C *can be represented uniquely as* $C = p^r q$ *where* p *is the smallest prime factor of* C*, and* p *does not divide* q *(unless* $p = q$*).*

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 $O(N)$.

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

Algorithm 1: Linear SoE

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

While the RAM complexity is an improvement by a factor $\log\log N$ over the classic sieve of Eratosthenes, the algorithm (thematically) performs poorly in the DAM model. The overall complexity of this algorithm is $\langle O(N),\ O(N)\rangle$. In the rest of the section we improve the I/O complexity while maintaining good RAM performance.

Using a buffer-tree-like structure. As a first step, we introduce the classical buffer tree of Arge [3]; we will then modify the structure to improve the bounds of the linear sieve. We give a high-level overview of the data structure here; for details see Appendix A.

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

[5] Note that the other linear-sieve variants, such as [9, 14, 19] share the same underlying data-structural operations as the sieve of Gries and Misra.

$\lceil \log_{M/B} N/M \rceil = O(\log_{M/B} N/B)$. Newly-inserted elements are placed into the root buffer. If the root buffer is full of M elements, all of its elements are flushed: sorted, and then placed in their respective children; this takes $O(M/B)$ I/Os and $O(M \log M)$ RAM complexity. This process is repeated recursively for any newly-full buffers. 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 $\langle O(N/B \log_{M/B} N/B),\ O(N \log N)\rangle$.

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 $\langle O(M/B \log_{M/B} N/B),\ O(M \log_{M/B} N/B)\rangle$ for a single inverse successor query.

To achieve better bounds, we need to improve the inverse successor time to match the insert time. At the same time, we improve the computation time considerably; we only do $O(B)$ computations per I/O, the best possible for a given I/O bound.

As a first step, we perform a wheel sieve using the primes up to $\sqrt{\log N}$. By an analogue of Merten's Therem, 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, our buffer tree has branching factor $\sqrt{M/B}$ rather than M/B, doubling the height. We 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. Since we use the word RAM model (Section 1.2), we can read $\Theta(B \log N)$ of these bits in a single block transfer.

Storing the elements in a bit array 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. During an inverse successor scan, we maintain the path in memory; thus, we can flush all elements along the path without any I/O cost. This guarantees that we can get the correct inverse successor by scanning the array, maintaining the path as we scan.

Finally, since our array is static, we can improve the computation required during a flush. Specifically, since the leaves divide the array evenly, we can calculate the child being flushed to using modular arithmetic (see Appendix A for details).

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 costs $O(1/B)$ amortized I/Os and $O(1)$ computation. 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 3. *The linear sieve of Eratosthenes implemented using buffer trees, 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* $\langle \text{SORT}\,(N/\log\log N)\,,\ O((N \log_{M/B} N/B)/\log\log N)\rangle$.

4 Sieve of Atkin

The sieve of Atkin [6, 13] 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, including the key-sensitive priority queue from Section 2, 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 must satisfy one of the three congruences. For an excellent introduction to the underlying number theoretic concepts, see [11].

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 solutions to $x^2 + 4y^2 = k$ (resp. $3x^2 + y^2 = k$, $3x^2 - y^2 = k$) is odd.*

For each quadratic form $f(x, y)$, the number of solutions can be computed by brute force, iterating over the set $L = \{(x, y) \mid 0 < f(x, y) \leq N\}$. This requires $O(N)$ memory; however, by "tracing" the level curves of f, this can be reduced to $O(\sqrt{N})$ (see Appendix B). Then, the number of solutions that occur an even number of times are removed. Then 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 Merten's 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 $M = O(\sqrt{N})$. 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 easily adapted to omit this. See appendix B.3 for a more detailed algorithm description.

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

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

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 and for every prime we find, we maintain a record of the multiples of its square. We will track these as pairs $\langle p, v\rangle$ in

another I/O+RAM efficient priority queue Q'. With each value v we pull from Q, we repeatedly extract the min value $\langle p, w\rangle$ from Q' and insert $\langle p, w + p^2\rangle$ until either v is found in which case it is not square-free and thus not a prime, or exceeded, in which case it is prime.

For each prime p less than $\sqrt{N}$ that was not sieved by the wheel, this part of the algorithm will perform $\langle O(N(\log_{M/B} N)/Bp^2),\ O(N(\log_{M/B} N + \log\log M)/p^2)\rangle$ operations. Integrating over p, the total number of operations in this phase of the algorithm is less than $\langle O\left(\text{SORT}\left(N\right)/(B\log N)\right),\ O\left(\left(\text{SORT}\left(N\right) + \log\log M\right)/\log N\right)\rangle$.

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

See Appendix B.1 for a description of how to reduce the space usage to $O(\sqrt{N})$.

5 Sieve of Sorenson

The sieve of Sorenson [28] uses a hybrid approach. It first uses a wheel sieve to remove multiples of small primes. Then, it eliminates nonprimes using a test based on 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 $q \le p$. The sieve of Sorenson is based around the following lemma—its steps satisfy each requirement of the lemma explicitly. Following the theorem, we set p so that L_p is the smallest pseudosquare satisfying $L_p > N/(\pi(p)\log^2 N)$, and $s = \lfloor N/L_p\rfloor + 1$.

Theorem 6. [28] *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 pth pseudosquare for some prime p, and (iii) $p_i^{(x-1)/2} \equiv \pm 1 \pmod x$ for all primes $p_i \le p$, and (iv) $2^{(x-1)/2} \equiv -1 \pmod x$ when $x \equiv 5 \pmod x$ and $p_i^{(x-1)/2} \equiv -1 \pmod x$ for some prime $p_i \le p$ when $x \equiv 1 \pmod 8$, then x is a prime or a prime power.*

To begin, the algorithm must calculate L_p. We refer to the original paper for a method that performs this calculation in $o(N)$, but which further points out that the first 73 pseudosquares (available online at https://oeis.org/A002189/b002189.txt) are sufficient for any $N < 2.9 \times 10^{24}$. Next, the algorithm calculates the first s primes.

The algorithm divides all N integers into segments of size $\Delta = \pi(p)\log N$. For each such segment, it goes through the following three phases. We assume that $M >> \pi(p)$.

In the first phase, the algorithm performs a (linear) wheel sieve to eliminate multiples of the first s primes. All remaining numbers satisfy the first requirement of Theorem 6.

In the second phase, the algorithm considers each remaining integer k in turn. It first performs a base-2 pseudoprime test, determining if $2^{(k-1)/2} \equiv -1 \pmod k$. If k does satisfy this, then for each $p_i \le p$, it determines if $p_i^{(k-1)/2} \equiv \pm 1 \pmod k$, with $p_i^{(k-1)/2} \equiv -1 \pmod k$ for some $p_i \le p$, as well as if $k \equiv 1 \pmod 8$. Note that this is testing the remaining requirements of Theorem 6.

To analyze the RAM complexity, first note that only $O(N/\log N)$ numbers up to N pass the base-2 pseudoprime test (mentioned in [22, 28], among other places).

Furthermore, in a single segment, only $O(\Delta/\log s)$ elements remain after the wheel sieve. Performing the base 2 pseudoprime test takes $O(\log N)$ time, for a total time of $O(N\log N/\log s) = o(N\log N)$. Performing the remaining tests, if required, takes $\pi(p)$ exponentiations, costing $O(\log N)$ operations each, leading to a total cost of $O(N\pi(p))$ over all segments.

In the third phase, the algorithm must remove all prime powers. If $N \leq 6.4 \times 10^{37}$, only primes remain and this phase is unnecessary [28, 30]. Otherwise the algorithm explicitly removes all perfect powers as follows. First, the algorithm constructs by brute force a list of all the perfect powers less than N by repeatedly exponentiating every element of the set $\{2, \dots, \lfloor\sqrt{N}\rfloor\}$ until it passes N. This list has $O(\sqrt{N}\log N)$ elements, so these can be sorted and removed from the list of prime candidates in $\langle O(N/B), O(N)\rangle$. Therefore the complexity of the algorithm is dominated by the second phase, leading to the following theorem.

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

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 $O(N^{1.132})$. On the other hand, the Extended Riemann Hypothesis implies $p < 2\log^2 N$, and Sorenson conjectures that $p \sim \frac{1}{\log 2}\log N\log\log N$ [28]; under these conjectures the RAM complexity is $O(N\log^2 N/\log\log N)$ and $O(N\log N)$ respectively.

Sieving an interval. Similar analysis shows that we can efficiently sieve an interval with Sorenson as well. See Appendix C.2 for details.

Acknowledgments

We thank Oleksii Starov for suggesting this problem to us.

References

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, pages 781–793, 2004.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc./ of the 34th Annual Symposium on Theory of Computing*, pages 268–276, 2002.

[5] L. Arge and M. Thorup. Ram-efficient external memory sorting. In *Algorithms and Computation*, volume 8283, pages 491–501. 2013.

[6] A. Atkin and D. Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246):1023–1030, 2004.

[7] J. Barkley Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math*, 6:64–94, 1962.

[8] C. Bays and R. H. Hudson. The segmented sieve of Eratosthenes and primes in arithmetic progressions to 1012. *BIT Numerical Mathematics*, 17(2):121–127, 1977.

[9] S. Bengelloun. An incremental primal sieve. *Acta informatica*, 23(2):119–125, 1986.

[10] R. P. Brent. The first occurrence of large gaps between successive primes. *Mathematics of Computation*, 27(124):959–963, 1973.

[11] D. A. Cox. *Primes of the form $x^2 + ny^2$: Fermat, Class Field Theory, and Complex Multiplication*. Wiley, 1989.

[12] B. Dunten, J. Jones, and J. Sorenson. A space-efficient fast prime number sieve. *IPL*, 59(2):79–84, 1996.

[13] M. Farach-Colton and M. Tsai. On the complexity of computing prime tables. In *Algorithms and Computation - 26th International Symposium, ISAAC'15*, 2015.

[14] R. Gale and V. Pratt. CGOL–an algebraic notation for MACLISP users, 1977.

[15] D. Gries and J. Misra. A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12):999–1003, 1978.

[16] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[17] S. Horsley. KOΣKINON EPATO$\Sigma\Theta$ENOYΣ. or, The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers, by the Rev. Samuel Horsley, FRS. *Philosophical Transactions*, pages 327–347, 1772.

[18] H. W. Lenstra Jr and C. Pomerance. Primality testing with gaussian periods. *Lecture Notes in Computer Science*, pages 1–1, 2002.

[19] H. G. Mairson. Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20(9):664–669, 1977.

[20] F. News. World's largest prime number discovered – all 17 million digits. `https://web.archive.org/web/20130205223234/http://www.foxnews.com/science/2013/02/05/worlds-largest-prime-number-discovered/`, February 2013.

[21] M. Patrascu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *FOCS*, pages 166–175, 2014.

[22] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

[23] P. Pritchard. A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1):18–23, 1981.

[24] P. Pritchard. Linear prime-number sieves: A family tree. *Science of computer programming*, 9(1):17–35, 1987.

[25] P. Pritchard. Linear prime-number sieves: A family tree. *Science of computer programming*, 9(1):17–35, 1987.

[26] R. C. Singleton. Algorithm 357: An efficient prime number generator. In *Communications of the ACM*, pages 563–564, 1969.

[27] J. Sorenson. An introduction to prime number sieves. Technical Report 909, University of Wisconsin-Madison, Computer Sciences Department, 1990.

[28] J. P. Sorenson. The pseudosquares prime sieve. In *Algorithmic number theory*, pages 193–207. 2006.

[29] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[30] H. C. Williams. Edouard lucas and primality testing. *Canadian Mathematics Society Series of Monographs and Advanced Texts*, (22), 1998.

A Linear Eratosthenes's sieve with buffer trees

In this section, we provide further details for the algorithm given in Section 3 and prove its correctness.

We first recall the version of the linear sieve of Gries and Misra [15].

Note that we begin by pre-sieving the interval by the $\sqrt{\log N}$ smallest primes. This operation speeds up the execution without violating its correctness, as the resulting candidate primes are exactly those encountered by the algorithm after the loop where $p = p_{1+\sqrt{\log N}}$. This has a cost of $\langle O(N/B \log N),\ O(N/\log N)\rangle$, using the algorithm in Appendix C and leaves $\bar{N} = N/\log\log N$ potential primes. We will use $\bar{N}$ in the following to refer to the number of elements inserted.

```
Data: S = {2, 3, ··· , N}
Result: S = {p | p ∈ ℙ, p ≤ N}
// p and q are global variables, accessible in any
   function
// 𝒞 represents the set of numbers known as
   composites. The operations Insert and
   InverseSuccessor are implicitly on 𝒞
1  𝒞 ← integers less than N multiple of any of the first √(log N) primes;
2  𝒯 ← bitarray where 𝒯[i] = 1 iff i ∈ 𝒞;
3  p ← p_{1+√(log N)};
4  while p ≤ √N do
5  |  q ← p;
6  |  while q ≤ N/p do
7  |  |  for r = 1, 2, ..., log_p N/q do
8  |  |  |  Insert(p^r q);
9  |  |  q ← InverseSuccessor(q);
10 |  p ← InverseSuccessor(p);
11 return GetSet();                       // this is [2; N] \ 𝒞
```

Algorithm 2: Linear Sieve with Buffer Tree

We now present how we implement these subroutines to achieve an efficient algorithm in both I/O and RAM complexity, then prove its correctness and complexity.

A.1 Implementation

We expose in this part how the subroutines `Insert` and `InverseSuccessor` are actually implemented in our algorithm. First, we give a global description of the data structure used. Then, after setting some preliminary definitions and notations, we present the actual implementation of the subroutines.

Description of the data structure. We use a modified buffer tree which can efficiently handle the two necessary operations to maintain the set $\mathcal{C}$, `Insert` and `InverseSuccessor`. The original structure has been introduced by Arge [3], but our implementation is significantly different to achieve a lower RAM complexity.

The buffer tree is a complete tree with branching factor $\sqrt{M/B}$ and N/M leaves. Its depth is then $d = 2\left\lceil \log_{M/B} \frac{N}{M} \right\rceil$. We will assume for simplicity that the tree is complete even at the leaf level.

Each node has an associated buffer of size M. This buffer consists of $\sqrt{M/B}$ pages of size $\sqrt{MB}$, one for each child, which are internally unsorted. These pages contain the elements in that buffer that are intended for the corresponding child; see Figure 2.

Each leaf corresponds to M consecutive elements between 1 and N. Similar to the internal nodes, the buffer of a leaf is separated in $\sqrt{M/B}$ pages, each associated with exactly $\sqrt{MB}$ elements.

In addition to the buffer tree, the data structure used contains a boolean array $\mathcal{T}$, indexed from 1 to N, where, at the end of the algorithm, $\mathcal{T}[i] = 1$ if and only if i is composite. Intervals of $\mathcal{T}$ corresponding to a leaf page are considered *linked* to this page: when the entire page is brought into memory, this interval is too. This array is saved as a bit-array, which means that one machine word contains at least $\log N$ bits, and operations on a machine word can be done in constant time.

Each page P of an internal (non-leaf) node is partitioned into $\sqrt{M/B} + 1$ unsorted lists. The first one, called P^*, is the list where the new elements are appended. Each other list corresponds to a page Q of the child linked to P, and is denoted by P_Q. Elements moved to a page Q of the next level are either moved directly from P^*, or first moved to the list P_Q then later to Q. See Figure 3 for an illustration; an element can follow blue or red arrows to go to the next level. For consistency, the unique list of a leaf page L will be denoted L^*.

At each level, the numbers present in a node are smaller than the numbers present in the next node. Therefore, inside each node, the numbers present in a page are also smaller than the numbers present in the next page. Each page of the level k, counting the root as the level 0, can then contain $\sqrt{MB}$ numbers among a fixed interval of length $N/\left(MB\right)^{(k+1)/2}$. Therefore, for example, the ith leave, which is at level d (starting the count at 0), consists in M slots to store a subset of $[iM+1\ ;\ (i+1)M]$.

Note that no element is ever moved to an upper level nor removed from the tree, except to be inserted in $\mathcal{T}$. Elements can only be moved deeper or to $\mathcal{T}$. In addition, no duplicates are possible.

See Figure 2 for an illustration of the data structure.

Preliminaries We expose here some notations and definitions used throughout the explanation of the algorithm and the proof.

The nodes are indexed by the letter N, the pages by the letter P, and the leaf pages by the letter L.

The least common ancestor page of two leaf pages L and L' will be noted $\mathbf{LCA_P}(L,\ L')$ and its level $\mathbf{LCA}(L,\ L')$.

Above means closer to the root level and *deeper* means closer to the leaf level.

`LCA-ABOVE`(L) is the property: For all leaves L', no element of L' is above $\mathbf{LCA}(L,\ L')$. In addition, no element of L is in L^*: they are all in $\mathcal{T}$.

`LCA-ABOVE`(L, k) is the property: For all leaf L', no element of L' is above $\min(\mathbf{LCA}(L,\ L'),\ k)$. Note that `LCA-ABOVE`$(L, d)$ allows elements of L to be in L^*, and, by convention, `LCA-ABOVE`$(d+1)$ is equivalent to `LCA-ABOVE`(L).

Note that if $k' \geq k$, `LCA-ABOVE`(L, k') implies `LCA-ABOVE`(L, k).

Access functions: Due to the static structure of the tree, the following operations can be implemented with a complexity of $\langle 0,\ O(1)\rangle$. When a page P is passed in argument or

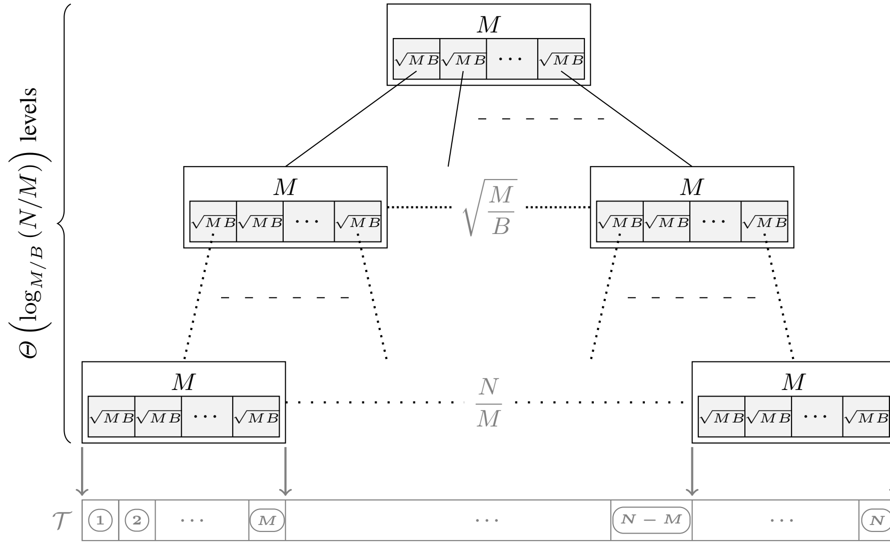

Fig. 2. Illustration of the buffer tree and $\mathcal{T}$.

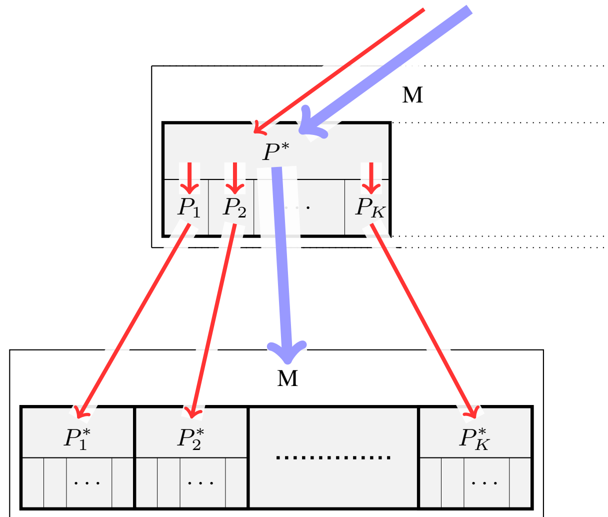

Fig. 3. Illustration of a page of the buffer tree: the list P^* can be flushed to any page of the corresponding child node in `Flush`. The other lists can only be flushed to one page in `PartialFlush` or in `Flush`. An insertion is always executed on the list P^* of each page. We have $K = \sqrt{M/B}$.

returned by a function, only an identifier is implied, and not all the numbers contained, hence the null I/O cost.

- `GetPage` (x, k) : returns the page associated to the number x at level $k \leq d$
- `GetPage` (L, k) : returns the page associated to numbers of leaf page L at level $k \leq d$

For the sake of simplicity, for any number x, we will note $L_x = \texttt{GetPage}(x, d)$. For instance, the leaf pages $L_p = \texttt{GetPage}(p, d)$ and $L_q = \texttt{GetPage}(q, d)$ can be computed in any function.

We define the set $\mathcal{P}_{p,q}$ by the set of pages associated to p or q plus the pages of the root. This set will be assumed to be in memory in the design of the algorithms. This assumption in proved later, together with the complexity proof. Note that $\mathcal{P}_{p,q}$ is modified during the execution of the algorithm. More formally, the definition of $\mathcal{P}_{p,q}$ is:

$$\begin{aligned}\mathcal{P}_{p,q} = &\{P \mid \exists k \leq d,\ P = \texttt{GetPage}(p, k) \text{ or } P = \texttt{GetPage}(q, k)\} \\ &\cup \left\{P \mid P \text{ is of level } 0\right\}\end{aligned}$$

Note that saying that $\mathcal{P}_{p,q}$ is in memory includes the relevant slots of $\mathcal{T}$.

Subroutines We expose here a detailed explanation on how both subroutines are implemented, along with the associated pseudo-code.

Management of $\mathcal{T}$: The implementations of basic operations performed on $\mathcal{T}$ are detailed in Algorithm 6. An insertion in $\mathcal{T}$ simply modifies the corresponding bit. The function `NextIn`$\mathcal{T}$ (x) computes the next candidate to be prime. All integers skipped are confirmed composites. This function uses the bit-array structure of $\mathcal{T}$ to gain a $\log N$ speedup, as we will show later.

Insertions: The algorithm `Insert` is presented in Algorithm 2.

Basically, an element x is inserted at a given level k by computing the appropriate page P and appending it to the list P^*. If this page is full, i.e., it already contains $\sqrt{MB}$ elements, these elements are moved to the next level via Procedure `Flush`. In addition, if x is associated to a page of $\mathcal{P}_{p,q}$ in the next level, it is inserted to the deepest page of $\mathcal{P}_{p,q}$ possible.

These moves are done directly from the list P^*, and the appropriate page of the next level is computed for each element. This process follows the blue arrows on Figure 3.

A call to `Insert` in Algorithm 2 triggers a call to insert the element at level 0. It can be inserted deeper according to $\mathcal{P}_{p,q}$ as illustrated in Figure 4, and trigger some flushes.

At the end of the algorithm, the call to `GetSet` flushes all the tree into the array $\mathcal{T}$, then returns $\mathcal{T}$.

Inverse Successor: The algorithm `InverseSuccessor` is presented in Algorithm 13.

The objective of `InverseSuccessor` is to compute the next element that is not in $\mathcal{C}$, which means that is not in the tree or in $\mathcal{T}$. A naive algorithm would be to check in each page if the element is present or not, but this achieves a very high complexity. The strategy of `InverseSuccessor` is to ensure that the next elements cannot be in

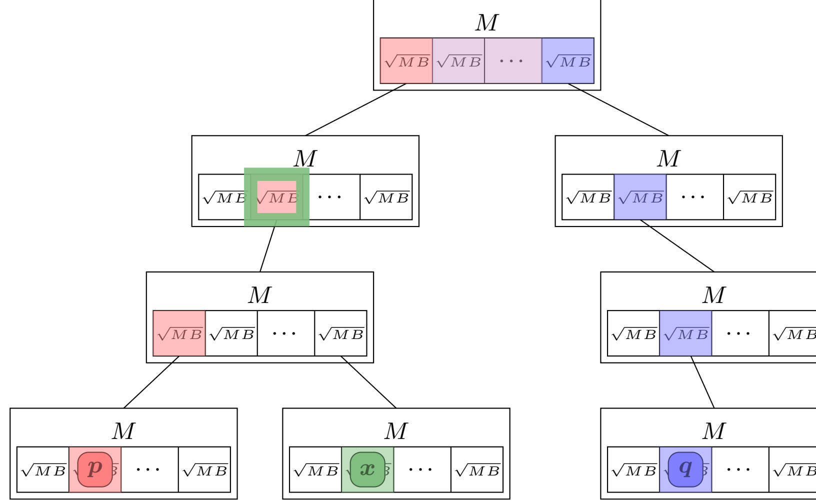

Fig. 4. Illustration of the insertion of x. The leaves in which p, q and x belong are drawn. The green rectangle is the page where x will be inserted. The colored pages are pages of $\mathcal{P}_{p,q}$. $\mathcal{T}$ is not represented.

Procedure `InsertIn`$\mathcal{T}$ (x)

Input: A multiple x with $\mathcal{T}[x] = 0$
Result: Add x in $\mathcal{T}$
$\mathcal{T}[x] \leftarrow 1$; `// This only modifies one bit`

Procedure `NextIn`$\mathcal{T}$ (x)

Input: An integer x with $\mathcal{T}[x] = 0$
Result: An integer $z > x$ such that $\forall t \in (x, z),\ \mathcal{T}[t] = 1$
`// We even have` $\mathcal{T}[t] = 0$ `in this version but it is not required`

1. $w' \leftarrow$ machine word containing $T[x]$;
2. $w \leftarrow w'$ with all bits not after $T[x]$ set to 1;
3. **while** *w has no bit equal to 0* **do**
4. $\quad w \leftarrow$ machine word representing the elements of T after w;
5. $z \leftarrow$ index of $\mathcal{T}$ corresponding to the first bit of w equal to 0;
6. **return** z;

Algorithm 3: Operations on $\mathcal{T}$

Algorithm `Insert(x)`

Input: A number $x \notin \mathcal{C}$
Invariant: `LCA-ABOVE`(L_p) and `LCA-ABOVE`(L_q)
Result: Insertion of x in the buffer tree or in $\mathcal{T}$, so in $\mathcal{C}$

```
Insertion(x, 0);
```

Procedure `Insertion(x, k)`

Input: A number x and a level $k \leq d$
Data: x is not in the buffer tree nor in $\mathcal{T}$
Result: Insertion of x at level $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q),\ k)$, deeper, or in $\mathcal{T}$

```
1  if L_x equals L_p or L_q then
2      InsertInT(x) ;
3      return;
4  P ← GetPage(x, k);
   // if the page of the next level is in P_{p,q}, insert deeper
5  P_p ← GetPage(p, k+1); P_q ← GetPage(q, k+1);
6  if k < d and GetPage(x, k+1) is equal to P_p or to P_q then
7      Insertion(x, k+1) ;                                   // k < d
8  else
9      if k < d and |P| = √(MB) then
10         Flush(P, k);
11     append x to P* ;
```

Procedure `Flush(P, k)`

Input: A full page P of level $k < d$
Result: P is empty

```
   // Note that all the elements will be inserted in the
      same node
1  foreach x ∈ P do
2      remove x from P;
3      Insertion(x, k+1);
```

Procedure `GetSet()`

Result: A bit array caracterising the set $[2; N] \setminus \mathcal{C}$ by the value 0

```
1  Call Flush on every page for any pre-ordering (children after parents);
2  return T;
```

Algorithm 4: Insertion algorithm and sub-functions

a node: they are in $\mathcal{T}$. This is done by a call to the procedure `PartialFlush`. Then, it is efficient to compute the next element that is not in $\mathcal{C}$ by scanning $\mathcal{T}$.

It is still inefficient if `PartialFlush` has to scan each level of the tree to move the appropriate elements to $\mathcal{T}$, so this function uses the fact that the inserts are done at the deepest page possible in $\mathcal{P}_{p,q}$. This way, as `InverseSuccessor` is only called on the previous value of p or q, `PartialFlush` does not need to scan a page high in $\mathcal{P}_{p,q}$: no relevant element has been inserted here. For instance, on Figure 4, when `PartialFlush` will be called on L_x, it will not check the root node.

This process avoids a high I/O complexity, but still needs a high RAM complexity: for each page scanned, all the elements are checked to see if they can be inserted deeper in $\mathcal{P}_{p,q}$. This means that an element at a page can be scanned $\sqrt{\frac{M}{B}}$ times, once per child page, where we want a constant cost. Thus, `PartialFlush` actually scans only the list P^* of each page, and move the elements to the appropriate list P_Q. Then, it moves all elements from the relevant list P_Q to the next level. This process follows the red arrows in Figure 3.

A.2 Analysis

We first begin by proving the correctness of the implementation, then its complexity. The optimization of the RAM complexity will be discussed after.

Correctness of the algorithm. We need to prove that the algorithms `Insert` and `InverseSuccessor` are correct. *Correct* means that if the input, the data, and the invariant requirements are verified when a function is called, then the output and invariant requirements are verified when the function terminates, and the function does not violate Lemma 2. In addition, no insertion of a new element in the tree not mentioned in the result requirement is performed.

Lemma 2. *An element is never moved to an upper level. No duplicates are possible. If an element is removed from the tree, it is added to $\mathcal{T}$.*

Proof. This lemma will be proved by Theorem 8, which states that the algorithms are correct.

Lemma 3. *For any k, x, P, the procedures* `Insertion(`*x*, *k*`)` *and* `Flush(`*P*, *k*`)` *are correct.*

Proof. We prove this result by induction on $d - k$.

If $k = d$, the call to `Insertion(`*x*, *k*`)` adds x either to the array $\mathcal{T}$ or to the appropriate page leaf so is correct. `Flush` cannot be called on such input, so the property is verified.

Suppose the property true for $k - 1$, and we now prove it for k.

Consider the procedure `Flush`. For each element of the page P, it is removed from P, then, by induction, inserted to a deeper level or in $\mathcal{T}$. So `Flush` is correct as it does not violate Lemma 2 and empty P.

Consider the procedure `Insertion`.

If the test Line 2 occurs, x is inserted in $\mathcal{T}$ so the call is correct.

Algorithm `InverseSuccessor`(x)

Input: A number $x \notin \mathcal{C}$
Data: LCA-ABOVE(L_x)
Output: The smallest number y greater than x and not present in $\mathcal{C}$
Result: LCA-ABOVE(L_y)

```
1   y ← x;
    // scan T to find y
2   repeat
3       tmp ← y;
4       y ← NextInT(y);
5       if L_y ≠ L_tmp then
            // flush the next elements from the buffer tree to
               T
6           PartialFlush(d, L_y, L_tmp);
7   until T[y] = 0;
8   return y;
```

Procedure `PartialFlush`(k, L, L_{tmp})

Input: A level $k \leq d$ and two leaves L and L_{tmp}
Data: LCA-ABOVE(L_{tmp})
Result: LCA-ABOVE($L,\ k+1$)

```
1   P ← GetPage(L, k);
    // Flush partially the ancestor pages starting from
       LCA_P(L, L_tmp)
2   if k > 0 and GetPage(L_tmp, k) ≠ P then
3       PartialFlush(k − 1, L, L_tmp)
4   if k = d then                      // in this case, we have P = L
        // move the leaf page to T
5       foreach z ∈ L* do
6           remove z from L*;
7           InsertInT(z);
8   else
        // move each unlabeled element to its corresponding
           list
9       foreach z ∈ P* do
10          move z to P_GetPage(z, k + 1) ;
        // Move deeper the elements that are in the same page
           as L
11      foreach z ∈ P_GetPage(L, k + 1) do
12          Remove z from P;
13          Insertion(z, k + 1);
```

Algorithm 5: Insertion algorithm and sub-functions

Now suppose that $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q)) \quad > \quad k$. Then, the recursive call Line 7 occurs. By induction, x is inserted at a level larger than $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q)\ , k)$, so the current call is correct.

Otherwise, `Flush` is called line 10, then x is inserted at level k. So the function is correct as it does not violate Lemma 2 and respects the result requirements.

Lemma 4. *For any k, L, L_x, the procedure* `PartialFlush(`*k, L, L_{tmp}*`)` *is correct.*

Proof. We prove this result by induction on k.

If $k = 0$, for any L', if $\mathbf{LCA}(L,\ L') > 0$ then elements of L' in the root are moved deeper or to $\mathcal{T}$ in Line 13, as the `Insertion` procedure is correct by Lemma 3, so the call of `PartialFlush` is correct.

Suppose the property true for $k - 1$, and we now prove it for $k < d$.

First, suppose that $\mathbf{LCA}(L,\ L_{tmp}) < k$. Then, the recursive call Line 3 occurs, and by induction, we have `LCA-ABOVE`$(L,\ k)$. For any L' such that $\mathbf{LCA}(L,\ L') > k$, if elements of L' are in P, they are moved deeper or to $\mathcal{T}$ at Line 13, as the `Insertion` procedure is correct. If they were in P^*, they are moved to the appropriate page on Line 10. So we have `LCA-ABOVE`$(L,\ k+1)$.

Therefore, as the procedure respects Lemma 2, it is correct.

Then, suppose that $\mathbf{LCA}(L,\ L_{tmp}) \geq k$. We have `LCA-ABOVE`$(L_{tmp})$, let's prove that we then have `LCA-ABOVE`$(L,\ \mathbf{LCA}(L,\ L_{tmp}))$. We will illustrate the cases by Figure 5. Let L' be a leaf page.

If $\mathbf{LCA}(L,\ L_{tmp}) \geq \mathbf{LCA}(L,\ L')$, then $\mathbf{LCA}(L,\ L') \leq \mathbf{LCA}(L_{tmp},\ L')$. As we have `LCA-ABOVE`$(L_{tmp})$, no element of L' is above $\mathbf{LCA}(L_{tmp},\ L')$ so no element of L' is above $\mathbf{LCA}(L,\ L')$. This corresponds to the case where L' is at the position of Node B, C or D in Figure 5. Note that only the position B implies $\mathbf{LCA}(L,\ L') < \mathbf{LCA}(L_{tmp},\ L')$.

Otherwise, we have $\mathbf{LCA}(L,\ L_{tmp}) \quad < \quad \mathbf{LCA}(L,\ L')$. Then, we have $\mathbf{LCA}(L_{tmp},\ L') = \mathbf{LCA}(L,\ L_{tmp})$ and by definition of `LCA-ABOVE`(L_{tmp}), no element of L' is above $\mathbf{LCA}(L_{tmp},\ L')$ so $\mathbf{LCA}(L,\ L_{tmp})$. This corresponds to the page A of Figure 5.

In both cases, the property `LCA-ABOVE`$(L,\ \mathbf{LCA}(L,\ L_{tmp}))$ is then respected.

Therefore, we have `LCA-ABOVE`$(L,\ \mathbf{LCA}(L,\ L_{tmp}))$ so we have the weaker property `LCA-ABOVE`$(L,\ k)$. Then, as previously, the moves of Line 13 ensure `LCA-ABOVE`$(L,\ k+1)$ and the correctness of the procedure.

Then, if both cases, the procedure is correct.

Now, we prove the result for $k = d$. By induction, we have `LCA-ABOVE`$(L,\ k)$ after Line 3, if $L = L_{tmp}$ or not. In Line 6, all the elements of L_y are moved to $\mathcal{T}$, so we get `LCA-ABOVE`$(L,\ k+1)$.

Therefore, the procedure is correct for any k.

Theorem 8. *The algorithms* `Insert` *and* `InverseSuccessor` *are correct.*

Proof. First, we study the `Insert` algorithm.

As the procedure `Insertion` is correct and $x \notin \mathcal{C}$, the call to `Insertion` has a valid input and x is inserted in the tree at a level not smaller than $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q))$. If L_x equals L_p or L_q, x is inserted directly in

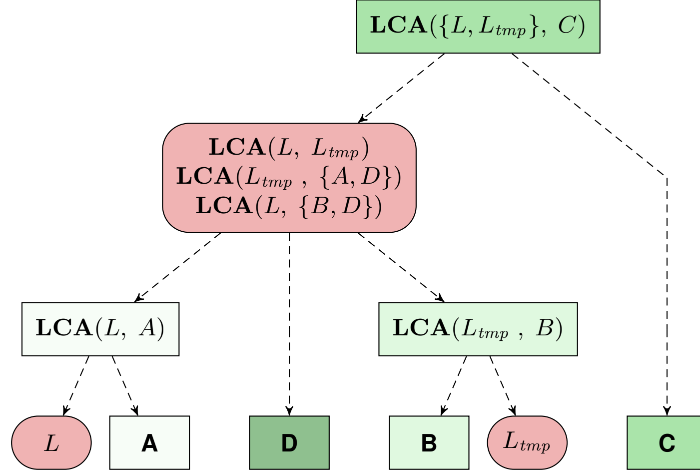

Fig. 5. Abstract tree representing the position of the Least Common Ancestors between two leaf pages L, L_{tmp} and four leaf pages $A,\ B,\ C,\ D$ representing all the possible cases.

$\mathcal{T}$. So the invariant is respected as no element has been moved to a smaller level and x does not violate it.

Now, we study the `InverseSuccessor` algorithm. After each call to `PartialFlush` Line 5, we have `LCA-ABOVE`$(L_y,\ k+1)$, which is `LCA-ABOVE`(L_y). In particular, all elements of L_y are in $\mathcal{T}$. This property is ensured at the beginning by the requirement `LCA-ABOVE`(L_x). The algorithms maintain this property each time y is in a new leaf page, so when the tests Line 7 occur, they are equivalent to testing $y \notin \mathcal{C}$. Therefore, the output of `InverseSuccessor` is correct. Furthermore, there exists a list of leaf pages $\mathcal{L} = \{l_1 \ldots l_t\}$ (which is the list of successive L_{tmp}) such that $l_1 = L_x$, $l_t = l_y$ and for each i, a call to `PartialFlush(`d, l_i, l_{i+1}`)` has been triggered. When such a call is triggered, if we had `LCA-ABOVE`(l_i), we get `LCA-ABOVE`(l_{i+1}).

So, as we have `LCA-ABOVE`(L_x) at the beginning of the `InverseSuccessor` call, we have `LCA-ABOVE`(L_y) at the end. So the procedure `InverseSuccessor` is correct.

Complexity of the algorithm. Now, we analyze the total complexity of the algorithm.

We assume Assumption 9 concerning the size of B and M. Apart from the tall cache assumption, this assumes that the order of B is larger than a logarithmic function of N. We recall that $\bar{N}$ represents the number of elements inserted. $\bar{N}$ is equal to N in the original algorithm and to $N/\log\log N$ with the pre-sieving step.

Assumption 9 *We suppose that* $\sqrt{M/B} > \log_{M/B}(N/M)$ *and* $\sqrt{M/B} > \log^2 \log N / \log_{M/B} \frac{N}{B}$.

We analyze the cost of the inserts and the inverse successor queries separately. Note that the height of the tree is $d = \lceil \log_{\sqrt{M/B}} N/M \rceil = O(\log_{M/B} N/M)$.

Lemma 5. *It is possible to always maintain simultaneously in memory all the pages of $\mathcal{P}_{p,q}$.*

Proof. These pages represent one node of size M and $\Theta\left(\log_{M/B}\frac{N}{M}\right)$ pages of size $\sqrt{MB}$. By Assumption 9, this sums to a number of elements m equal to:

$$m = O\left(M + \sqrt{MB}\log_{M/B}\frac{N}{M}\right)$$
$$m = O\left(M + \sqrt{MB}\sqrt{\frac{M}{B}}\right)$$
$$m = O\left(M\right)$$

Then, we assume, to compute the I/O complexity, that the set $\mathcal{P}_{p,q}$ is in memory at the beginning of the functions, and when p or q is changed, the new set $\mathcal{P}_{p,q}$ must be brought in memory at the end of the function. In other words, during the execution of `InverseSuccessor`, $\mathcal{P}_{p,q}$ is always in memory, and at the end, $\mathcal{P}_{p,q}$ plus all the pages related to y are in memory.

Lemma 6. *The complexity of performing all* `Insert` *and* `Insertion` *calls is* $\langle O\left(\frac{\bar{N}}{B}\log_{M/B}\frac{N}{B}\right), O\left(\bar{N}\log_{M/B}\frac{N}{B}\right)\rangle$*, assuming that when* `PartialFlush` *calls* `Insertion` *on Line 13, the corresponding page is already in memory.*

Proof. The cost of performing one flush, ignoring the cost of the recursive flushes, is $\langle O\left(\sqrt{M/B}\right), O\left(\sqrt{MB}\right)\rangle$. We must move $\sqrt{MB}$ elements to the next level; each requires $O(1)$ computation to find the page it occurs in. The cost to write out $\sqrt{MB}$ elements consecutively requires $O(\sqrt{M/B})$ I/Os. Then we get the above time, plus we may need to do an extra I/O per list when the block is initially brought in. Since there are $O(\sqrt{M/B})$ lists, this gives a total of $O(\sqrt{M/B})$ I/Os per flush. We move $\sqrt{MB}$ elements during this flush, so the per-element flush cost is $O(1/B)$.

Each element can be involved in at most $d = \Theta\left(\log_{M/B}\frac{N}{B}\right)$ flushes and insertions, the depth of the tree. Therefore, the amortized cost of the total number of flushes per element is $\langle O\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right), O\left(\log_{M/B}\frac{N}{B}\right)\rangle$.

When a call to `Insertion` occurs, only the actions listed below can have a non-null I/O cost and a non-constant RAM cost.

On Line 7, this extra insertion has a null I/O cost, because the deeper page is already in memory by Lemma 5.

On Line 10, the cost of this flush is already counted above.

On Line 11, we need to separate the cases. If the call of `Insertion` comes from `Insert`, then by Lemma 5, the root is already in memory so the I/O cost is null. If the call comes from `Flush`, then the cost is already counted above. If the call comes from `PartialFlush`, by hypothesis of the Lemma, the page P is already in memory so the I/O cost is null. If the call comes from the recursive call Line 7, the I/O cost is

null because by Lemma 5, this page is already in memory. In all cases, the RAM cost is constant.

On Line 2, by Lemma 5, the corresponding slot of $\mathcal{T}$ is already in memory.

Therefore, as there are less than N elements inserted, the complexity of performing the total number of insertions is $\langle O\left(\frac{\bar{N}}{B}\log_{M/B}\frac{N}{B}\right),\ O\left(\bar{N}\log_{M/B}\frac{N}{B}\right)\rangle$.

We now need to prove Lemma 7, which analyzes the complexity of a toy algorithm, Algorithm 6, before proving Lemma 8 on the I/O complexity of the `InverseSuccessor` calls.

Lemma 7. *The I/O complexity, amortized against the calls to* `Insertion`*, of Algorithm 6 is* $O\left(\frac{b}{\sqrt{MB}}+\frac{b}{B\log N}\right)$ *and the total number of recursive calls to* `PartialFlush` *is* $O\left(\frac{b}{\sqrt{MB}}\right)$.

Proof. Let's compute the cost of Algorithm 6, assuming it is launched in Algorithm 2, so that all the pages related to a are in memory.

```
Algorithm InverseSuccessorLoop(a, b)
    Input: Two numbers such that a < b, and a = p in Algorithm 2
1   x ← a;
2   while x < b do
3       x ← InverseSuccessor(x);
```

Algorithm 6: Theoretical study algorithm

First, we have to show that when `PartialFlush(`d, L_y, L_{tmp}`)` occurs, the I/O complexity, without the calls to `Insertion`, is $O\left(\sqrt{\frac{M}{B}}\left(d-\mathbf{LCA}(L_y,\ L_{tmp})\right)\right)$.

Indeed, during one call, only the page P has to be brought into memory, plus what the recursive call requires. The recursive call cannot be called on a level higher than $\mathbf{LCA}(L_y,\ L_{tmp})$, so there are at most $d-\mathbf{LCA}(L_y,\ L_{tmp})$ recursive calls. Note that for the last call, the page is also associated to L_y, so is already in memory. The cost to bring the page P into memory is $\Theta(1+k/B)=O(\sqrt{MB})$ where k is the number of elements contained in P at the time when the page is brought.

We now compute the I/O complexity of Algorithm 6. The subarray of $\mathcal{T}$ between a and b has to be brought into memory for the tests on $\mathcal{T}$ Line 7 in `InverseSuccessor`, which has a cost of $O\left(\frac{b-a}{B\log N}\right)$. Indeed, each machine word contains $\log N$ bits.

Then, we define the list $\mathcal{L}$ as in the proof of Theorem 8. $\mathcal{L}$ is the list of leaves $\mathcal{L}=\{l_1\dots l_t\}$ (which is the list of successive L_{tmp}) such that $l_1=L_a$, $l_t=L_b$ and for each i, a call to `PartialFlush(`d, l_i, l_{i+1}`)` has been triggered inside a call to `InverseSuccessor`. The number of pages contained in $\mathcal{L}$ is exactly

$$\sum_i\left(d-\mathbf{LCA}(l_i,\ l_{i+1})\right)$$

This term is bounded by the number of pages of the tree surrounding L_a and L_b. See Figure 6 for an illustration.

We split this set of pages into a set of $O\left(\frac{b-a}{M}\right)$ internal pages and $O\left(\frac{b-a}{\sqrt{MB}}\right)$ leaf pages. We count a cost of $O\left(\sqrt{\frac{M}{B}}\right)$ I/Os to bring into memory an internal page and

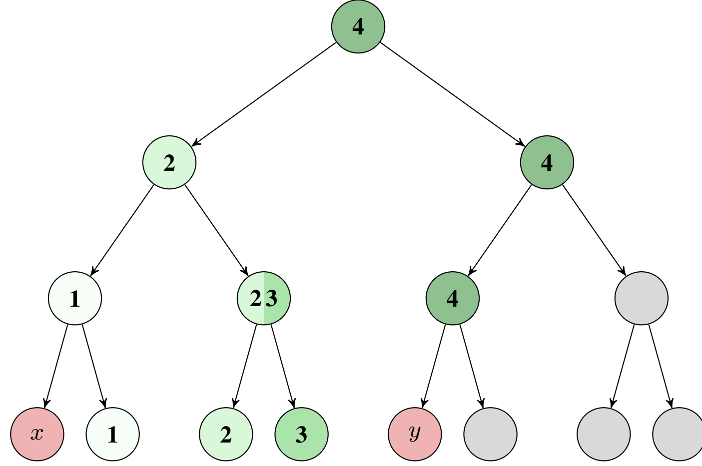

Fig. 6. Illustration of the successive calls to `PartialFlush` in a call of `InverseSuccessor`. The ancestors of x are assumed to be in memory. The calls are done on a leaf, then on its ancestors. The ith call to `PartialFlush` scans the nodes labeled by i.

$O\left(1 + \frac{k_P}{B}\right)$ I/Os to bring into memory a leaf page, where k_p is the number of elements in this page.

Therefore, the I/O complexity is

$$O\left(\frac{b-a}{M}\sqrt{\frac{M}{B}} + \frac{b-a}{\sqrt{MB}} + \sum_{P \in \mathcal{L}} \frac{k_P}{B}\right) = O\left(\frac{b-a}{\sqrt{MB}} + \frac{1}{B}(\textit{\# insertions in } \mathcal{T})\right)$$

Indeed, each element present in a leaf page will be added to $\mathcal{T}$. As each element can only be inserted once in $\mathcal{T}$, we can amortize this cost against the insertion cost, so the additional I/O cost of Algorithm 6 is

$$O\left(\frac{b-a}{\sqrt{MB}} + \frac{b-a}{B \log N}\right)$$

Now, we need to compute the total number of recursive calls of `PartialFlush`. During a call to `InverseSuccessorLoop(a, b)`, each page of the tree surrounding L_a and L_b makes one recursive call. So the number of recursive calls is $O\left(\frac{b}{\sqrt{MB}}\right)$, by the same argument as above.

Lemma 8. *The complexity of performing all* `InverseSuccessor` *calls, in addition to the total cost of the* `Insertion` *calls, is*

$$\langle O\left(\frac{N \log\log N}{B \log N}\right), O\left(\frac{N \log\log N}{\log N}\right)\rangle$$

Furthermore, when a call to `Insertion` *is done, the concerned page is already in memory.*

Proof. We can group the calls to `InverseSuccessor` in Algorithm 2 in one call to `InverseSuccessorLoop(1,` $\sqrt{N}$`)` for the ps and for each p, one call to `InverseSuccessorLoop(`p`,` $\frac{n}{p}$`)` for the qs associated. Indeed, as by Lemma 5, the appropriate pages are kept in memory, the I/O complexity is not changed by this modification.

By Lemma 7, the total I/O complexity of all the calls to `InverseSuccessor` is then:

$$C_{I/O} = O\left(\frac{\sqrt{N}}{B} + \sum_{p\in\mathbb{P},\ p<\sqrt{N}} \left(\frac{N}{p}\left(\frac{1}{\sqrt{MB}} + \frac{1}{B\log N}\right)\right)\right)$$

$$= O\left(\frac{\sqrt{N}}{B} + \frac{N\log\log N}{\sqrt{MB}} + \frac{N}{B}\frac{\log\log N}{\log N}\right)$$

$$= O\left(\frac{N\log\log N}{B\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)$$

Concerning the RAM complexity, in the total calls to `PartialFlush`, the cost of moving the elements is bounded by the complexity of the total calls to `Insertion`. Indeed, only a constant RAM complexity per element per level is needed: to move it from a list P^* to its list P_Q, which can happen only once per element per page. For the leaves level, again, only a constant cost per element is required. Therefore, we amortize this cost against the insertion cost, and do not count it here.

We have to add a constant cost per recursive call, to take into account the calls where no element is moved, which, by Lemma 7, sums to:

$$C^1_{\mathrm{RAM}} = O\left(\frac{N}{\sqrt{MB}}\log\log N\right)$$

The only term remaining to compute the total RAM complexity of `InverseSuccessor` is the term without counting the calls to `PartialFlush`. We know that there are $O(\bar{N} + \sqrt{N}) = O(\bar{N})$ calls to `InverseSuccessor`. Indeed, there is one call per modification of the value of p or q.

After each call to `NextIn`$\mathcal{T}$ Line 4, we have $\mathcal{T}[y] = 0$. Therefore, either a call to `PartialFlush` is triggered Line 5, or the call terminates. There are $O\left(\frac{N}{\sqrt{MB}}\log\log N\right)$ calls to `PartialFlush`, so $O\left(\frac{N}{\sqrt{MB}}\log\log N + \bar{N}\right)$ calls to `NextIn`$\mathcal{T}$.

Now, note that the RAM cost of the function `NextIn`$\mathcal{T}$ called on x and returning y is $O\left(1 + \frac{y-x}{\log N}\right)$, as each line executes in constant time and a machine word contains $\log N$ bits. Therefore, the remaining RAM term is equal to :

$$
\begin{aligned}
C^2_{\mathrm{RAM}} &= O\left(\frac{N}{\sqrt{MB}}\log\log N + \bar{N} + \sum_{p\in\mathbb{P},\ p<\sqrt{N}}\left(\frac{N}{p\log N}\right)\right) \\
&= O\left(\bar{N} + N\log\log N\left(\frac{1}{\sqrt{MB}} + \frac{1}{\log N}\right)\right) \\
&= O\left(\bar{N} + \frac{N\log\log N}{\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)
\end{aligned}
$$

So the additional RAM complexity, with regards to the cost of the insertions, is:

$$
C_{\mathrm{RAM}} = O\left(\frac{N\log\log N}{\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)
$$

Now, note that by Assumption 9, we have $\sqrt{M/B} = \Omega(\log^2\log N/\log_{M/B}\frac{N}{B})$, so

$$
\frac{N\log\log N}{\sqrt{MB}} = \frac{N}{B}\frac{\log\log N}{\sqrt{M/B}} = O\left(\frac{N\log_{M/B}\frac{N}{B}}{B\log\log N}\right)
$$

Therefore, the total additional cost of the `InverseSuccessor` calls is:

$$
\langle O\left(\frac{N\log\log N}{B\log N}\right),\ O\left(\frac{N\log\log N}{\log N}\right)\rangle
$$

Therefore, by combining the above results, we get

Theorem 10. *The linear sieve of Eratosthenes implemented with buffer trees, assuming that* $\sqrt{M/B} > \log_{M/B} N$ *and* $\sqrt{M/B} > \log^2_{M/B}(N/B)/\log\log N$*, has a complexity of*

$$
\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right),\ O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle
$$

and a space requirement of

$$
O\left(N\left(\sqrt{\frac{B}{M}} + \frac{1}{\log N}\right)\right)
$$

Proof. Indeed, the insertions and flushes, including the last call to `GetSet` that empties the tree, have a complexity of

$$
\begin{aligned}
&\langle O\left(\frac{\bar{N}}{B}\log_{M/B}\frac{N}{B}\right),\ O\left(\bar{N}\log_{M/B}\frac{N}{B}\right)\rangle \\
=&\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right),\ O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle
\end{aligned}
$$

and the additional complexity of the pre-sieve step and the `InverseSuccessor` calls is

$$\langle O\left(\frac{N\log\log N}{B\log N}\right),\ O\left(\frac{N\log\log N}{\log N}\right)\rangle$$

Now, as we have $M/B = O(N)$ and $\log N = \Omega(\log^2\log N)$, we have

$$\frac{\log\log N}{\log N} = O\left(\frac{1}{\log\log N}\right) = O\left(\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)$$

So the global complexity for the entire algorithm is

$$\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right),\ O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle$$

Concerning the space complexity:

For the pre-sieve, we need a space $O(N/\log N)$. For the whole tree but the leaves, we need a space of $O(N\sqrt{B/M})$. For the bitarray, we need a space $O(N/\log N)$.

For the leaf pages, we can actually shrink their size by a factor $\log N$, so that they can contain at most $\sqrt{MB}/\log N$ elements. Indeed, as these elements will be flushed in a portion of $\mathcal{T}$ that fits in $\sqrt{MB}/\log N$ machine words, the amortized cost of such a flush per element is $O(1/B)$, which is the desired bound. The space used per each page is then $O(1+\sqrt{MB}/\log N)$. So the space used to store all the $O(N/\sqrt{MB})$ leaves is $O\left(N/\sqrt{MB}+N/\log N\right)$. Note that this space optimization has not been depicted in the pseudo-code for clarity.

Therefore, the space requirement is

$$O\left(N\left(\sqrt{\frac{B}{M}}+\frac{1}{\log N}\right)\right)$$

B Sieve of Atkin

We present here the pseudo-code depicting the different versions of the sieve of Atkins [6]. The two first sections only reformulate the original algorithms, and Appendix B.3 depicts our contribution: an I/O-efficient version of the sublinear sieve of Atkins.

B.1 Level Curve Tracing

We first present the non-optimized version of the sieve of Atkins, which has a linear time complexity, and does not optimize I/Os.

If $M = N^{1/2+o(1)}$, then the sieve can be performed in memory. Let $f(x,y)$ be a binary quadratic form and let $L = \{(x,y)\in\mathbb{N}^2 | f(x,y)\le N\}$. Suppose that M can hold an array of Δ values. Then we can subdivide $L = L_0\cup L_1\cup\cdots\cup L_{\lceil n/m\rceil-1}$, where $L_i = \{(x,y)\in L | i\Delta < f(x,y)\le (i+1)\Delta\}$, and use our array to count the values of f over each L_i. For conciseness, we only describe the algorithm for the first quadratic form, but the other cases are similar.

```
generate a list of primes P up to √N by any reasonable means;
for i ← 0 to ⌈N/Δ⌉ − 1 do
    A[1] ← 0, A[2] ← 0, ..., A[Δ] ← 0;
    x ← 1;
    while x^2 + 4 ≤ (i + 1)Δ do
        y ← ⌈1/2 √((iΔ) − x^2)⌉, k ← x^2 + 4y^2;
        while k ≤ (i + 1)Δ do
            if k ≡ 1 (mod 4) then
                A[k − iΔ] ← A[k − iΔ] + 1;
            y ← y + 2, k ← x^2 + 4y^2;
        x ← x + 2;
    foreach p ∈ P do
        j ← ⌈iΔ/p^2⌉;
        while p^2 j ≤ (i + 1)Δ do
            A[p^2 j − iΔ] ← 0, j ← j + 1;
    for j ← 1 to Δ do
        if Odd(A[j + iΔ]) then
            Print(j + iΔ);
```

Algorithm 7: The "linear" sieve of Atkin for primes congruent to 1 mod 4

Each L_i is the region between two level curves, and the algorithm operates on them individually. For each x the algorithm calculates the smallest viable y within the region and then keeps incrementing it until it escapes the region. Because of the size of M and the choice of Δ, each x with $f(x, 1) \leq (i + 1)\Delta$ has at least one y such that $(x, y) \in L_i$. Thus the overhead is at most linear.

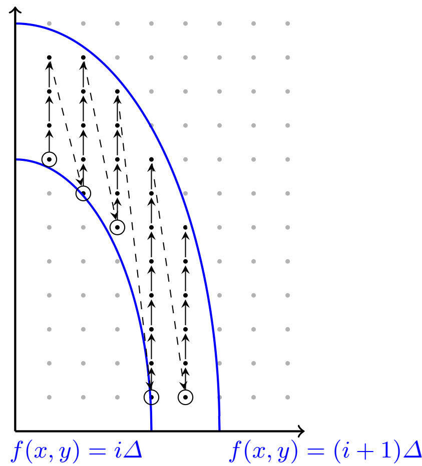

Fig. 7. This figure depicts the algorithm "tracing" the points between level curves of f. The y values of the encircled points must be calculated.

B.2 Pre-sieving with a wheel

Here we show the algorithm with a wheel sieve on L. For brevity we describe the strategy for a general binary form $f(x, y)$. This can then be implemented for each of the binary forms from Theorem 4 in Section 4. This algorithm has a time complexity of $N/\log\log N$, and is the main contribution of [6].

Let $W = 12 * \prod_{i=1}^{\sqrt{\log N}} p_i = N^{o(1)}$, and let $U \subseteq [W]^2$ be the set of points (x, y) such that $f(x, y)$ is a unit mod W. Because $f(x + aW, y + bW) \equiv f(x, y) \pmod W$ for any $a, b \in \mathbb{Z}$, we can reduce the domain on which we work to the W-translates of U. This is because the value of f on each of the remaining points must be of the form $kW + c$ where c shares a factor with W. Thus we loop through U, and for each point $d = (x, y) \in U$, we count the occurrences of the values of f on each of the W-translates within L. This is illustrated for a particular d in Appendix B.2. Then those values which occur an odd number of times will be primes or squareful. Those squareful numbers must be sieved, which can be done in a manner analogous to the sieve of Eratosthenes.

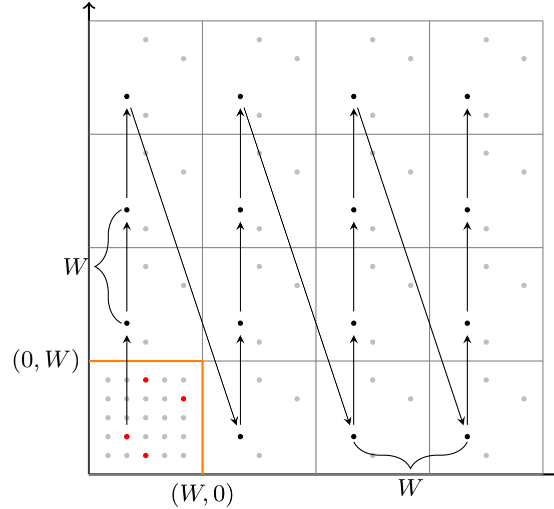

Fig. 8. A visualization of the wheel pre-sieve. Here the red points are the unit-valued points on $[W]^2$, and the grey points in $[W]^2$ have been eliminated. The black points are W-translates of $d = (x, y)$, and the grey points outside of $[W]^2$ are W-translates of other points in U.

Because of the choice of W, it follows from Merten's Theorem that $|U| = O(|W|^2/\log\log N)$. Thus the counting phase of the algorithm will take $O(N/\log\log N)$ time. Since we have already sieved the first $\sqrt{\log N}$ primes, it can be shown that the squarefree sieve can also be completed in $O(N/\log\log N)$.

B.3 Using a priority queue

The pseudo-code functions below describe our variant of the sieve of Atkins. Priority queues are used instead of arrays to improve the I/O efficiency. This version requires

$O\left(N^{1+o(N)}\right)$ space, but it can be segmented among level curves as in Appendix B.1 to use $O\left(N^{1/2+o(N)}\right)$. Note that in what follows objects are passed to functions as references. The code for some functions has been omitted.

```
W ← ComputeWheelModulus(N);
U ← ComputeUnitsMod(W);
create three empty lists of pairs L_1, L_2 and L_3;
L1 ← ConstructPrincipalDomain(W,U,1,4,1,4);
L2 ← ConstructPrincipalDomain(W,U,3,1,7,12);
L3 ← ConstructPrincipalDomain(W,U,3,-1,11,12);
create an empty min priority queue Q that only stores values;
InsertValuesFromDomain(Q, W, L_1, 1, 4) ;
InsertValuesFromDomain(Q, W, L_2, 3, 1) ;
InsertValuesFromDomain(Q, W, L_3, 3, −1) ;
Q.Insert(∞);
create an empty queue S;
S ← EliminateEven(Q);
Print all the primes dividing W;
EliminateSquaresAndPrint(S);
```

Algorithm 8: The main process of the Sieve of Atkin in external memory

```
ConstructPrincipalDomain(W, U, a, b, c, d)
    create an empty list L;
    foreach (x, y) ∈ [W]^2 do
        if ax^2 + by^2 ∈ U + WZ and ax^2 + by^2 ≡ c (mod d) then
            L.Add((x, y));
    return L;
```

Algorithm 9: ConstructPrincipalDomain: Relative to $f(x,y) = ax^2 + by^2$, returns a list of all the unit-valued (mod W) points (x,y) in $[W]^2$ with $f(x,y) \equiv c \pmod{d}$

```
InsertValuesFromDomain(Q, W, L, a, b)
    foreach (x, y) ∈ L do
        i ← 1;
        while a(x + iW)^2 + by^2 ≤ N do
            j ← 1;
            while a(x + iW)^2 + b(y + jW)^2 ≤ N do
                Q.Insert((x + iW)^2 + 4(y + jW)^2);
                j ← j + 1;
            i ← i + 1;
    return;
```

Algorithm 10: InsertValuesFromDomain: Relative to $f(x, y) = ax^2 + by^2$, inserts the value of f on every W-translate of every point in L into Q.

```
EliminateEven(Q)
    create an empty queue S;
    p' ← 0, c ← 0, k ← 0;
    while Q ≠ ∅ do
        p ← Q.Extract-Min();
        if p ≠ p' then
            if Odd(c) then
                k ← k + 1, S.Enqueue(p');
            c ← 1;
        else
            c ← c + 1;
        p' ← p;
    return S;
```

Algorithm 11: EliminateEven: Returns a queue with all the values in Q that occur an odd number of times.

```
EliminateSquaresAndPrint(S)
    create an empty key-sensitive min priority queue Q' that can store ⟨key, value⟩ pairs;
    c ← S.Dequeue();
    Q'.Insert(⟨c, c^2⟩);
    while S ≠ ∅ do
        ⟨p, v⟩ ← Q'.Find-Min();
        c ← S.Dequeue();
        while v < c do
            Q'.Extract-Min();
            Q'.Insert(⟨p, v + p^2⟩);
            ⟨p, v⟩ ← Q'.Find-Min();
        if v = c then
            while v = c do
                Q'.Extract-Min();
                Q'.Insert(⟨p, v + p^2⟩);
                ⟨p, v⟩ ← Q'.Find-Min();
        else
            Print(c);
            Q'.Insert(⟨c, c^2⟩);
    return;
```

Algorithm 12: EliminateSquaresAndPrint: Prints the squarefree numbers in S, which in this context are the primes (excluding those removed by the wheel, which are printed in the main procedure).

C Sieving the first $\sqrt{\log N}$ primes

We describe here a method to compute the numbers smaller than N that are co-prime to the first $\sqrt{\log N}$ primes. This method is used by the algorithm in Appendix A. First, we present the pseudo-code of the algorithm, before proving its correctness and its complexity.

Data: $S = \{2, 3, \cdots, N\}$
Result: A bit vector expliciting the co-primes to $\{p_1 \dots p_{\sqrt{\log N}}\}$ up to N
Compute the first $\sqrt{\log N}$ primes $p_1 \dots p_{\sqrt{\log N}}$;
Compute $P = \prod_{1<i<\sqrt{\log N}} p_i$;
Sieve $S_P = \{1 \dots P\}$ with the first primes in a bit vector s_P (value `COMPOSITE` or `COPRIME`);
Compute s'_P equal to s_P but with bits before $p_{\sqrt{\log N}}$ set to `COMPOSITE`;
Concatenate s_P with copies of s'_P to form a N-long bit vector s;
return s;

Algorithm 13: Low-primes sieving

Lemma 9. *This algorithm is correct: the returned bit vector explicits the co-primes to to $\{p_1 \dots p_{\sqrt{\log N}}\}$ up to N.*

Proof. We need to show that for all $x < N$, $s[x]$ is `COMPOSITE` if and only if there exists $k \leq \sqrt{\log N}$ such that p_k divides x.

First, suppose $x < P$. This property is ensured by the explicit sieving of S_P.

If x is greater than P, let $i = x \mod P$. Then, for any $k \leq \sqrt{\log N}$, p_k divides x if and only if p_k divides i. If $i \leq p_{\sqrt{\log N}}$, there exists $k \leq \sqrt{\log N}$ such that p_k divides i, so x. And $s[x] = s'_P[i] = 0$, so the property is true. If $i > p_{\sqrt{\log N}}$, there exists $k \leq \sqrt{\log N}$ such that p_k divides i if and only if $s_P[i] =$ `COMPOSITE`, and $s[x] = s'_P[i] = s_P[i]$.

Theorem 11. *The complexity of this algorithm is $\langle O\left(N/(B \log N)\right), O\left(N/\log N\right)\rangle$.*

Proof. First, note that P is equivalent to

$$P = \exp\left((1 + o(1))\sqrt{\log N} \log\log N\right) = O(N^{\frac{\log\log N}{\sqrt{\log N}}})$$

Computing the first primes, P, and s'_P from s_P do not exceed the bound.

Sieving S_P successively with $\sqrt{\log N}$ primes has a time complexity of $O(\sqrt{\log N} P)$ and an I/O complexity of $O(\sqrt{\log N} P/(B \log N))$, which does not exceed the bound.

Creating s from s'_P means achieving N/P copies of s'_P, which has a time complexity of $O(N/\log N)$ and an I/O complexity of $O(N/(B \log N))$.

C.1 Sieve of Eratosthenes using a RAM-efficient external-memory priority queue.

The RAM and I/O performance of sieve of Eratosthenes can be improved using the recently proposed RAM-efficient external-memory priority queue [5] in a folklore priority queue based implementation of the sieve.

The straightforward folklore sieve implementation is shown in Figure $1(a)$. The priority queue Q stores $\langle k, v \rangle$ pairs, where k is a prime (***key***) and v is its multiple (***value***). Initially, $\langle 2, 4 \rangle$ is inserted into Q. When a pair $\langle k, v \rangle$ is deleted from Q, we check if v is two more than the last value v' deleted, and if so, $p = v - 1$ is not a multiple of any prime, and hence must be a prime itself. We then insert $\langle p, p^2 \rangle$ into Q. We always insert the next multiple $v + k$ of k into Q.

The performance bounds of the sieve above with a RAM-efficient external-memory priority queue [5] Q is given by the theorem below. The bounds follow from the observation that the sieve performs $\Theta\left(\sum_{prime\ p \in [1, \sqrt{N}]} \frac{N}{p}\right) = \Theta\left(N \log\log N\right)$ operations on Q costing $\langle O\left(\frac{1}{B} \log_{\frac{M}{B}} N\right),\ O\left(\log_{\frac{M}{B}} N + \log\log M\right)\rangle$ each.

Theorem 12. *The sieve of Eratosthenes (shown in Figure 1(a)) implemented using a RAM-efficient external-memory priority queue [5] has a complexity of* $\langle O\left(\text{SORT}\left(N \log\log N\right)\right),\ O\left(N \log\log N \left(\log_{\frac{M}{B}} N + \log\log M\right)\right)\rangle$ *and uses* $O\left(\sqrt{N}\right)$ *space for sieving primes in* $[1, N]$.

C.2 Sieve of Sorenson on a Segment

The sieve of Sorenson can be adapted to sieve for primes on the interval $[a, b]$ provided a sufficiently large pseudosquare table is available. We further assume that $M = \Omega(s) = \Omega(\pi(p) \log^2 b)$, where here and below p is determined as above but by b rather than N. In that case, we can determine the primes up to s in $O(s)$. We then perform the initial wheel sieve phase in memory on each segment, which takes $O((b-a)+s) = O((b-a)+\pi(p)\log^2 b)$ operations and $O((b-a)/B+s/B+1) = O((b-a)/B+\pi(p)\log^2 b/B+1)$ I/Os.

In the second phase we must exponentiate each number in the segment for (potentially) each pseudoprime up to p. It takes $\langle O(\frac{b-a}{B}+1), O((b-a)\pi(p))\rangle$.

In the third phase we can for each $k = 2, 3, \ldots, \lfloor \log b \rfloor$ compute $r = \lceil a^{1/k} \rceil$. Then we create the list of perfect powers in $[a, b]$ by taking $r^k, (r+1)^k, \ldots$ for each k until we reach b. This list will have $O((\sqrt{b} - \sqrt{a}) \log b)$ elements and can be computed in $O((\sqrt{b} - \sqrt{a}) \log^2 b + \log^2 b)$. Thus all the perfect powers can be sorted and removed from the candidate list in $\langle O((b-a)/B), O((b-a) + \log^2 b)\rangle$. We have shown:

Theorem 13. *On a segment from a to b, the sieve of Sorenson runs in* $\langle O((b - a + \pi(p)\log^2 b)/B + 1), O((b-a)\pi(p) + \pi(p)\log^2 b)\rangle$