Efficient Computation of the Hard Special Leaves in the Combinatorial Prime Counting Algorithms

Kim Walisch

June 13, 2025

Abstract

This paper introduces an optimized method for computing the contribution of the hard special leaves in combinatorial prime counting algorithms. Unlike previous approaches, which rely on binary indexed trees for counting in the hard special leaves algorithm, the proposed method employs an alternative tree-like data structure with fewer than $O(\log z)$ levels, offering improved memory locality and scalability. By better balancing sieve and count operations, the new algorithm achieves a runtime complexity improvement of at least a factor of $O(\log \log x)$ over traditional implementations.

1 Introduction

The combinatorial type prime counting algorithms (Lagarias-Miller-Odlyzko [1], Deléglise-Rivat [3], Gourdon [4]) consist of many formulas and the formula that usually takes longest to compute and is by far the most difficult to implement is the formula of the so-called hard special leaves. Unlike the easy special leaves, each of which can be computed in $O(1)$ time using a $\pi(x)$ lookup table, the hard special leaves require evaluating the partial sieve function $\phi(x, a)$, which generally cannot be computed in constant time.

The implementation of the hard special leaves algorithm in the primecount C/C++ library, developed by the author, is different from the algorithms that have been described in any of the combinatorial prime counting papers thus far. This paper reviews the evolution of primecount's implementation and presents an alternative counting method devised by the author in February 2020. This alternative counting method improves the balancing of sieve and count operations in the hard special leaf algorithm and thereby improves its runtime complexity by a factor of at least $O(\log \log x)$. The alternative counting method uses a new tree-like data structure, with improved memory locality and scalability, and using fewer than $O(\log z)$ levels (tree depth) where each node has $O(z^{\frac{1}{levels}})$ children. This data structure replaces the binary indexed tree (a.k.a. Fenwick tree [2]), which has been used for counting in the hard special leaves algorithm in all previously published papers on combinatorial prime counting algorithms.

2 Basic algorithm

Implementing the hard special leaves formula requires use of a prime sieve. The algorithm is basically a modified version of the well known segmented sieve of Eratosthenes which consists of two main parts that are executed alternately:

1. Sieve out primes and multiples of primes.
2. Count the number of unsieved elements in the sieve array.

3 Speed up counting using binary indexed tree

Since there is a large number of leaves for which we have to count the number of unsieved elements in the sieve array Lagarias-Miller-Odlyzko [1] have suggested using a binary indexed tree data structure (a.k.a. Fenwick tree [2]) to speedup counting. For any number n the binary indexed tree allows to count the number of unsieved elements $\leq n$ using only $O(\log n)$ operations. However, the binary indexed tree must also be updated whilst sieving which slows down the sieving part of the algorithm by a factor of $O(\log n / \log \log n)$ operations. All more recent papers about the combinatorial type prime counting algorithms that the author is aware of, have also suggested using the binary indexed tree data structure for counting the number of unsieved elements in the sieve array. Tomás Oliveira e Silva's paper [5] contains a simple C implementation of a count function using a binary indexed tree:

```
// Count the number of unsieved elements <= pos in
// the sieve array using a binary indexed tree.
int count(const int* tree, int pos)
{
  int sum = tree[pos++];
  while (pos &= pos - 1)
      sum += tree[pos - 1];
  return sum;
}
```

4 Alternative counting method

Despite its theoretical advantages, the binary indexed tree data structure has two major practical drawbacks: it consumes substantial memory, as each thread must allocate its own instance, and more critically, it performs poorly in practice due to its non-sequential memory access patterns, which modern CPUs are notoriously bad at. For this reason, many programmers who have implemented any of the combinatorial prime counting algorithms (Christian Bau 2003, Dana Jacobsen 2013, James F. King 2014, Kim Walisch 2014) have chosen to avoid using the binary indexed tree altogether. The method that has turned out to perform best thus far has been to remove the binary indexed tree, which speeds up the sieving part of the algorithm by a factor of $O(\log n / \log \log n)$ and count the number of unsieved elements by simply iterating over the sieve array.

There are many known optimizations to speed up counting:

- Using the POPCNT instruction in combination with a bit sieve array allows counting many unsieved sieve elements using a single instruction. This improves the runtime complexity by a large constant factor. In primecount, this factor is 240.
- One can keep track of the total number of unsieved elements that are currently in the sieve array as the total number of unsieved elements is used frequently (once per sieving prime in each segment).

- One can batch count the number of unsieved elements in the sieve array for many consecutive leaves i.e. instead of starting to count from the beginning of the sieve array for each leaf we resume counting from the last sieve index of the previous leaf.

```
// Count the number of unsieved elements (1 bits) in
// the sieve array using the POPCNT instruction.
uint64_t count(const uint64_t* sieve, uint64_t startIndex, uint64_t stopIndex)
{
    uint64_t cnt = 0;

    for (uint64_t i = startIndex; i < stopIndex; i++)
        cnt += __builtin_popcountll(sieve[i]);

    return cnt;
}
```

The alternative method, incorporating the three optimizations described above, was used in primecount up to version 5.3 (January 2020). According to our benchmark results, this approach runs up to 3× faster than implementations based on the binary indexed tree data structure. However, a fundamental issue remains: all known alternative algorithms that avoid binary indexed trees exhibit worse asymptotic runtime complexity. The question, then, is why they are faster in practice. There are two main reasons: First, the memory access patterns in the alternative algorithms are mostly sequential, which is particularly important for large computations where the data structures do not fit into the CPU's caches. Second, these algorithms can count multiple unsieved elements using a single POPCNT instruction, resulting in a significant constant factor speedup. In primecount, this factor is 240.

Despite their practical advantages, the alternative methods suffer from inferior asymptotic runtime complexity, which leads to performance degradation at large input sizes. In particular, primecount's implementation of the hard special leaves formula—based on one of these alternative approaches—begins to scale poorly beyond $x = 10^{23}$. This limitation became a significant bottleneck during record computations. For instance, the calculation of $\pi(10^{28})$, initially projected to complete in approximately 8 months, ultimately required 2.5 years.

5 Improved alternative counting method

The scaling issue discussed in the Alternative Counting Method section—present in algorithms that do not use a binary indexed tree—is caused by counting the number of unsieved elements by simply iterating over the sieve array. When there are many consecutive leaves that are close to each other then simply iterating over the sieve array for counting the number of unsieved elements works great. However, when there are very few consecutive leaves in the current segment the alternative method becomes inefficient. In order to compute the special leaves we need to sieve up to some limit z. Since we are using a modified version of the segmented sieve of Eratosthenes the size of the sieve array is $O(\sqrt{z})$. This means that if e.g. there is a single leaf in the current segment we will use $O(\sqrt{z})$ operations to count the number of unsieved elements in the sieve array, whereas the binary indexed tree would have used only $O(\log z)$ operations. This is too much, this deteriorates the runtime complexity of the algorithm.

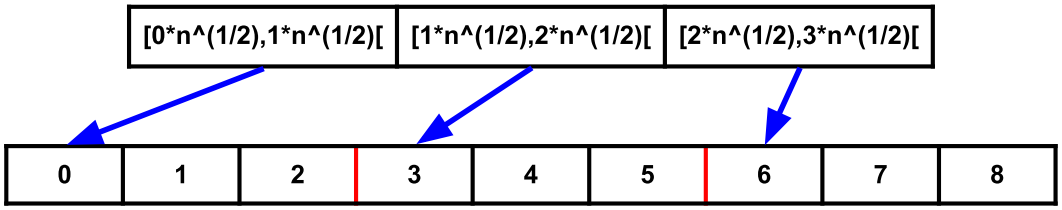

Figure 1: Two counter levels

Now that we have identified the problem, we can think about whether it is possible to further improve counting by more than a constant factor in our alternative algorithm. It turns out this is possible and even relatively simple to implement: We add a counter array to our sieving algorithm. The counter array has a size of $O(\sqrt{segment\ size})$, with $segment\ size = \sqrt{z}$. Each element of the counter array contains the current count of unsieved elements in the sieve array for the interval $[i \times \sqrt{segment\ size}, (i+1) \times \sqrt{segment\ size}[$. Similar to the algorithm with the binary indexed tree data structure this counter array must be updated whilst sieving i.e. whenever an element is crossed off for the first time in the sieve array we need to decrement the corresponding counter element. However, since we only need to decrement at most 1 counter when crossing off an element in the sieve array, this does not deteriorate the sieving runtime complexity of the algorithm (unlike the binary indexed tree which deteriorates sieving by a factor of $\log z / \log \log z$). We have to give credit to Christian Bau here, who already used such a counter array back in 2003; however, he chose a counter array size of $O(segment\ size)$ with a constant interval size, which does not improve the runtime complexity.

```
// Sieve out a bit from the sieve array and update the
// counter array if the bit was previously 1
is_bit = (sieve[i] >> bit_index) & 1;
sieve[i] &= ~(1 << bit_index);
counter[i >> counter_log2_dist] -= is_bit;
```

Now whenever we need to count the number of unsieved elements in the sieve array, we can quickly iterate over the new counter array and sum the counts. We do this until we are close $< O(\sqrt{segment\ size})$ to the limit up to which we need to count. Once we are close, we switch to our old counting method: we simply iterate over the sieve array and count the number of unsieved elements using the POPCNT instruction. With this modification we improve the runtime complexity for counting the number of unsieved elements for a single leaf from $O(segment\ size)$ to $O(\sqrt{segment\ size})$. Below is the C++ implementation of this counting algorithm using two counter levels from the primecount C/C++ library [6].

```
// Count 1 bits inside [0, stop]
uint64_t Sieve::count(uint64_t stop)
{
    uint64_t start = prev_stop_ + 1;
    prev_stop_ = stop;

    // Quickly count the number of unsieved elements (in
    // the sieve array) up to a value that is close to
    // the stop number i.e. (stop - start) < segment_size^(1/2).
    // We do this using the counter array, each element
    // of the counter array contains the number of
    // unsieved elements in the interval:
    // [i * counter_dist, (i + 1) * counter_dist[.
    while (counter_start_ + counter_dist_ <= stop)
    {
        counter_sum_ += counter_[counter_i_++];
        counter_start_ += counter_dist_;
        start = counter_start_;
        count_ = counter_sum_;
    }

    // Here the remaining distance is relatively small i.e.
    // (stop - start) < counter_dist, hence we simply
    // count the remaining number of unsieved elements by
    // linearly iterating over the sieve array.
    count_ += count(start, stop);
    return count_;
}
```

When this improvement was first implemented in primecount, it was initially believed to be a minor optimization that would not fundamentally resolve the scaling issue in the hard special leaves implementation for large inputs. The alternative counting method was initially estimated to have a runtime complexity of approximately $O(number\ of\ special\ leaves \times \sqrt{segment\ size})$, since counting the number of unsieved elements for a single leaf requires $O(\sqrt{segment\ size})$ operations. However, empirical measurements revealed that the average number of count operations per leaf was significantly lower than expected. It was later observed that batch counting the number of unsieved elements for many consecutive leaves improves the runtime complexity by more than a constant factor.

When this improved alternative counting method was implemented in primecount, it fully resolved the long-standing severe scaling issue in the computation of special leaves. While no performance gains are observed below $x = 10^{20}$, the method becomes increasingly effective at higher input sizes. At $x = 10^{25}$, it already achieves a $2\times$ speedup compared to the previous implementation. This optimization is particularly beneficial when used with the Deléglise–Rivat [3] and Gourdon [4] variants of the combinatorial prime counting algorithm, where the average distance between consecutive special leaves is relatively large. In contrast, the Lagarias–Miller–Odlyzko [1] algorithm features much smaller distances between special leaves, limiting the practical effectiveness of the

new method in that context.

6 Gradually increase counter distance

Thus far we have focused on improving counting for the case when there are very few leaves per segment which are far away from each other. Generally there is a very large number of leaves that are close to each other at the beginning of the sieving algorithm, and gradually as we sieve up the leaves become sparser and the distance between the leaves increases. We can take advantage of this property, we start with a counter array whose elements span over small intervals and then gradually increase the interval size. We can update the counter size and distance e.g. at the start of each new segment, since the counter array needs to be reinitialized at the start of each new segment anyway. The ideal counter distance for the next segment is $\sqrt{average\ leaf\ distance}$. In practice we can approximate the average leaf distance using $\sqrt{segment\ low}$. The author's measurements using primecount indicate that gradually increasing the counter distance further improves counting by a small factor. This optimization is primarily useful when using a very small number of counter levels, for example 2.

```
// Ideally each element of the counter array
// should represent an interval of size:
// min(sqrt(average_leaf_dist), sqrt(segment_size))
// Also the counter distance should be regularly
// adjusted whilst sieving.
//
void Sieve::allocate_counter(uint64_t segment_low)
{
    uint64_t average_leaf_dist = sqrt(segment_low);
    counter_dist_ = sqrt(average_leaf_dist);
    counter_dist_ = nearest_power_of_2(counter_dist_);
    counter_log2_dist_ = log2(counter_dist_);

    uint64_t counter_size = (sieve_size_ / counter_dist_) + 1;
    counter_.resize(counter_size);
}
```

7 Multiple levels of counters

It is also possible to use multiple levels of counters. In this case, the data structure becomes a tree where each node has $O(segment\ size^{\frac{1}{levels}})$ children ($levels$ corresponds to the tree depth) and each node stores the current number of unsieved elements in an interval of size $O(segment\ size^{\frac{levels-level}{levels}})$ from the sieve array. The last level of this tree corresponds to the sieve array used in the algorithm. Like in the original algorithm with the binary index tree, we can count the number of unsieved elements $\leq n$ in the sieve array using the new tree-like data structure by traversing the tree from the root node to the bottom and summing the current number of unsieved elements stored in each node.

For example, consider the case with three counter levels, which requires the use of $3 - 1 = 2$

counter arrays. We only need to use two counter arrays because for the last level we will count the number of unsieved elements by iterating over the sieve array. For each level the size of the counter array can be calculated using $segment\ size^{\frac{level}{levels}}$ and the interval size of the counter array's elements can be calculated using $segment\ size^{\frac{levels-level}{levels}}$. Hence, our first counter array (1st level) is coarse-grained, its elements span over large intervals of size $O(segment\ size^{\frac{2}{3}})$. This means that each element of the first counter array contains the current number of unsieved elements in the interval $[i \times segment\ size^{\frac{2}{3}}, (i+1) \times segment\ size^{\frac{2}{3}}[$. Our second counter array (2nd level) is more fine-grained, its elements span over smaller intervals of size $O(segment\ size^{\frac{1}{3}})$. Hence, each element of the second counter array contains the current number of unsieved elements in the interval $[i \times segment\ size^{\frac{1}{3}}, (i+1) \times segment\ size^{\frac{1}{3}}[$. Now when we need to count the number of unsieved elements $\leq n$, we first iterate over the first counter array and sum the counts of its elements. Once the remaining distance becomes $< segment\ size^{\frac{2}{3}}$, we switch to our second counter array and sum the counts of its elements until the remaining distance becomes $< segment\ size^{\frac{1}{3}}$. When this happens, we count the remaining unsieved elements $\leq n$ by simply iterating over the sieve array. Below is a graphical representation of 3 counter levels with a segment size of 8 (last level). At each level we perform at most $segment\ size^{\frac{1}{levels}}$ count operations to find the number of unsieved element $\leq n$.

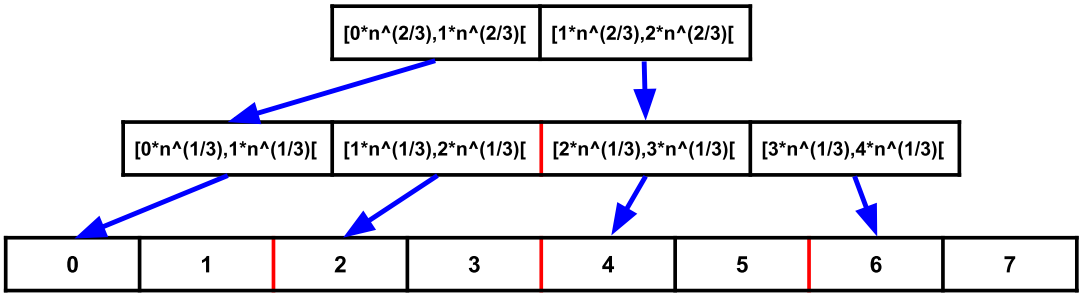

Figure 2: Three counter levels

Using 3 counter levels reduces the worst-case complexity for counting the number of unsieved elements for a single special leaf to $O(segment\ size^{\frac{1}{3}})$, but on the other hand it slightly slows down sieving as we also need to update the two counter arrays whilst sieving. The author has benchmarked using 3 counter levels vs. using 2 counter levels in primecount. When using 3 counter levels, the computation of the hard special leaves used 6.69% more instructions at $x = 10^{20}$, 6.55% more instructions at $x = 10^{21}$, 5.73% more instructions at $x = 10^{22}$ and 5.51% more instructions at $x = 10^{23}$. Hence for practical use, using 2 counter levels (i.e. a single counter array) in primecount both runs faster and uses fewer instructions. However, for extremely large input values—for example, $x \geq 10^{28}$—using three counter levels is likely to result in fewer executed instructions, as the difference in instruction count gradually shifts in favor of the three-level approach with increasing input size. Below is an example C++ implementation of the improved alternative counting method which supports multiple counter levels.

```
// Count 1 bits inside [0, stop]
uint64_t Sieve::count(uint64_t stop)
{
    uint64_t start = prev_stop_ + 1;
    uint64_t prev_start = start;
    prev_stop_ = stop;

    // Each iteration corresponds to one counter level
    for (Counter& counter : counters_)
    {
        // To support resuming from the previous stop number,
        // we have to update the current counter level's
        // values if the start number has been increased
        // in any of the previous levels.
        if (start > prev_start)
        {
            counter.start = start;
            counter.sum = count_;
        }

        // Sum counts until the remaining distance becomes
        // < segment_size^((levels - level) / levels).
        // This uses at most segment_size^(1/levels) iterations.
        while (counter.start + counter.dist <= stop)
        {
            counter.sum += counter[counter.start >> counter.log2_dist];
            counter.start += counter.dist;
            start = counter.start;
            count_ = counter.sum;
        }
    }

    // Here the remaining distance is very small i.e.
    // (stop - start) < segment_size^(1/levels), hence we
    // simply count the remaining number of unsieved elements
    // by linearly iterating over the sieve array.
    count_ += count(start, stop);
    return count_;
}
```

Even though using more than two counter levels does not seem particularly useful from a practical point of view, it is very interesting from a theoretical point of view. If we used $O(\log z)$ counter levels, then the worst-case runtime complexity for counting the number of unsieved elements for a single special leaf would be $O(\log z \times segment\ size^{\frac{1}{\log z}})$, which can be simplified to $O(\log z \times e)$ and which is the same number of operations as the original algorithm with the binary indexed tree, which uses $O(\log z)$ operations. This means that using $O(\log z)$ counter levels, our alternative algorithm has the same runtime complexity as the original algorithm with the binary indexed tree.

This naturally leads to the following question: is it possible to use fewer than $O(\log z)$ counter levels and thereby improve the runtime complexity of the hard special leaf algorithm? For further details, see the Runtime complexity section.

8 Batch counting

After having removed the i-th prime and its multiples from the sieve array in the hard special leaf algorithm, we proceed to the counting part of the algorithm. For each hard leaf, that is composed of the $(i+1)$-th prime and another larger prime (or square-free number) and that is located within the current segment, we have to count the number of unsieved elements in the sieve array $\leq$ leaf. When iterating over these hard leaves their positions in the sieve array steadily increase. This property can be exploited, i.e. instead of counting the number of unsieved elements from the beginning of the sieve array for each leaf, we resume counting from the last sieve array index of the previous hard leaf. This technique is referred to as batch counting, as it computes the number of unsieved elements for many consecutive leaves by iterating over the sieve array only once.

Nowadays, most open source implementations of the combinatorial prime counting algorithms use batch counting. The earliest implementation that used batch counting that the author is aware of is from Christian Bau and dates back to 2003. But so far there have been no publications that analyze its runtime complexity implications and the author has also not yet been able to figure out by how much it improves the runtime complexity of the hard special leaf algorithm. The alternative counting methods that are presented in this paper have batch counting built-in. When turning off batch counting in the primecount C/C++ library [6] its performance deteriorates by more than 20x when computing the hard special leaves $\leq 10^{17}$. This shows that batch counting is very important for performance. It is likely that the use of batch counting enables using even fewer counter levels and thereby further improves the runtime complexity of the hard special leaf algorithm. The author has run extensive benchmarks up $x = 10^{26}$ using primecount and found that in practice, using only two counter levels provides the best performance. Thus the author's benchmarks seem to confirm that it is possible to use fewer than $O(\log z / \log \log x)$ levels of counters using batch counting. But we still assume, that using a constant number of counter levels deteriorates the runtime complexity of the algorithm.

9 Runtime complexity

When using $O(\log z)$ counter levels, the alternative algorithm requires $O(z \log z)$ operations, matching the runtime complexity of the original algorithm based on the binary indexed tree. For further details, see the Multiple Levels of Counters section. This naturally raises the question of whether the number of counter levels can be reduced below $O(\log z)$, thereby improving the overall runtime complexity of the hard special leaves algorithm.

Tomás Oliveira e Silva's paper [5] provides the following runtime complexities for the computation of the hard special leaves in the Deléglise-Rivat algorithm: sieving uses $O(z \log z)$ operations, the number of hard special leaves is $O(z/(\log^2 x \times \log \alpha))$ and for each leaf it takes $O(\log z)$ operations to count the number of unsieved elements. This means that the original algorithm is not perfectly balanced; sieving is slightly more expensive than counting. Using the improved alternative algorithm, it is possible to achieve perfect balancing by using fewer than $O(\log z)$ levels of counters. If the number of counter levels is decreased, sieving becomes more efficient but on the other hand

counting becomes more expensive. The maximum average number of allowed counting operations per leaf that do not deteriorate the runtime complexity of the algorithm is slightly larger than $O(\log^2 x)$. This bound can be achieved by using $O(\log z / \log\log x)$ levels of counters. If we set the number of counter levels $l = \log z / \log\log x$, then the number of count operations per leaf becomes $O(l \times \sqrt[l]{z})$ which is less than $O(\log^2 x)$ since:

$l \times \sqrt[l]{z} < \log^2 x$
$\Leftrightarrow \log z / \log\log x \times z^{\log\log x / \log z} < \log^2 x$
$\Leftrightarrow \log z / \log\log x \times \sqrt[\log z]{z}^{\log\log x} < \log^2 x$
$\Leftrightarrow \log z / \log\log x \times e^{\log\log x} < \log^2 x$
$\Leftrightarrow \log z / \log\log x \times \log x < \log^2 x$

Hence, by using $O(\log z / \log\log x)$ levels of counters we improve the balancing of sieve and count operations and reduce the runtime complexity of the hard special leaf algorithm by a factor of $O(\log\log x)$ to $O(z \log z / \log\log x)$ operations. In the original Deléglise-Rivat paper [3] the number of hard special leaves is indicated as $O(\pi(\sqrt[4]{x}) \times y)$, which is significantly smaller than in Tomás Oliveira's version of the algorithm [5]. This lower number of hard special leaves makes it possible to use even fewer counter levels and further improve the runtime complexity of the algorithm, here we can use only $O(\log\log z)$ counter levels which reduces the runtime complexity of the algorithm to $O(z \log\log z)$ operations.

10 Open questions

The alternative counting methods presented in this paper have batch counting built-in, but as mentioned in the Batch counting section, the author does not know whether the use of batch counting enables using fewer than $O(\log z / \log\log x)$ counter levels and thereby further improves the runtime complexity of the hard special leaf algorithm. Ideally, we want to use only $O(\log\log z)$ counter levels in which case the runtime complexity of the hard special leaf algorithm would be $O(z \log\log z)$ operations (provided that the use of $O(\log\log z)$ counter levels does not deteriorate the runtime complexity of the algorithm).

The distribution of the hard special leaves is highly skewed, most leaves are located at the beginning of the sieving algorithm and as one sieves up the leaves become sparser and the distance between consecutive leaves increases. In the Runtime complexity section the author suggested using a fixed number of counter levels for the entire computation. But this is not optimal; one could likely further improve the balancing of sieve and count operations by adjusting the number of counter levels for each segment. However, the author does not know how one would calculate the optimal number of counter levels for individual segments or how this would affect the algorithm's runtime complexity.

11 Appendix

- In the original Deléglise-Rivat paper [3] its authors indicate that the use of a binary indexed tree deteriorates the sieving part of the hard special leaf algorithm by a factor of $O(\log z)$ to $O(z \times \log z \times \log\log z)$ operations. Tomás Oliveira e Silva in [5] rightfully points out that this is incorrect and that it only deteriorates the sieving part of the algorithm by a factor of $O(\log z / \log\log z)$. This is because we don't need to need to perform $O(\log z)$ binary indexed

tree updates for each elementary sieve operation, of which there are $O(z \log \log z)$. But instead we only need to perform $O(\log z)$ binary indexed tree updates whenever an element is crossed off for the first time in the sieve array. When we sieve up to z, there are at most z elements that can be crossed off for the first time, therefore the runtime complexity of the sieving part of the hard special leaf algorithm with a binary indexed tree is $O(z \log z)$ operations.

Our new alternative algorithm also relies on the above subtlety to improve the runtime complexity. When using multiple counter levels, the related counter arrays should only be updated whenever an element is crossed off for the first time in the sieve array. However, when using a small constant number of counter levels, for example ≤ 3, it may be advantageous to always update the counter array(s) when an element is crossed off in the sieve array. This reduces the branch mispredictions and can significantly improve performance. See the first code section in Improved alternative counting method for how to implement this.

- When using a bit sieve array it is advantages to count the number of unsieved elements in the sieve array using the POPCNT instruction. The use of the POPCNT instruction allows counting many unsieved elements (1-bits) using a single instruction. In primecount each POPCNT instruction counts the number of unsieved elements within the next 8 bytes of the sieve array and these 8 bytes correspond to an interval of size $8 \times 30 = 240$. When using multiple counter levels, it is important for performance that on average the same number of count operations is executed on each level. However, using the POPCNT instruction dramatically reduces the number of count operations in the last level and hence causes a significant imbalance. To fix this imbalance one can multiply the counter distance for each level by $POPCNT\_distance^{\,level/levels}$. In primecount the POPCNT distance is 240. If primecount were modified to use 3 counter levels, one would multiply the counter distance of the 1st level by $\sqrt[3]{240}$ and the counter distance of the 2nd level by $240^{\frac{2}{3}}$.

References

[1] J. C. Lagarias, V. S. Miller, and A. M. Odlyzko, *Computing $\pi(x)$: The Meissel–Lehmer method*, Math. Comp., 44 (1985), pp. 537–560.

[2] P. M. Fenwick, *A new data structure for cumulative frequency tables*, Software: Practice and Experience 24, 3 (1994), pp. 327–336.

[3] M. Deléglise and J. Rivat, *Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko Method*, Math. Comp., 65 (1996), pp. 235–245.

[4] X. Gourdon, *Computation of $\pi(x)$: Improvements to the Meissel, Lehmer, Lagarias, Miller, Odlyzko, Deléglise and Rivat method*, Feb. 15, 2001.

[5] T. Oliveira e Silva, *Computing $\pi(x)$: The combinatorial method*, Revista do DETUA, 4(6) (2006), pp. 759–768.

[6] K. Walisch, *primecount: Fast C/C++ prime counting function library*, Version 7.19, 2025. Available at: https://github.com/kimwalisch/primecount