Classic Linear Sieves Revisited

– An Exercise in Sieve Optimization –

Mircea GHIDARCEA * **Decebal POPESCU** **

* *National University of Science and Technology POLITEHNICA Bucharest – Computer Science Department (e-mail: mircea.ghidarcea@stud.acs.upb.ro).*
** *National University of Science and Technology POLITEHNICA Bucharest – Computer Science Department (e-mail: decebal.popescu@upb.ro) Splaiul Independentei 313, Romania*

Abstract: Recent IT research on prime numbers has focused on generating large primes for cryptography, neglecting the advancement of **systematic generation of prime numbers in sequence** using techniques like **sieving**. Despite being relegated to educational contexts, sieving techniques hold untapped potential. Much of the existing work remains obscure to contemporary researchers, leaving open numerous opportunities to enhance the field. This article aims to highlight the relevance of old sieving algorithms, demonstrating that modern approaches and creativity can yield significant progress.

Keywords: Prime numbers sieving, Prime number generation, Algorithms, Algorithm optimization, Software performance optimization.

1. INTRODUCTION

The first occurrence of a primes sieving technique (Nicomachus, 1926) is due to **Eratosthenes** (Sieve of Eratosthenes – **SoE**), and other primes sieving algorithms were later devised by mathematicians like **Euler** (Sieve of Euler – **SoE+**) or, more recently, **Sundaram** (Sieve of Sundaram – **SoS**).

Given the importance of prime numbers, several articles were initially written on the topic of **systematic generation of prime numbers in sequence** – the first computer algorithm for a sieve generating prime numbers was published as early as 1961 by T.C.Wood (Wood, 1961). Many attempts were made to optimize SoE as it is until the late 70s, when (Mairson, 1977), (Pratt, 1977) and (Gries and Misra, 1978) introduced the first linear sieves. Paul Pritchard made several important advances in the 80s, including the beautiful, sub-linear Sieve of Pritchard – **SoP** (Pritchard, 1981). The list of classic linear sieves is closed by (Bengelloun, 1986) with another linear incremental sieve. Afterwards, the field was practically abandoned and no real progress was made in the domain of sieving algorithms, with the notable exception of Sieve of Atkin – **SoA** (Atkin and Bernstein, 2003).

This article wishes to take over from the work of H.G.Mairson, R.Gale / V.Pratt, D.Gries / J.Misra and S.Bengelloun, the original linear sieves authors, and carry their algorithms further, several steps closer to their true potential. The presentation includes a step by step demonstration for Mairson, to get the gist of a basic methodology that can be applied to the problem of sieve optimization. It then follows with optimized variants for all the other classic linear sieves and closes with an optimized variant of SoP.

***NOTE 1**: As performance is the most important attribute of a sieve, C/C++ is a very suitable language for this purpose, and it was selected by us for all the experiments described here. The accompanying code used in this review, including all the algorithms described here and all our experiments to optimize them, can be found on* **GitHub** *at* `https://github.com/mirceag70/Linear_Sieved_Revisited`.

***NOTE 2**: As a general observation, for a better standardisation of results, all the components of the generation process must be included in the timings that are to be compared, including here any data preparation that occurs before sieving (like root primes generation) or after sieving (like, really, effectively obtaining the value of all prime numbers and getting those values in order)* [1]

1.1 Basic Notions

The whole sieving domain was ignited by Eratosthenes' sieve — **SoE** remains the most popular technique and, as explained in (Crandal and Pomerance, 2005), the complexity of SoE is:

$$\mathbf{C}(SoE) = \mathbf{O}(N \ln(\ln(N))) \tag{1}$$

The Sieve of Euler (**SoE+**) is an improved version of SoE: While SoE computes many more multiples than required, SoE+ will compute exactly as many multiples as are necessary to eliminate all composites. As this is the basis for most of the linear sieve algorithms, Algorithm 1 describes the SoE+ technique.

Ignoring the overhead, there is exactly one main operation per number (either remove it from list A or move it to list B) resulting that this technique is linear:

[1] Usually an optimizing compiler will discard futile code, so very often it is not sufficient to simply compute the number in code without using it – one must assure that the number is really computed in the benchmarking process.

$$\mathbf{C}(SoE+) = \mathbf{O}(N) \tag{2}$$

Algorithm 1 Basic SoE+ algorithm

(1) Initialize a list A with all the numbers from 2 up to the maximum value N we are looking for;
(2) Create an empty list B for the primes;
(3) Mark down for removal from list A all the numbers equal to the product of the first number in the list multiplied with each number remaining in the list, including itself;
(4) Remove from list A all marked numbers;
(5) Move the first number from list A in list B;
(6) Repeat from step 3 until list A is empty;
(7) List B contains now all the primes lower than N.

For large numbers x, we can approximate (Crandal and Pomerance, 2005) the number of primes lower than x — we call this number $\pi(x)$ — with:

$$\pi(x) \approxeq \frac{x}{\ln(x)} \tag{3}$$

In the early years of computing, RAM was a very precious resource — its cost and scarcity drove computer scientist to sacrifice performance over space complexity, hence the algorithms from Wood, Chartres and Singleton — basically all have $\mathbf{O}(\pi(\sqrt{n}))$ space complexity. The fastest one was **Singleton's 357** (Singleton, 1969), which has quite an impressive performance for such an old algorithm and remains the fastest classic sieving algorithm. These old works introduced some very important ideas that will shape the future of prime sieving and will be used here, too:

- Skip over some classes of candidates and multiples of primes to eliminate from start a whole bunch of useless operations; first, using *step* = 2 inside loops to skip even numbers; then skipping over multiples of 2 and 3 by cleverly alternating between step 2 and 4, thus hitting only candidates and multiples in the form $n = 6k \pm 1$ (an earlier form of *static wheel*);
- Processing only the primes up to $\sqrt{n}$ — we shall call these first $\pi(\sqrt{n})$ primes **the root primes**;
- For each p of the *root primes*, we start sweeping at p^2;
- The *bool vector* technique to represent the prime/composite status of a sequence of numbers at the level of individual bytes, then going forward with the idea to represent the status at the bit level.

2. THE LOW HANGING FRUIT

2.1 Original Mairson

The Sieve of Mairson (**SoM**) is the first published attempt into linear sieving algorithms (Mairson, 1977). The technique is very similar to Euler's, as represented in Table 7. It is not clear if Mairson knew about Euler's sieve — in any case, it is not mentioned in the paper.

SoM uses an intricate data structure based on a triple vector representation of a double linked list and manages to compute each composite only once. Algorithm 2 is quite an accurate rendition of the algorithm as devised by Mairson.

Algorithm 2 Mairson's sieve

- To be faithful, the implementation here retains the 1-based indexes

```
void Mairson(int Nmax, int DELETE[],
             int rlink[], int llink[]) {
    auto crossoff = [&](int i)
        { rlink[llink[i]] = rlink[i];
          llink[rlink[i]] = llink[i]; };

    for (int i = 1; i <= Nmax; i++)
        { rlink[i] = i + 1; llink[i] = i - 1;
          DELETE[i] = 0; }
    rlink[Nmax] = llink[1] = 0;

    int prime = 2;
    int nsqrt = floor(sqrt(Nmax));
    while (prime <= nsqrt) {
        int factor = prime, pointer = 0;
        int k = prime * factor;
        while (k <= Nmax)
        {
            DELETE[++pointer] = k;
            factor = rlink[factor];
            k = prime * factor;
        }
        for (int i = 1; i <= pointer; i++)
            crossoff(DELETE[i]);
        prime = rlink[prime];
    }

    int numPrimes = 0;
    for(int i = rlink[1]; i > 0; i = rlink[i])
        { AddPrime(i); numPrimes++; }
}
```

***NOTE 3**: All algorithms presented in this text are somewhat streamlined for brevity: check the accompanying code for full implementation and all the details.*

Table 1 presents the timings for SoM compared with the other classic linear algorithms (in their basic versions) – **357** is included as a baseline. These initial results were not only worse than **357**, but even worse than a basic SoE, so unfortunately everybody took these algorithms only as theoretical endeavours and — except one isolated attempt for Bengelloun in (Pritchard, 1994) — practically nobody really tried to push them further.

***NOTE 4**: All the timings in this paper were measured on the same desktop computer with Intel Core i9-9900KF CPU overclocked at 4.9MHz all cores.*

Mairson claims an excellent time complexity for his algorithm, but in practice the memory access complexity is quite high and the performance does not shine, as seen in Table 1. The basic operations primary identifying the composites may have $\mathbf{O}(N)$, but when added all the overhead necessary to maintain the relatively complex data structure things get rapidly out of control. It is clear that both problems regarding memory and operational complexity have to be addressed.

Because Mairson keeps absolute indexes in *llink, rlink* and *DELETE* (see Algorithm 2), the element size of these vectors has to accommodate the maximum number of elements, i.e. the value N_{max} up to which we want to compute the primes, hence the necessity to use a 64bit (8bytes) element to get beyond 4.3 billion. For N_{max} = 10 billion the memory requirements would be $3 \times 8 \times$ 10GB= 240GB, which is ridiculous. Even if we keep to 32bit (4bytes) per element, we still need $(3 \times 4 \times 2^{32})$ bytes, which is more than 50GB to get all primes up to 2^{32} — a 32GB workstation will seriously swap, while a 16GB

n	$\pi(10^n)$	Algorithms				
		Mairson	Gale/Pratt	Gries/Misra	Bengelloun	S.357
5	9'592	1	1	1	1	1
6	78'498	8	11	9	9	2
7	664'579	189	95	207	110	19
8	5'761'455	1'768	908	2'204	1'336	174
9	50'847'534	22'351	14'354	24,143	13'761	1'707
10	455'052'511	-	-	-	-	18'415

Table 1. Linear sieves timings [ms] for generating primes up to 10^n

PC will have problems as low as 1 billion. Moreover, the performance is lower in 64bit, as seen in Table 2.

n	$\pi(10^n)$	Algorithms	
		Mairson 32b	Mairson 64b
5	9'592	1	1
6	78'498	8	17
7	664'579	189	241
8	5'761'455	1'768	2'556
9	50'847'534	22'351	29,107

Table 2. Mairson timings [ms], 32b vs 64b

First things first, one needs to address the memory footprint to get a practical algorithm.

2.2 Better underlying data structure

One observation here is that Mairson uses three vectors, but *llink* is only necessary for *crossof* — if we eliminate *crossof* we eliminate a third of the memory footprint. Generally speaking, double linked lists are required only when the list must be traveled both ways: we don't have to travel back in the list for striking composites out — we need *crossof* only because composites are removed at the end of the cycle via their pointer kept in *DELETE*.

(Gries and Misra, 1978) includes also a discussion regarding the underlying data structure, where it is shown how one can eliminate *llink* vector with some cost in operations or how to use a smaller number of bits for *link* elements, that would result in further memory savings — because of the great similarity between the two, these optimizations can be applied very well to Mairson, too.

Also preoccupied by the memory footprint, (Misra, 1981) proposes to compute and eliminate multiples in reverse order, thus eliminating the need for the third *DELETE* vector (Pritchard (1981) will later use the idea for SoP), but finding the *pointer* for that greatest value and traversing down from there has a negative impact on processing time, which is what we value the most.

Alternatively, one can mark the composite for cancellation, but not cancel it immediately, nor at the end of the inner loop: do the cancellation only at the next encounter, assuring that a) travels always in only one direction, b) does not operate unnecessary cancellations, and c) is iterating the list only for the necessary process of marking composites and not just for cancellation.

Using this technique we do not need to get the address of the node to be canceled out of sync (we are already there), so we do not have to keep in *DELETE* the whole address, but a simple flag at the index of the node to be canceled. Here is the technique modified accordingly:

Algorithm 3 Mairson — single linked

- The most important alterations are in the inner loop

```
void SglLnk(int Nmax, int rlink[], bool DELETE[]) {
    int i;
    for (i = 1; i <= Nmax; i++)
    { rlink[i] = i + 1; DELETE[i] = false; }

    int nsqrt = floor(sqrt(Nmax));
    for (int prime = 2; prime <= nsqrt;
                    prime = rlink[prime]) {
        int factor = prime;
        for (int k = prime * factor; k <= Nmax;
                                k = prime * factor)
        {
            DELETE[k] = true;
            int newfactor = rlink[factor];
            if (DELETE[newfactor])      // unlink
                rlink[factor] = rlink[newfactor];
            factor = newfactor;
        }
    }

    int numPrimes = 0;
    for (i = 2; i <= Nmax; i++)
        if (not DELETE[i])
            { AddPrime(i); numPrimes++; }
}
```

Up to 2^{32} we need now only $(4+1) \times 4.3\text{GB} = 21.5\text{GB}$, which is quite practical. For $N_{max} = 10$ billion the memory requirements are down to $(8+1) \times 10\text{GB} = 90\text{GB}$, which is feasible, and we have more or less five times improvement in performance for 64bit — see the timings in Table 3.

2.3 Incremental steps

Next step would be to implement an incremental approach for the indexes, as suggested by Misra — instead of storing the absolute values, store the difference from the previous one. The maximum value we have to accommodate is the maximum gap between two consecutive prime numbers [2]: for primes lower than 2^{64} the maximum gap is lower than 1600, which is less than 2^{11}, so we can very well use a 16bit integer to store these values (even 12 bits would be enough). Here is the code modified accordingly:

Algorithm 4 Mairson — delta link

- The most important alterations are again in the inner loop

```
int Optim1(const int Nmax, int link[], bool mark[]) {
    [...]
    int nsqrt = (int)floor(sqrt(Nmax));
    for (p = 2; p <= nsqrt; p += link[p])
    {
        numPrimes++; AddPrime(p);

        int f = p;
        for (int n = p * f; n <= Nmax; n = p * f)
        {
```

[2] which can be consulted for example at Chris Caldwell's **PrimePages** https://t5k.org/notes/GapsTable.html

n	$\pi(10^n)$	Mairson iterations						
		Standard	One link	Gap	Step2	Step23	8bit	1bit
5	9'592	1	1	1	1	1	1	1
6	78'498	17	4	3	2	1	1	1
7	664'579	241	41	35	16	12	12	12
8	5'761'455	2'556	476	393	208	154	146	120
9	50'847'534	29'107	4,927	3'934	2'192	1'669	1'521	1'480
10	455'052'511	-	-	55'905	23'508	17'036	15'957	16'046
11	4'118'054'813	-	-	-	-	-	*138'243	166'808

Table 3. Timings (ms) for the gradual optimization of Mairson (64bit)
**on a workstation with i9-12900k*

N_{max}	word	Mairson iterations						
		Standard	One link	Gap	Step2	Step23	8bit	1bit
2^{32}	32 bit	50	21.5	13	6.5	4.3	1.6	0.31
10^{10}	64 bit	240	90	30	15	10	4	0.72

Table 4. Memory footprint (GB) for the gradual optimization of Mairson

```
            mark[n] = true;
            int fnew = f + link[f];
            if (mark[fnew])     // unlink
                link[f] += link[fnew];
            f = fnew;
        }
    }
    [...]
}
```

Up to 2^{32} we need $(2+1) \times 4.3\text{GB} = 13\text{GB}$, which is now workable on a 16GB computer. For 10 billion the memory requirements are down to $(2+1) \times 10\text{GB} = 30\text{GB}$, which is quite practical; and we have another decent improvement in performance — see the timings in Table 3.

At this point one would be tempted to eliminate completely the *mark* vector and codify the cancellation info in *link*, as we do not need more than 12 bits there. We tried this in two variants: a) codifying the info in the most significant bit and b) using the negative/positive signal (where negative increments are to be cancelled); both resulted in 10-15% performance loss, so we abandoned this idea and kept the parallel *mark* vector.

2.4 Pick up the pace

The following logical step is to eliminate from the start all even values, thus halving the space:

Algorithm 5 Mairson — exclude evens

- All absolute values (p, f, n) can be computed from indexes and eliminated — we tried it and there is no sensible improvement in performance (*a shout out to the modern optimizing compilers*), so we kept this version for clarity

```
int Optim4(int Nmax, int link[], bool mark[]) {
    [...]
    unsigned numPrimes = 1; AddPrime(2); //account for 2
    int i_p, p, i_f, f, i_n, n,;
    int i_p_max = ((int)floor(sqrt(Nmax)) - 3) / 2;
    for (i_p = 0, p = 3; i_p <= i_p_max;
            i_p += link[i_p], p = 2 * i_p + 3) {
        numPrimes++; AddPrime(p);
        i_f = i_p;
        f = p;  // 2 * i_p + 3
        n = p * f; // 4 * i_p * i_p + 12 * i_p + 9
        while (n <= Nmax) {
            i_n = (n - 3) / 2; mark[i_n] = true;
            int i_fnew = i_f + link[i_f];
            if (mark[i_fnew])     // unlink
                link[i_f] += link[i_fnew];
            i_f = i_fnew;
            f = 3 + 2 * i_f;
            n = p * f; //(2*i_p+3)*(2*i_f+3)
        }
    }
    [...]
}
```

Up to 2^{32} we need $(2+1)/2 \times 4.3\text{GB} = 6.5\text{GB}$, which is now workable on a 8GB computer. For $N_{max} = 10$ billion the memory requirements are now down to $(2+1) \times 5\text{GB} = 15\text{GB}$, which is really good; both time and space are halved again — see the timings in Table 3.

Of course, we should skip also multiples of 3 and implement the $6k \pm 1$ pattern, for which we crafted new formulas to avoid costly pipeline flushes:

Algorithm 6 Mairson — $6k \pm 1$

- All the magic is in lines 2-5

```
void Optim8(const int Nmax, int link[], bool mark[]) {
    auto idx2no = [](int idx)
        { return 3 * idx + 5 - (idx & 1); };
    auto no2idx = [](int no)
        { return no / 3 - 1; };

    unsigned numPrimes = 2; //account for 2 and 3
    AddPrime(2); AddPrime(3);

    int i, i_max = no2idx(Nmax);
    for (i = 0; i <= i_max; i++)
        { mark[i] = false; link[i] = 1; }

    int p, i_p, f, i_f, n, i_n;
    int i_p_max = no2idx((int)floor(sqrt(Nmax)));
    for (i_f = i_p = 0; i_p <= i_p_max;
            i_f = i_p += link[i_p]) {
        f = p = idx2no(i_p); numPrimes++; AddPrime(p);
        for (n = f * p; n <= Nmax; n = f * p)
        {
            i_n = no2idx(n);
            mark[i_n] = true;

            int i_fnew = i_f + link[i_f];
            if (mark[i_fnew])     // unlink it
                link[i_f] += link[i_fnew];
            i_f = i_fnew;
            f = idx2no(i_f);
        }
    }

    for (i = i_p; i <= i_max; i++)
        if (!mark[i])
            if ((p = idx2no(i)) <= Nmax)
                { numPrimes++; AddPrime(p); }
            else break;
}
```

Up to 2^{32} we need $(2+1)/3 \times 4.3\text{GB} = 4.3\text{GB}$. For $N_{max} = 10$ billion the memory requirements are now down to $(2+1)/3 \times 10\text{GB} = 10\text{GB}$, which is excellent. The performance also improves a little, closing in to **357** — see the timings in Table 3.

2.5 Final touches

Analyzing the behaviour of the program, it became evident that the values for *link* start to be modified only at the second run, for $p = 7$ ($p = 5$ may mark a lot of values to be cancelled, but the marking is done in the second vector and the cancellation is effectively performed later). Thus, f can only go as far as $N_{max}/7$ when modifying *link* (cancellation itself will not be performed further, neither the alteration of *link*).

Moreover, the values in *link* are not actually the gaps between primes, but the gap until the next entry of the next prime in the vector: while the algorithm implements the $6k \pm 1$ pattern, this value is 2 to 4 times lower then the real prime gap value itself — as the gap became larger it will encounter more 2/4 sequences, so for large gaps it will be around one third of the real gap. And each loop will not mark two consecutive gaps — many loops are needed to mark a larger gap. Each loop k will only mark gaps lower than N_{max}/p_k, the next loop will stop at N_{max}/p_{k+1}, so the gaps corresponding to primes between N_{max}/p_{k+1} and N_{max}/p_k will never be completely marked out. Actual gaps will correspond to primes much, much lower than N_{max}. The maximum value represented in 8 bits is 256, corresponding to a real gap of $3 \times 256 = 768$, which appear for a prime larger than 19 billion. And this requires 256 vector gaps to be cancelled: it is more than safe to assume that many loops are required, going far over $p = 53$, which puts us safe over $19 \times 53 = 1007$ billion: we can safely use an 8bit *link* at least up to 1 trillion, which is more than sufficient for any practical goal within the reach of Mairson.

N_{max}	max link	real gap	prime	N_{max}/prime
10^5	11	34	1'327	75
10^6	12	36	9'551	105
10^7	24	72	31'397	319
10^8	29	86	155'921	641
10^9	44	132	1'357'201	737
10^{10}	51	154	4'652'353	2'149

Table 5. Gap related values

The first real maximum values encountered are compiled in Table 5, and these values can only grow larger with N_{max}. It is visible that the real ratio between N_{max} and the prime with the greatest gap is actually well over 2000 for big limits, so N_{max} will be over thousands of billions when the 8bit value will overflow — by that point, memory consumption will have made this approach impractical anyway.

Another consequence here is that the length of *link* vector does not have to be $N_{max}/3$ — $N_{max}/15$ is sufficient, and even $N_{max}/21$ if we hard-code the first inner loop:

Algorithm 7 Mairson — 8bit

- With $p = 5$ hard-coded

```
void Optim9(const int Nmax, int next[], bool mark[]) {
    auto idx2no = [](int idx)
        { return 3 * idx + 5 - (idx & 1); };
    auto no2idx = [](int no)
        { return no / 3 - 1; };

    unsigned numPrimes = 3; //account for 2 and 3 and 5
    AddPrime(2); AddPrime(3); AddPrime(5);

    int i, i_max = no2idx(Nmax);
    for (i = 0; i <= Nmax / 21; i++)
        { next[i] = 1; mark[i] = false; }
    for (; i <= i_max; i++) { mark[i] = false; }

    int p, i_p, f, i_f, n, i_n;
    //hard-code 5
    f = p = 5; n = f * p;
    for (uint8_t step = 2; n <= Nmax; step = 6 - step) {
        i_n = no2idx(n);
        mark[i_n] = true;
        f += step;
        n = f * p;                                      }
    //start from 7
    int i_p_max = no2idx((int)floor(sqrt(Nmax)));
    for (i_f = i_p = 1; i_p <= i_p_max;
         i_f = (i_p += next[i_p]))                      {
        f = p = idx2no(i_p); numPrimes++; AddPrime(p);
        for (n = f * p; n <= Nmax; n = f * p)           {
            i_n = no2idx(n); mark[i_n] = true;

            int i_fnew = i_f + next[i_f];
            if (mark[i_fnew])     // unlink it
                next[i_f] += next[i_fnew];
            i_f = i_fnew; f = idx2no(i_f);            }}
    for (; i_p <= i_max; i_p++)
        if (!mark[i_p])
            if ((p = idx2no(i_p)) <= Nmax)
                { numPrimes++; AddPrime(p); }
            else break;
}
```

For $N_{max} = 10$ billion the memory requirements are now down to $(\frac{1}{3\times5} + \frac{1}{3}) \times 10\text{GB} = \mathbf{4GB}$, which is outstanding. The performance is now marginally better than **357** — see the timings in Table 3. On a machine with 64GB RAM and a last generation, high IPC CPU it will take less than 2 minutes to get all primes up to 100 billion.

Of course, one could stop unlinking the composites that will not be used in the next iteration, at least for $p = 7$ — this may save some operations, but, more important, it will reduce link size from $\frac{1}{3\times7}$ to $\frac{1}{3\times11}$. Also, for good measure, we can use one bit representation instead of a byte in *mark* to further reduce memory consumption, as in Algorithm 8:

Algorithm 8 Mairson — 1bit

- With $p = 5$ and 7 hard-coded

```
const byte BIT_MASK[] = { 1, 2, 4, 8, 16, 32, 64, 128 };
void Optim10(const int Nmax, byte next[], byte mark[]){
    auto SetBit = [&](int bitidx)
    {   int idx = bitidx / 8; byte bit = bitidx % 8;
        mark[idx] |= BIT_MASK[bit];             };
    auto GetBit = [&](int bitidx)
    {   int idx = bitidx / 8; byte bit = bitidx % 8;
        return (mark[idx] & BIT_MASK[bit]);     };

    unsigned numPrimes = 4; //account for 2  3  5 and 7
    AddPrime(2); AddPrime(3); AddPrime(5); AddPrime(7);

    int i, p, i_p, f, i_f, n, i_n; uint8_t step;
    const int i_max = Nmax / 24, i_max_7 = Nmax / 33;
    for (i = 0; i <= i_max; i++) { mark[i] = false; }
    for (i = 0; i <= i_max_7; i++) { next[i] = 1; };
```

```
    f = p = 5; n = f * p;   //hard-code 5
    for (step = 2; n <= Nmax; step = 6 - step)  {
        i_n = no2idx(n); SetBit(i_n);
        f += step; n = f * p;                   }
    f = p = 7; n = f * p;  //hard-code 7
    for (i_f = 1; i_f <= i_max_7; n = f * p)    {
        i_n = no2idx(n); SetBit(i_n);
        int i_fnew = i_f + next[i_f];
        if (GetBit(i_fnew))     // we need to unlink it
            next[i_f] += next[i_fnew];
        i_f = i_fnew; f = idx2no(i_f);          }
    for (; n <= Nmax; n = f * p)                {
        i_n = no2idx(n); SetBit(i_n);
        i_f++; f = idx2no(i_f);                 }
    //continue from 11
    const int i_p_max = 1 + no2idx(floor(sqrt(Nmax)));
    for (i_f = i_p = 2; i_p <= i_p_max;
            i_f = (i_p += next[i_p]))           {
        f = p = idx2no(i_p);
        numPrimes++; AddPrime(p);

        for (n = f * p; n <= Nmax; n = f * p)   {
            i_n = no2idx(n); SetBit(i_n);

            int i_fnew = i_f + next[i_f];
            if (GetBit(i_fnew))     // unlink it
                next[i_f] += next[i_fnew];
            i_f = i_fnew; f = idx2no(i_f);      }}
    //count rest
    p = idx2no(i_p); i = (p - 5) / 24;
    int pbase = (i * 24) + 5;
    uint8_t b = (uint8_t)(p - pbase);
    switch (b)                                  {
     case  0: b = 0; step = 2; break;
     case  2: b = 1; step = 4; break;
     case  6: b = 2; step = 2; break;
     [...]
     default: assert(false);                    }
    uint8_t flags = mark[i];
    assert(!(flags & BIT_MASK[b]));
    for (; b < 8; b++)                          {
        if (!(flags & BIT_MASK[b]))
            { numPrimes++; AddPrime(p); }
        p += step; step = 6 - step;             }
    for (i++; i < i_max-1; i++)                 {
        flags = mark[i];
        if (!(flags & BIT_MASK[0]))
            numPrimes++; AddPrime(p); }
        if (!(flags & BIT_MASK[1]))
            { numPrimes++; AddPrime(p + 2); }
        if (!(flags & BIT_MASK[2]))
            { numPrimes++; AddPrime(p + 6); }
        [...]
        p += 24;                                }
    while (p <= Nmax)                           {
        flags = mark[i++];
        for (b = 0; b < 8; b++)                 {
            if (p > Nmax) break;
            if (!(flags & BIT_MASK[b]))
                { numPrimes++; AddPrime(p); }
            p += step; step = 6 - step;         }}}
```

For $N_{max} = 10$ billion the memory requirements are now down to $(\frac{1}{3\times11} + \frac{1}{3\times8}) \times 10\text{GB} = \mathbf{720MB}$, which is impressive, and the performance does not degrades, due to even better locality: SoM can now compute primes up to **100 billion** with less than 8GB RAM — see the timings in Table 3.

The performance gain is remarkable, twenty times faster than original. Even more for the footprint, where we have more than 160 times less memory required for 32bit. These results only show what can be achieved working on an inspired algorithm. Surely, much more can be squeezed out of SoM with appropriate dedication: for example, one can jump from $6k \pm 1$ pattern to $30k+$ — this will firstly require finding a simple enough formula to replace *idx2no* and *no2idx* in the code above, then make some other little alterations. Memory will get down to $(\frac{1}{3\times13} + \frac{1}{30}) \times 10\text{GB} = \mathbf{590MB}$ and suddenly we can compute up to 1 trillion in 64GB, more than standard for a workstation nowadays; the performance should also get another boost, closing in to 1 second for 1 billion. Of course, if the memory is the only concern one can use a more complicated function with logical constructs, but there is no fun in such approach.

Here is a starting point for such a fast function generating the $30k+$ pattern (all the operations involving powers of 2 are very well optimized by the compiler; there are only two divisions that look difficult — 6 and 7 — but those are divisions by constant and, again, the optimizing compiler will transform them in fast shift/mul ops.):

```
int idx2no30k(int idx) {
    int k = idx / 8;   int i = idx % 8;
    return 4 * (i + (i / 6)) + 30 * k -
           2 * ((i / 2) + (i / 7)) + 7;
}
```

This arithmetic function is six times faster than the logical variant, but we are certain that somebody can do much better than this.

2.6 Gries/Misra

(Gries and Misra, 1978) take over from Mairson: while SoM iterates for each first number p left in the list and strikes out the products between p and every number left in the list at that point, Gries/Misra iterates for each first number p left in the list and strikes out the products between all powers of p and the second number in the list. Table 7 depicts the steps in SoM vs. Gries/Misra. Both techniques strike out each composite exactly once — hence the linear character of the algorithms — but SoM may strike out numbers that are to be used later in the same iteration, bringing more complications in data management to preserve the numbers already marked until the end of the current cycle. Moreover, Gries/Misra eliminate the need for the DELETE vector, thus lowering memory requirements. Here is the corresponding algorithm:

Algorithm 9 Gries/Misra standard linear sieve

- The skeleton is very similar to SoM

```
void GM_Standard(int Nmax, int rlink[], int llink[]) {
    for (int i = 1; i <= Nmax; i++)
        { rlink[i] = i + 1; llink[i] = i - 1; }
    rlink[Nmax] = llink[1] = 0;

    int nsqrt = ceil(sqrt(Nmax));
    for (int p = 2; p <= nsqrt; p = rlink[p])
    {
        int q = p;
        for (int pq = p * q; pq <= Nmax; pq = p * q)
            for(int x = pq; x <= Nmax; q = rlink[q])
              { crossoff(x); x *= p; }
    }

    unsigned numPrimes = 0;
    for (int i = rlink[1]; i > 0; i = rlink[i])
        { AddPrime(i); numPrimes++; }
}
```

In theory, Gries/Misra should perform marginally better, because of the same number of composites stroke out and simpler data management — in practice, because Gries/Misra has the triple nested loop with worse locality

n	$\pi(10^n)$	Gries/Misra iterations			SoE 6k
		Standard	SoM based	No link	
5	9'592	1	1	1	1
6	78'498	9	1	2	2
7	664'579	207	11	17	16
8	5'761'455	2'204	123	182	149
9	50'847'534	24'143	1,723	2'312	2'496
10	455'052'511	-	17'794	24'982	29'102
11	4'118'054'813	-	195'957	266'256	330'583
12	37'607'912'018	-	-	*2'893'693	-

Table 6. Timings (ms) for the gradual optimization of Gries/Misra (64bit).
**on a workstation with i9-12900k*

(fewer composites identified per cycle, so a lot of cache trashing), Mairson does better; see the timings in Table 1.

As Gries/Misra is so similar to Mairson, of course we can apply most of the techniques used there — yet, the process of striking out composites is significantly more hectic due to the three nested loops, thus making it very difficult to hard-code out $p = 5$ and 7 for more gain. Nevertheless, here is our optimized version:

Algorithm 10 Gries/Misra optimized linear sieve

- All SoM optimizations except 5/7 hard-coding

```
void GM_Optim3(int Nmax, uint8_t next[], uint8_t mark[]) {
    [ ... ]
    unsigned numPrimes = 2; //account for 2 and 3
    AddPrime(2); AddPrime(3);

    int p, q, pq, i, i_max = Nmax / 24;
    for (i = 0; i <= i_max; i++) { mark[i] = false; };
    for (i = 0; i <= Nmax / 15; i++) { next[i] = 1; };

    int i_q, i_p = 0, nsqrt = floor(sqrt(Nmax));
    for (p = 5; p <= nsqrt; p = idx2no(i_p))
    {
        numPrimes++; AddPrime(p);

        q = p; i_q = i_p;
        for (pq = p * q; pq <= Nmax; pq = p * q)
        {
            for (int n = pq; n <= Nmax; n *= p)
                SetBit(no2idx(n));

            int i_qnew = i_q + next[i_q];
            while (GetBit(i_qnew))      // unlink it
            {
                next[i_q] += next[i_qnew];
                i_qnew += next[i_qnew];
            }
            i_q = i_qnew; q = idx2no(i_q);
        }
        i_p += next[i_p];
    }
    [ ... ]
}
```

Gries/Misra was marginally less performant from start, so the same optimization strategy is meant to fall short, as it can be seen in Table 6 — most probable, any further optimization will apply to Mairson, too, and with a higher return. Another approach is needed to get better performance, and this should speculate on the most meaningful difference between SoM and Gries/Misra: the fact that the composites are canceled in such an order that it does not impede on further cancel operations. One idea then is to get rid of the whole *link* apparatus and keep the cancellation only in *mark* vector: this will basically result in a variant of SoE that uses that optimal Gries/Mairson pattern in marking out composites:

Algorithm 11 Gries/Misra without links

- Repeated code left out

```
void GM_Optim4(int Nmax, uint8_t mark[]) {
    [ ... ]
    unsigned numPrimes = 2; AddPrime(2); AddPrime(3);
    int i, i_max = Nmax / 24;
    for (i = 0; i <= i_max; i++) { mark[i] = false; };

    int pq, i_q, i_p = 0, nsqrt = floor(sqrt(Nmax));
    for (int p = 5; p <= nsqrt; p = idx2no(i_p))
    {
        numPrimes++; AddPrime(p);

        int q = p; i_q = i_p;
        for (pq = p * q; pq <= Nmax; pq = p * q)
        {
            for (int n = pq; n <= Nmax; n *= p)
                SetBit(no2idx(n));
            // advance to the next uncancelled
            for( i_q++; GetBit(i_q); i_q++);
            q = idx2no(i_q);
        }
        for( i_p++; GetBit(i_p); i_p++);
    }
    [ ... ]
}
```

As we can see in Table 6, the variant based on SoM technique is indeed worse than optimized SoM and has a higher memory footprint, but the non-linked one manages to be significantly better than a similar optimized SoE that implements the $6k \pm 1$ pattern and uses the same 1bit coding approach. The linearity advantage of Gries/Misra over SoE is amplified as N_{max} increases, keeping the same $\frac{N}{24}$ low memory footprint: this allows us to compute up to 1 trillion in 64GB under fifty minutes.

3. TOUGHER NUTS

Of course, not all attempts at optimization will have the same initial efficiency – smaller steps are often in order, but in time they can and will take us far. Here are our first tentatives at another two linear algorithms.

3.1 Gale/Pratt

Gale/Pratt sieve (Pratt, 1977) was presented by V.Pratt in the same year with Mairson. The logic behind this sieve is somewhat convoluted: for each prime p it strikes out composited formed by powers of p and all reasonable composites stroked out in the previous iteration(s). Still, the published sieve was already pretty optimized, packing info at bit level and skipping over even numbers, thus

Euler / Mairson

2	3	5	7
2	2	2	2
3	3	3	3
4			
5	5	5	5
6			
7	7	7	7
8			
9	9		
10			
11	11	11	11
12			
13	13	13	13
14			
15	15		
16			
17	17	17	17
18			
19	19	19	19
20			
21	21		
22			
23	23	23	23
24			
25	25	25	
26			
27	27		
28			
29	29	29	29
30			
31	31	31	31
32			
33	33		
34			
35	35	35	
36			
37	37	37	37
38			
39	39		
40			
41	41	41	41
42			
43	43	43	43
44			
45	45		
46			
47	47	47	47
48			
49	49	49	49
50			
51	51		
52			
53	53	53	53
54			
55	55	55	
56			
57	57		
58			
59	59	59	59
60			
61	61	61	61
62			
63	63		
64			
65	65	65	
66			
67	67	67	67
68			
69	69		
70			
71	71	71	71
72			
73	73	73	73
74			
75	75		
76			
77	77	77	77
78			
79	79	79	79
80			
81	81		
82			
83	83	83	83
84			
85	85	85	
86			
87	87		
88			
89	89	89	89
90			
91	91	91	91
92			
93	93		
94			
95	95	95	
96			
97	97	97	97
98			
99	99		
100			

Gries & Misra

p	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5	7	7	7
q	2	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	3	5	7	11	13	17	19	23	25	29	31	5	7	11	13	17	19	7	11	13
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4																																												
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6																																												
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8																																												
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9																			
10	10	10																																										
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
12	12	12																																											
13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
14	14	14	14																																									
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15																		
16																																												
17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
18	18	18	18	18																																								
19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
20	20	20																																										
21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
22	22	22	22	22	22																																							
23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
24	24	24																																										
25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25								
26	26	26	26	26	26	26																																						
27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27																			
28	28	28	28																																									
29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
30	30	30	30	30	30	30	30																																						
31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
32																																												
33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33								
34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34																			
35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39
40	40	40																																										
41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
42	42	42	42	42	42	42	42	42	42	42	42																																		
43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43
44	44	44	44	44	44	44																																						
45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45
46	46	46	46	46	46	46	46	46	46	46	46																																		
47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47	47
48	48	48																																										
49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49
50	50	50	50	50	50	50	50	50	50	50	50	50																																	
51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51
52	52	52	52	52	52	52	52																																					
53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53
54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54	54																			
55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55
56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56
57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57
58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58	58
59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59
60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60
61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62	62
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
64	64	64																																											
65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65	65
66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66
67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67	67
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68
69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69
70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70
71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71	71
72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73
74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74	74
75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75
76	76	76	76	76	76	76	76	76	76	76	76																																		
77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77
78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78	78																			
79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79
80	80	80																																										
81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81	81																		
82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82	82																			
83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83
84	84	84	84	84	84	84	84	84	84	84	84																																		
85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85	85
86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86	86																			
87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87	87
88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88	88
89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89	89
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90
91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91	91
92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92	92
93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93	93
94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94	94																			
95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95
96	96	96																																										
97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97
98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98	98																			
99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99
100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100																			

Table 7. Euler/Mairson vs Gries&Misra combing techniques applied to $N = 100$

having from the start a respectable performance and a relatively good memory footprint — here is an updated but accurate version of this algorithm:

Algorithm 12 Gale and Pratt sieve

- The management of the lists can be quite taxing

```
void GalePratt(int Nmax) {
    int i, k, m, p;
    std::vector<int> s = { 1 }, ts = {};
    int numPrimes = 1; AddPrime(2);
    for (p = 3; p <= Nmax / 2; p += 2)
        if (!GetBit(p)) {
            numPrimes++; AddPrime(p);
            for (auto j : s)
                for(k = j; k < Nmax; k = m)
                {
                    if ((k != 1) && (k != p) && (k != j))
                        SetBit(k);
                    m = k * p;
                    if (m < Nmax)
                        ts.push_back(k);
                }
            s = ts; ts.resize(0);
        }
    for (; p <= Nmax; p += 2)
        if (!GetBit(p)) { numPrimes++; AddPrime(p); }
}
```

There are several points of attack here: some are small, other quite promising. First of all we look at those lists of multiplication factors and we observe that their length is not at all as big as we expected and its maximum occurs quite early in the sieving process: we can see in Table 8, it is thousands times lower than N_{max}, so the list can be reused without the need to copy it by simply using two lists and swapping them at each iteration. Also, the last sieving prime is not at $N_{max}/2$, but somewhere at $N_{max}/3$ — actually, the last sieving prime occurs when no more items are added to the list (the only element left in the list is that prime), which is a better exit condition.

We also observe that we step by 2 in the loops, only to divide by 2 in *Set/GetBit* — a simple increment would avoid this. Furthermore, the conditions in the inner loop are obscure and quite complicated: the CPU pipeline will get very confused, as will be the person who will try to read the algorithm the first time — we should simplify the logic of the algorithm to avoid all those *if*s for better performance and comprehension.

Algorithm 13 Gale and Pratt sieve — optimized

- The vectors are now static — double the size with double performance

```
void GP_Opt(int Nmax, byte mark[], int v1[], int v2[]) {
    int* s = v1; int* ts = v2;
    int numPrimes = 1; AddPrime(2);

    int i, j, p, idx_s = 0;
    for (i = 0; i <= Nmax / 16; i++) mark[i] = 0;

    for (i = 1; true; i++)
        if (!GetBit(i)) {
            ts[0] = p = 2 * i + 1;
            numPrimes++; AddPrime(p);
            int m, idx_ts = 1;
            for(m = p * p; m <= Nmax; m *= p)
                { SetBit(m/2); ts[idx_ts++] = m; }

            for (j = 0; j < idx_s; j++) {
                int k = s[j];
                for (m = k*p; m <= Nmax; k = m, m = k*p)
                    { ts[idx_ts++] = k; SetBit(m/2); }
            }
            if (idx_ts == 1) break; //no more items
            idx_s = idx_ts; std::swap(s, ts);
        }
    for (i++; i < Nmax/2; i++)
        if (!GetBit(i))
            { numPrimes++; AddPrime(2 * i + 1); }
}
```

This version more than doubles the performance (timings in Table 9) with some memory cost, which is negligible compared with size of the vector required to mark composites.

Most of the time is spent passing values from one list to another — a solution that would use a single vector, somehow preserving the useful values from one iteration to another using a simple enough mechanism, could further improve performance (for example, zero out exit values and add new ones over).

***NOTE 5**: In the accompanying code there is also a brief tentative to optimize Pritchard's 4.2 algorithm (Pritchard, 1987), his attempt for a generalization of Gale/Pratt.*

3.2 Bengelloun

(Bengelloun, 1986) comes with the last innovative attempt to create a new linear sieve, achieving also an incremental one. Bengelloun keeps track of the least prime factor (lpf) of composites in a vector that increases incrementally with n. For each odd composite another greater composite is stroked out: any composite n can be expressed as $n = p \times q$, where p is the least prime factor of n — getting p_{next} = the immediate next prime greater than p, the number $n_{next} = p_{next} \times q$ is stroke out by inscribing p_{next} in the lpf vector at position n_{next}.

Algorithm 14 Bengelloun Continuous Incremental Primal Sieve

- The locality is not the best, and the divisions and multiplications further impair the performance.

```
int Ben_Std(int Nmax, int p[], int lpf[]) {
    p[0] = lpf[0] = 0xFFFF; //idx 0 not used
    lpf[1] = lpf[2] = 0;
    int sz_lpf = 2, sz_p = 0;

    for (int n = 2; n <= Nmax; n++) {
        if (lpf[n] == 0)
            { p[++sz_p] = n; lpf[n] = sz_p; }
        else {
            int q = n / p[lpf[n]];
            if (lpf[n] < lpf[q]) {
                int r = lpf[n] + 1;
                lpf[q * p[r]] = r; } }
        lpf[++sz_lpf] = 0; lpf[++sz_lpf] = 1; }
    return sz_p;
}
```

This sieve is very elegant, but the performance does not shine, and the "incremental" character is more theoretical than practical, due to continuous doubling in size.

There are several ideas we can employ to optimize the sieve. First, studying the behaviour of the algorithm, we can observe that the values for q are actually only computed initially for the even numbers and then propagated on. For those even ns, $q = n/2$, which is not a real division operation. Storing this value for further usage would save all those real divisions. Second, we can see how all the space used for the even values is not really required and can be used instead to store those q values. Furthermore, most operations done for even ns are useless or can be pre-computed during initialization. Also, there is no real need to keep the parallel vector p: those values can also be kept in the main vector as index of the next prime value. Algorithm 15 depicts the optimized algorithm, including some other minor refactoring.

n	Maximum values up to $N_{max} = 10^n$				
	Vector size	Vector value	p at max size	last p	N_{max} / max size
2	5	15	5	31	20
3	20	135	13	331	50
4	83	1'875	23	3'331	120
5	347	19'683	47	33'331	288
6	1'550	177'147	113	333'331	645
7	7'166	1'594'323	113	3'333'331	1'395
8	34'466	18'984'375	283	33'333'331	2'901
9	170'504	199'290'375	467		5'865
10	875'840	9'603'838'835	661	etc.	11'418

Table 8. GP vector related values

n	$\pi(10^n)$	Gale/Pratt iterations			SoE 6k
		Standard	Two vectors	All Refactoring	
5	9'592	1	1	1	1
6	78'498	4	4	2	2
7	664'579	34	20	15	16
8	5'761'455	380	237	173	149
9	50'847'534	9'273	6,733	4'989	2'496
10	455'052'511	-	-	-	29'102

Table 9. Timings [ms] for Gale/Pratt optimization

Besides the very noticeable performance gains visible in Table 10, we have also marginally improved footprint by eliminating the parallel vector p. Nevertheless, although the sieve is decent within millions range, the 64bit vector makes it absolutely impractical above 1 billion.

Algorithm 15 Bengelloun Sieve optimized

- The sieve is capped at N_{max} to be able to compare it with all the others. It can be easily transformed back in incremental by moving the initialization inside the main loop and eliminating the exit test

```
void Ben_Opt3(int Nmax, int lpf[]) {
    lpf[2] = 2; AddPrime(2);
    int numPrimes = 1, last_p = 2;
    int q, r, n, ix, lpfn;
    for (n = 3; n < Nmax; n += 4)
        { lpf[n] = lpf[n + 2] = 0; }
    for (n = 3, ix = 9; ix <= Nmax; n += 2, ix += 6)
        { lpf[ix] = 3; lpf[ix - 1] = n; }
    for (n = 3; n <= Nmax; n += 2) {
        lpfn = lpf[n];
        if (lpfn == 0) {
            last_p = lpf[last_p - 1] = lpf[n] = n;
            numPrimes++;  AddPrime(n);
        }
        else {
            q = lpf[n - 1]; assert(q == n / lpfn);
            if (lpfn < lpf[q]) {
                r = lpf[lpfn - 1];
                ix = q * r;
                if (ix <= Nmax)
                    { lpf[ix] = r; lpf[ix - 1] = q; }
            }
        }
    }
}
```

n	Standard	Optimized
5	1	1
6	8	2
7	105	47
8	1'314	601
9	13'945	6'194

Table 10. Bengelloun optimization timings [ms]

4. SUB-LINEAR

In 1981 Paul Pritchard published a beautiful prime sieving algorithm – **The Wheel of Pritchard** (SoP) (Pritchard, 1981). SoP technique is somehow derived from Mairson's, except it does not parse the whole set of candidates for each p to eliminate its multiples, but instead tries to iteratively generate a smaller set of candidates to work on. In order to understand the algorithm we need to recap some basic mathematical ideas:

- The product P of the first n prime numbers is called a **Primorial**

$$P(n) = \prod_{i=1}^{n} p_i \; (\leftarrow \; p_1 = 2, \ldots \quad p_i \text{ is prime}) \qquad (4)$$

  $P(1) = 2, P(2) = 6, P(3) = 30, P(4) = 210$ etc.
- If δ is coprime with any p_i, then $P(n) + \delta$ is also coprime with any p_i.
- A **Wheel** W_k is the set of numbers not divisible by any of the first k primes:
  $W_k = \{n \mid 1 \leqslant n \leqslant P(k) \text{ and } (n, P(k)) = 1\}$
  $W_1 = \{1\}, W_2 = \{1, 5\}, W_3 = \{1, 7, 11, 13, 17, 19, 23, 29\}$ etc.
- We say that the **Length** of wheel W_k is the primorial $P(k)$

SoP iterates larger and larger wheels up to the desired maximum and then sieves out composites on this reduced set of candidates. The algorithm is further clarified by the author in (Pritchard, 1982). Some very intuitive animations for the Wheel of Pritchard can be found on the web, for example at `https://en.wikipedia.org/wiki/Sieve_of_Pritchard`[3].

The arithmetic complexity of SoP is $\mathbf{O}(\frac{N}{\ln \ln N})$, the best one achieved so far. The sieve is theoretically sub-linear, but it needs a data structure smart enough to not destroy the potential performance of the sieve: Pritchard's brilliant idea is to use a single vector s that keeps in the lower part all p_i and in the rest the current s_i. The vector s combines both *rlink* and *llink* in one vector, capitalizing on the fact

[3] verified December 2024

that, except 2, even numbers are not prime, so $s[i]$ will be $rlink[i]$ and $s[i-1]$ will be $llink[i]$.

Algorithm 16 Sieve of Pritchard

- As suggested by Misra, in each iteration the multiples of current p are deleted in reverse order to avoid the need for an extra vector for that info

```
void SoP(const int N, int s[]) {
    for (int i = 0; i <= N; i++) s[i] = 0;
    int p, maxS = 1, length = 2;

    auto next = [&](int w) { return s[w]; };
    auto prev = [&](int w) { return s[w - 1]; };
    auto Append = [&](int w) {
        s[maxS] = w; s[w - 1] = maxS; maxS = w; };
    auto Delete = [&](int pf) {
        int temp1 = s[pf - 1], temp2 = s[pf];
        s[temp1] = temp2; s[temp2 - 1] = temp1; };
    auto ExtendTo = [&](int n) {
        int w = 1, x = length + 1;
        while (x <= n)
            { Append(x); w = next(w); x = length + w; }
        length = n;
        if (length == N)
            Append(N + 2); };
    auto DeleteMultiples = [&](int p) {
        int f = p;
        while (p * f <= length)
            f = next(f);
        while (f > 1)
            { f = prev(f); Delete(p * f); } };

    for (p = 3; p * p <= N; p = next(1)) {
        if (length < N)
            ExtendTo(std::min(p * length, N));
        DeleteMultiples(p); }
    if (length < N) ExtendTo(N);

    int  numPrimes = 1;  AddPrime(2);
    for (p = 3; p <= N; p = next(p))
        { numPrimes++; AddPrime(p); }
}
```

The algorithm is extremely clever, but it has $\mathbf{O}(N)$ 64bit memory complexity, which puts it outside any practical reach as it is: we must address both memory and performance to get to a realistic sieve. Several ideas can be applied here:

- Use a gap index instead of an absolute one to reduce the size of the vector
- Apply a similar delayed delete technique as for Mairson — of course, there are major differences, so the implementation is also quite different
- Start each iteration from the current p instead of 1 to streamline a lot of complications
- Extend the wheel in a repeated loop only over the elements of the previous wheel to avoid unnecessary delete operations
- Differentiate between cases when the advancement in the list may need or not the actual deletion.
- Use the Mairson pattern for the actual sieving

Algorithm 17 Sieve of Pritchard — optimized

- Including other minor refactoring for better performance and lower ceiling of real delete operations.

```
void SoP5(const int N, uint8_t s[], uint8_t flags[]) {
    int i, i_maxS = 0, length = 6;
    const int i_max = N / 24;
    for (i = 0; i<=i_max; i++) flags[i] = 0;//no primes

    auto Delete = [&](int pf) { ResetBit(no2idx(pf)); };
    auto next = [&](int w) {
        int i_w = no2idx(w);
        int next_i_w = i_w + s[i_w];
        if (!GetBit(next_i_w))
          {next_i_w+=s[next_i_w]; s[i_w]=next_i_w-i_w;}
        return (idx2no(next_i_w)); };
    auto next_simple = [&](int w) {
        int i_w = no2idx(w);
        int next_i_w = i_w + s[i_w];
        return (idx2no(next_i_w)); };
    auto DeleteMultiples = [&](int p) {
        int i_f = no2idx(p);
        for (int f=p; p * f <= length; f=idx2no(i_f))
          { Delete(p * f); i_f += s[i_f]; } };
    auto Append = [&](int w) {
        int i_w = no2idx(w); SetBit(i_w);
        s[i_maxS] = i_w - i_maxS; i_maxS = i_w; };
    auto Append_simple = [&](int w)
    { int i_w = no2idx(w); SetBit(i_w); };
    auto ExtendToNextP = [&](int p) {
        int base = length;
        Append(base + 1);
        for (int w = p; w < length; w = next(w))
            Append(base + w);
        base += length;
        for (int i = 2; i < p; i++) {
            Append(base + 1);
            for (int w=p; w<length; w=next_simple(w))
                Append(base + w);
            base += length; }
        length = base; };
    auto ExtendToN = [&](int p) {
        int base = length;
        Append(base + 1);
        for (int w = p; w < length; w = next(w))
            if (base + w <= N)
                Append_simple(base + w);
            else
                break;
        base += length;
        for (int i = 2; i < p and base < N; i++) {
            Append_simple(base + 1);
            for (int w=p; w<length; w=next_simple(w))
                if (base + w <= N)
                    Append_simple(base + w);
                else
                    break;
            base += length; }
        length = N; };
    int  p, q, pq, numPrimes = 2;
    Append(5); AddPrime(2); AddPrime(3);

    for (p = 5; p * length <= N; p = next(p)){
        ExtendToNextP(p); DeleteMultiples(p);
        numPrimes++; AddPrime(p);
    }
    if (length < N) {
        ExtendToN(p); DeleteMultiples(p);
        numPrimes++; AddPrime(p); p = next(p);
    }
    int nsqrt = (int)floor(sqrt(N));
    for (; p <= nsqrt; p = next(p)) {
        numPrimes++; AddPrime(p); q = p;
        for (int pq = p * q; pq <= N; pq = p * q) {
            for (int n = pq; n <= N; n *= p)
                Delete(n);
            q = next(q);
        }
    }
    [ ... ]
}
```

There is significantly more complexity here, diminishing the beauty of the original, but performance gain is over 6x, as seen in Table 11.

Serious effort was put in the optimized algorithm to lower the ceiling of real delete operations (which directly determines the size of the gap vector) to the maximum wheel included in our set — thus, memory footprint is also seriously slashed to $\frac{N}{24} + \frac{1}{3}P_k$, where P_k corresponds to the largest wheel that is lower than N_{max}. Table 12 contains

n	Standard	Optimized
5	1	2
6	4	4
7	108	18
8	1'197	154
9	13'228	2'155
10	-	22'461

Table 11. SoP optimization timings [ms]

the values for all the wheels under 2^{64} . For $N = 200$ billion ($k = 11$) the memory is only $(\frac{200}{24} + 2.1) = 10.5$GB.

k	P_k	p_{k+1}	size of s
2	6	5	2
3	30	7	10
4	210	11	70
5	2'310	13	770
6	30'030	17	10'010
7	510'510	19	170'170
8	9'699'690	23	3'233'230
9	223'092'870	29	74'364'290
10	6'469,693,230	31	2'156'564'410
11	200'560'490'130	37	66'853'469'710
12	7'420'738'134'810	41	
13	304'250'263'527'210	43	
14	13'082'761'331'670'030	47	
15	614'889'782'588'491'410	53	
16	14'142'414'403'480'493'114	59	

Table 12. Values for s size

As a collateral benefit of this refactoring, the *Extend* method is now very easy to parallelize, looping again and again over the previous wheel; the parallelization becomes very easy for the extending and counting phases. Unfortunately, this is not true for the most time-consuming operation — sieving itself: any parallelization technique that would apply there will apply also to Mairson.

5. CONCLUSIONS

We have shown that all algorithms can be carried further, with serious 20x improvements for Mairson and Gries/Misra, a very respectable 6x gain for Pritchard and even a decent 2x improvement on Gale/Pratt and Bengelloun. All these optimizations were done while simultaneously reducing the memory footprint, even if not all with extreme memory savings like Mairson. And all these enhancements were performed only at the level of the single threaded algorithm, without capitalizing at all on parallel processing or pushing for extreme cache locality.

Of course, there is no apparent, immediate gain from this optimization effort. Nor any evidence that any of these algorithms are good candidates for the champion of tomorrow — actually, it is more or less clear that Gale/Pratt and Bengelloun will never match the others due to the need for $O(n)$ 64bit vectors.

But there is also clear that there is unseen potential in the old algorithms, and, even more important, such **fundamental research** ventures must be carried on at least for the valuable lessons that can be learn from these optimization endeavors than may apply successfully to other algorithms.

ACKNOWLEDGEMENTS

We express our gratitude to professors Nirvana POPESCU, Emil SLUSANSCHI and Vlad CIOBANU from National University of Science and Technology POLITEHNICA Bucharest, Computer Science Department, for their invaluable guidance and advice — their input was decisive for the quality of this paper.

CRediT [4] **author statement**: ***Mircea Ghidarcea***: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – Original & Editing, Visualization; ***Decebal Popescu***: Validation, Supervision, Project administration, Writing – Review

Funding: This research did not receive any specific grant from public, commercial or not-for-profit funding agencies.

Conflicts of Interest: The authors declare no conflict of interest.

REFERENCES

Atkin, A.O.L. and Bernstein, D.J. (2003). Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246), 1023–1030. doi:10.1090/S0025-5718-03-01501-1.

Bengelloun, S.A. (1986). An incremental primal sieve. *Acta Informatica*, 23(2), 119–125. doi:10.1007/BF00289493.

Crandal, R. and Pomerance, C. (2005). *Prime Numbers: A Computational Perspective*. Springer-Verlag, New York. doi:10.1007/0-387-28979-8.

Gries, D. and Misra, J. (1978). A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12), 999–1003. doi:10.1145/359657.359660.

Mairson, H.G. (1977). Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20(9), 664–669. doi:10.1145/359810.359838.

Misra, J. (1981). An Exercise in Program Explanation. *ACM Transactions on Programming Languages and Systems*, 3(1), 104–109. doi:10.1145/357121.357128.

Nicomachus (1926). *Introduction to Arithmetic*. Studies. Humanistic Series. Macmillan.

Pratt, V.R. (1977). CGOL - an Algebraic Notation For MACLISP users. *Working paper, MIT AI Lab, Cambridge*.

Pritchard, P. (1981). A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1), 18–23. doi:10.1145/358527.358540.

Pritchard, P. (1982). Explaining the wheel sieve. *Acta Informatica*, 17(4). doi:10.1007/BF00264164.

Pritchard, P. (1987). Linear prime-number sieves: A family tree. *Science of Computer Programming*, 9(1), 17–35. doi:10.1016/0167-6423(87)90024-4.

Pritchard, P. (1994). Improved incremental prime number sieves. Technical report, Springer-Verlag Springer e-books, Berlin, Heidelberg.

Singleton, R.C. (1969). Algorithm 357: An efficient prime number generator [A1]. *Communications of the ACM*, 12(10), 563–564. doi:10.1145/363235.363247.

Wood, T.C. (1961). Algorithm 35: Sieve. *Communications of the ACM*, 4(3), 151. doi:10.1145/366199.366257.

[4] `https://credit.niso.org/`