SIEVE OF ERATOSTHENES AND WHEEL FACTORIZATION

V. Barbera

Abstract

This paper presents a refinement of the Sieve method of Eratosthenes in conjunction with wheel factorization. The result is to use less memory.

Sieve Wheel or Modular Sieve

With the sieve of Eratosthenes[1] algorithm in the Boolean vector *SIEVE* of size n initially all set to *true* all elements associated with multiples of primes p can be set to *false* using this pseudocode:

```
for (p = 2; p < sqrt(n); p++)
   if ( SIEVE[p] )
      for (m = p * p;  m < n; m += p)
         SIEVE[m] = false;
```

An improvement can be made by using the Wheel factorization[2] or equivalently by the use of modular arithmetic[3].

Given an integer $m \geq 1$, called modulus, two integers p and q are congruent modulo m $p \equiv q \pmod{m}$ if m is a divisor of their difference $p - q$. As we will see, the modulus m corresponds to the size of the wheel which we will indicate with $bW = m$, furthermore we will indicate the operator modulo $q \bmod m$ also with $q \% m$.

We therefore consider the modulo operator $p \bmod bW$ which denotes the unique integer r such that $0 \leq r < bW$ and $r \equiv p \pmod{bW}$ then $p = r + k \cdot bW$ where r is the remainder of p when divided by bW.

In modular arithmetic the set of integers $\{0, 1, 2, \ldots, bW - 1\}$ is called the least residue system modulo bW so let's take a specific one residue system modulo bW set of $\varphi(bW)$ integers, where $\varphi(bW)$ is Euler's totient function[4], that are relatively prime to bW and mutually incongruent under modulus bW, called a reduced residue system modulo bW.

To make a sieve wheel for the purpose to find prime numbers less than n we choose $bW < \sqrt{n}$ and bW divisible by a set of prime numbers $\{p_1, p_2, \ldots, p_s\}$ then we choose an appropriate reduced residue system modulo bW stored in the vector RW of length $nR = \varphi(bW)$.

In this way we only store the numbers belonging to the congruence class or residue in RW and therefore multiples of $\{p_1, p_2, \ldots, p_s\}$ are automatically excluded.

Example if $bW=30$ then $\varphi(30)=8$ and $RW=[-23,-19,-17,-13,-11,-7,-1,1]$ is a reduced residue system modulo bW .
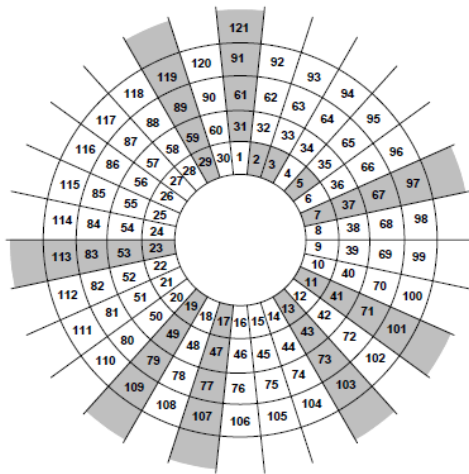
7	37	67	97
11	41	71	101
13	43	73	103
17	47	77	107
19	49	79	109
23	53	83	113
29	59	89	119
31	61	91	121

Analyzing the graph to describe the wheel factorization, it can be seen that, excluding 2 , 3 and 5 , if $p=RW[i]+30\cdot k$ with $k>0$ we can only store the useful areas in gray equal to a matrix with a number of rows equal to $nR=\varphi(30)=8$.

To find prime numbers different from $\{p_1,\ p_2,\ ...,\ p_s\}$ then we use a Boolean array $SIEVE$ of size $nR*\lceil n/bW\rceil$ in order to associate the possible residue in RW to each row of the array and so all elements associated with multiples of the prime numbers $\{p_1,\ p_2,\ ...,\ p_s\}$ are not stored.

So we want to get after the sieve that $p=RW[i]+bW\cdot j$ is prime if $SIEVE[i,j]$ == $true$.

Example in the case of $bW=6$ it's used a Boolean array $2*\lceil n/6\rceil$ or two Boolean vectors of size $\lceil n/6\rceil$.

In the second for loop of the pseudocode of the sieve of Eratosthenes for set to $false$ all elements associated multiples of p the initial index is $m_{min}=p\cdot p$ so now we have $p=r+k\cdot bW$ and $p\cdot p$ must be replaced by $(r+bW\cdot k)\cdot(s+bW\cdot k)$

where s is a remainder such that $(r\cdot s)\ \%\ bW=t$ and the residue t is the one associated with the row we are using, then

$$(r+bW\cdot k)\cdot(s+bW\cdot k)=(r\cdot s)\ \%\ bW\ +bW\cdot(bW\cdot k\cdot k+k\cdot r+k\cdot s+\lfloor(r\cdot s)/bW\rfloor)$$

and so for the row associated with remainder t for multiples of $p=r+k\cdot bW$ we use

$$m_{min}=bW\cdot k\cdot k+k\cdot r+k\cdot s+\lfloor(r\cdot s)/bW\rfloor$$

Then in the Boolean array *SIEVE* of size *2*(n/6+1)* initially all set to *true* and elements associated with multiples of primes *-1+6*k* and *1+6*k* can be set to *false* using this pseudocode:

```
for (k=1; k<=sqrt(n)/6; k++){
    if (SIEVE[0,k]){
        for (m=6*k*k; m<n/6+1; m+=-1+6*k)
            SIEVE[0,m]=false;
        for (m=6*k*k-2*k; m<n/6+1; m+=-1+6*k)
            SIEVE[1,m]=false;}
    if (SIEVE[1,k]){
        for (m=6*k*k; m<n/6+1; m+=1+6*k)
            SIEVE[0,m]=false;
        for (m=6*k*k+2*k; m<n/6+1; m+=1+6*k)
            SIEVE[1,m]=false;}
}
```

In general if $\quad p = RW[j] + bW \cdot k \quad$ (for convenience we consider $RW[j] \leq 1$ and $k > 0$) and if $s = RW[x]\quad$ we have:

$$(RW[x] + bW \cdot k) \cdot (RW[j] + bW \cdot k) = (RW[x] \cdot RW[j]) + bW \cdot (bW \cdot k \cdot k + k \cdot RW[x] + k \cdot RW[j]) =$$

$$= (RW[x] \cdot RW[j]) \% \ bW + bW \cdot (bW \cdot k \cdot k + k \cdot RW[x] + k \cdot RW[j] + \lfloor (RW[x] \cdot RW[j])/bW \rfloor)$$

and $\quad m_{min} = bW \cdot k \cdot k + k \cdot (RW[x] + RW[j]) + \lfloor (RW[x] \cdot RW[j]) / bW \rfloor$

or if $\ (RW[x] \cdot RW[j]) \% \ bW > 1\ $ adding and subtracting bW becomes

$$m_{min} = bW \cdot k \cdot k + k \cdot (RW[x] + RW[j]) + \lfloor (RW[x] \cdot RW[j]) / bW \rfloor + 1$$

we build two array of size $\ nR \ast nR\ $ for the coefficients C_1 and C_2 then for each $RW[i]$

and for each $RW[j]$ finding $RW[x]$ such that $\ (RW[x] \cdot RW[j]) \% \ bW = RW[i]$

then if $\ (RW[x] \cdot RW[j]) \% \ bW = RW[i]\ $ we have $\ C_1[i,j] = RW[x] + RW[j]$

and if $\ RW[i]=1\ $ or $\ RW[j]=1\ $ or $\ RW[i]=RW[j]\ $ then $C_2[i,j] = \lfloor (RW[x] \cdot RW[j]) / bW \rfloor$ otherwise $\ C_2[i,j] = 1 + \lfloor (RW[x] \cdot RW[j]) / bW \rfloor$

In the row corresponding to the residue $RW[i]$ if $\ p = RW[j] + bW \cdot k\ $ then

$$m_{min} = bW \cdot k \cdot k + k \cdot C_1[i,j] + C_2[i,j]$$

Example *bW = 30*

nR=8   and   RW=[-23, -19, -17, -13, -11, -7, -1, 1]

C1=

```
-22, -32, -28, -32, -28, -8,  -8,  -22
-30, -18, -30, -30, -12, -30, -12, -18
-34, -26, -16, -14, -34, -26, -14, -16
-42, -42, -18, -12, -18, -18, -18, -12
-46, -20, -34, -26, -10, -14, -20, -10
-24, -36, -36, -24, -24,  -6, -24, -6
-36, -30, -24, -36, -30, -24,  0,   0
-40, -38, -40, -20, -22, -20, -2,   2
```

C2=

```
 0,  9,  7, 9, 7, 1, 1 , 0
 6,  0,  8, 8, 1, 6, 1, 0
 9,  5,  0, 1, 9, 5, 1, 0
15, 15,  1, 0, 3, 3, 1, 0
18,  1, 10, 6, 0, 2, 1, 0
 1, 11, 11, 5, 5, 0, 1, 0
10,  7,  4, 10, 7, 4, 0, 0
13, 12, 13, 3, 4, 3, 0, 0
```

In the Boolean array *SIEVE* of size $nR * \lceil n/bW \rceil$ initially all set to *true* all elements associated with multiples of primes $p = RW[j] + bW \cdot k$ can be set to *false* using this pseudocode:

```
for (k = 1; k <= sqrt(n) / bW; k++)
    for (j = 0;  j < nR ;  j++)
        if( SIEVE[ j, k ] )
            for (i = 0; I < nR ; i++)
                for (m = bW * k  * k + k * C1[ i, j ] + C2[ i, j ]; m < n / bW + 1; m + =RW[ j ] + bW * k)
                    SIEVE[ i, m ] = false;
```

An improvement obtained is to have numbers smaller than n/bW and the use of a memory equal to $\varphi(bW) \cdot n / bW$.

In addition the possibility of making a segmented version using a bit space $\varphi(bW) \cdot \sqrt{n} / bW$, an example is shown below with the possible choice of the value of the wheel modulus.

Other sieves generally use $\sqrt{n}$ as memory for segmentation instead this wheel sieve uses the product of the prime numbers following the basis $\{p_1, p_2, \ldots, p_s\}$ with $p_1 = 2$ and $p_1 < p_2 < \ldots < p_s$ so that a pre-sieving can be done, in this way the memory used is slightly higher than $\varphi(bW) \cdot \sqrt{n} / bW$ but is always less than $\sqrt{n}$.

Segmented bit Wheel Sieve

Below is shown the C++ code of a segmented bit wheel sieve with adjustable modulus:

```cpp
///    This is an implementation of the bit wheel segmented sieve
///    with max modulus wheel choice 30,  210, 2310
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <cstdlib>
#include <stdint.h>
#include <chrono>


const int64_t PrimesBase[6] = {2, 3, 5, 7, 11, 13};
const int64_t n_PB_max = 5;

const int64_t del_bit[8] =
{
  ~(1 << 0),~(1 << 1),~(1 << 2),~(1 << 3),  ~(1 << 4),~(1 << 5),~(1 << 6),~(1 << 7)
};

const int64_t bit_count[256] =
{
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,  3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,  3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,  3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
  3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,  4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
};

int64_t Euclidean_Diophantine( int64_t coeff_a, int64_t  coeff_b)
{
    // return y in Diophantine equation coeff_a x + coeff_b y  = 1
    int64_t k=1;
    std::vector<int64_t> div_t;
    std::vector<int64_t> rem_t;
    std::vector<int64_t> coeff_t;
    div_t.push_back(coeff_a);
```

```
      rem_t.push_back(coeff_b);
      coeff_t.push_back((int64_t)0);
      div_t.push_back((int64_t)div_t[0]/rem_t[0]);
      rem_t.push_back((int64_t)div_t[0]%rem_t[0]);
      coeff_t.push_back((int64_t)0);
      while (rem_t[k] > 1)
      {
         k++;
         div_t.push_back((int64_t)rem_t[k-2]/rem_t[k-1]);
         rem_t.push_back((int64_t)rem_t[k-2]%rem_t[k-1]);
         coeff_t.push_back((int64_t)0);
      }
      k--;
      coeff_t[k] = -div_t[k+1];
      if (k > 0)
         coeff_t[k-1] = (int64_t)1;
      while (k > 1)
      {
         k--;
         coeff_t[k-1]=coeff_t[k+1];
         coeff_t[k] += (int64_t)(coeff_t[k+1]*(-div_t[k+1]));
      }
      if (k == 1)
         return (int64_t)(coeff_t[k-1] + coeff_t[k] * (-div_t[k]));
      else
         return (int64_t)(coeff_t[0]);
   }


   int64_t segmented_bit_sieve_wheel(uint64_t n, int64_t max_bW)
   {
      int64_t sqrt_n = (int64_t) std::sqrt(n);
      int64_t  count_p = (int64_t)0;
      int64_t n_PB = (int64_t)3;
      int64_t bW = (int64_t)30;
      //get bW modulus equal to p1*p2*...*pn <=max_bW  with n=n_PB
      while(n_PB < n_PB_max && (bW * PrimesBase[n_PB] <= std::min(max_bW, sqrt_n)))
      {
         bW *= PrimesBase[n_PB];
         n_PB++;
      }
      for (int64_t i = 0;  i <  n_PB; i++)
```

```cpp
         if (n > PrimesBase[i])
            count_p++;

   if (n > 1 + PrimesBase[n_PB - 1]){
      int64_t k_end = (n  < bW) ? (int64_t)2 : (int64_t) (n / (uint64_t)bW + 1);
      int64_t k_sqrt = (int64_t) std::sqrt(k_end / bW) + 1;


      //find reduced residue system modulo bW
      std::vector<char> Remainder_t(bW, true);
      for (int64_t i = 0; i <  n_PB; i++)
         for (int64_t j = PrimesBase[i]; j < bW; j += PrimesBase[i])
            Remainder_t[j] = false;
      std::vector<int64_t> RW;
      for (int64_t j = 2; j < bW; j++)
         if (Remainder_t[j]==true)
            RW.push_back(-bW+j);
      RW.push_back(1);
      int64_t  nR = RW.size();   // nR=phi(bW)
      std::vector<int64_t> C1(nR * nR);
      std::vector<int64_t> C2(nR * nR);
      for (int64_t j = 0; j < nR-2; j++)
      {
         int64_t rW_t, rW_t1;
         rW_t1=Euclidean_Diophantine(bW, -RW[j]);
         for (int64_t i = 0; i< nR; i++)
         {
            if (i == j)
            {
               C2[nR * i + j]=0;
               C1[nR * i + j] = RW[j] + 1;
            }
            else if(i == nR-3-j )
            {
               C2[nR * i + j] = 1;
               C1[nR * i + j] = RW[j] - 1;
            }
            else
            {
               rW_t = (int64_t)(rW_t1 * (-RW[i])) % bW;
               if (rW_t > 1)
                   rW_t -= bW;
```

```cpp
        C1[nR * i + j] = rW_t + RW[j];
        C2[nR * i + j]=(int64_t)(rW_t * RW[j]) / bW + 1;
        if (i == nR  -1)
           C2[nR * i + j] -= 1;
     }
   }
   C2[nR * j + nR-2]=(int64_t)1;
   C1[nR * j + nR-2]=-(bW + RW[j]) - 1;
   C1[nR * j + nR-1]=RW[j] + 1;
   C2[nR * j + nR-1]=(int64_t)0;
}
for (int64_t i= nR-2; i < nR; i++)
{
   C2[nR*i+nR-2] = (int64_t)0;
   C1[nR*i+nR-2] = -RW[i]-1;
   C1[nR*i+nR-1] = RW[i]+1;
   C2[nR*i+nR-1] = (int64_t)0;
}
int64_t  nB = nR / 8;
int64_t segment_size=1;
int64_t p_mask_i=(int64_t)4;
for (int64_t i=0; i<p_mask_i;i++)
   segment_size*=(bW+RW[i]);      // if bW=30  =7*11*13*17
while (segment_size<k_sqrt && p_mask_i<7)
{
   segment_size*=(bW+RW[p_mask_i]);      // if bW=30 max value =7*11*13*17*19*23*29
   p_mask_i++;
}

int64_t segment_size_b=nB*segment_size;
std::vector<uint8_t> Primes(nB+segment_size_b, 0xff);
std::vector<uint8_t> Segment_i(nB+segment_size_b, 0xff);
 int64_t  pb,mb,mmin,ib,i,jb,j,k,kb;
int64_t kmax = (int64_t) std::sqrt(segment_size/bW)+(int64_t)1;
for (k =(int64_t)1; k  <= kmax; k++)
{
   kb=k*nB;
   for (jb = 0; jb<nB; jb++)
   {
      for (j = 0; j<8; j++)
      {
```

```cpp
              if(Primes[kb+jb] & (1 << j))
                {
                  for (ib = 0; ib<nB; ib++)
                    {
                      for (i = 0; i<8; i++)
                        {
                          pb=nB*(bW*k+RW[j+jb*8]);
                          mmin=nB*(bW*k*k + k*C1[(i+ib*8)*nR+j+jb*8] + C2[(i+ib*8)*nR+j+jb*8]);
                          for (mb =mmin; mb <= segment_size_b && mb>=(int64_t)0; mb +=pb )
                            Primes[mb+ib] &= del_bit[i];
                          if (pb<nB*(bW+RW[p_mask_i]) && k_end>segment_size)
                            {
                              mb-=segment_size_b;
                              while (mb<(int8_t)0)
                                mb+=pb;
                              for (; mb <= segment_size_b; mb +=pb )
                                Segment_i[mb+ib] &= del_bit[i];
                            }
                        }
                    }
                }
        }
        for (kb = nB; kb < std::min (nB+segment_size_b,nB*k_end); kb++)
          count_p+=bit_count[Primes[kb]];
        if (kb==nB*k_end && kb<=segment_size_b && kb>(int64_t)0)
          for (ib = 0; ib<nB; ib++)
            for (i = 0; i  < 8; i++)
              if(Primes[kb+ib]& (1 << i) && RW[i+ib*8]<(int64_t)(n%bW-bW))
                count_p++;


        if (k_end>segment_size)
        {
          int64_t k_low, kb_low;
          std::vector<uint8_t> Segment_t(nB+segment_size_b);
          for (int64_t k_low = segment_size; k_low < k_end; k_low += segment_size)
          {
            kb_low=k_low*nB;
            for (kb = (int64_t)0; kb <(nB+segment_size_b); kb++)
              Segment_t[kb]=Segment_i[kb];
```

```cpp
                kmax=(std::min(segment_size,(int64_t)std::sqrt((k_low+segment_size)/bW)+2));
                j=p_mask_i;
                for(k=(int64_t)1; k<=kmax; k++)
                {
                    kb=k*nB;
                    for (jb = 0; jb<nB; jb++)
                    {
                        for (; j < 8; j++)
                        {
                            if (Primes[kb+jb]& (1 << j))
                            {
                                for (ib = 0; ib<nB; ib++)
                                {
                                    for (i = 0; i < 8; i++)
                                    {
                                        pb=bW*k+RW[j+jb*8];
                                        mmin=-k_low+bW*k*k+ k*C1[(i+ib*8)*nR+j+jb*8] + C2[(i+ib*8)*nR+j+jb*8];
                                        if (mmin<0)
                                            mmin=(mmin%pb+pb)%pb;
                                        mmin*=nB;
                                        pb*=nB;
                                        for (mb =mmin; mb <= segment_size_b; mb += pb)
                                            Segment_t[mb+ib] &= del_bit[i];
                                    }
                                }
                            }
                        }
                        j=(int64_t)0;
                    }
                }
                for ( kb =nB+kb_low; kb <std::min (kb_low+segment_size_b+nB,nB*k_end); kb++)
                    count_p+=bit_count[Segment_t[kb-kb_low]];
            }
            if (kb==nB*k_end && kb-kb_low<=segment_size_b && kb-kb_low>(int64_t)0)
                for (ib = 0; ib<nB; ib++)
                    for (i = 0; i  < 8; i++)
                        if(Segment_t[kb-kb_low+ib]& (1 << i) && RW[i+ib*8]<(int64_t)(n%bW-bW))
                            count_p++;
        }
    }
    return  count_p;
```

```cpp
}

int main()
{
    int64_t  count_p;
    auto ti = std::chrono::system_clock::now();
    // segmented_bit_sieve_wheel(n, max_bW) with max_bW= 30 , 210 , 2310 for set modulus
    count_p = segmented_bit_sieve_wheel(1000000000, 210);
    auto tf = std::chrono::system_clock::now();
    std::cout << count_p<< " prime numbers found "<< std::endl;
    std::chrono::duration<double> delta_temp = tf - ti;
    std::cout << " \n" << "t: " << delta_temp.count() << " s" << std::endl;
    return 0;
}
```

bit Wheel Sieve size 6

```cpp
#include <stdlib.h>
#include <math.h>
#include <vector>
#include <iostream>
#include <stdint.h>
#include <chrono>

typedef uint64_t word_type;
int word_exp = 6;
int word_m1 = (1 << word_exp) - 1;

unsigned long long int sieve_prime_wheel_6(unsigned long long int n)
{
    unsigned long long int count_p = (n < 3ull) ? 0 : (n < 4ull) ? 1 :(n < 6ull) ? 2 :(n < 8ull) ? 3 :(n <= 10ull) ? 4 : 2;
    if (n > 10ull){
        unsigned long long int kmax = n / 6;
        unsigned long long int dim_vec = ((kmax + word_m1) >> word_exp);
        std::vector<word_type> Prime5mod6(dim_vec, 0ull);
        std::vector<word_type> Prime1mod6(dim_vec, 0ull);
        Prime5mod6[0] |= 1;
        Prime1mod6[0] |= 1;
        if((kmax & word_m1) > 0){
            for(unsigned long long int kb = (kmax & word_m1) + 1; kb <= word_m1; kb++){
                Prime5mod6[dim_vec - 1] |= (1ull << kb);
                Prime1mod6[dim_vec - 1] |= (1ull << kb);
            }
        }
        unsigned long long int m, ksqrt;
        ksqrt = (unsigned long long int)std::sqrt(kmax) + 1;
        for(unsigned long long int k = 1; k <= ksqrt; k++){
            if((Prime5mod6[k >> word_exp] & (1ull << (k & word_m1))) == 0){
                for(m = 6 * k * k; m <= kmax; m += (6 * k - 1)){
                    Prime5mod6[m >> word_exp] |= 1ull << (m & word_m1);
                    Prime1mod6[(m - 2 * k) >> word_exp] |= 1ull << ((m - 2 * k) & word_m1);
                }
                m -= 2 * k;
                for(; m <= kmax; m += (6 * k - 1)){
                    Prime1mod6[m >> word_exp] |= 1ull << (m & word_m1);
```

```cpp
            }
        }
        if((Prime1mod6[k >> word_exp] & (1ull << (k & word_m1))) == 0){
            for(m = 6 * k * k; m + 2 * k <= kmax; m += (6 * k + 1)){
                Prime5mod6[m >> word_exp] |= 1ull << (m & word_m1);
                Prime1mod6[(m + 2 * k) >> word_exp] |= 1ull << ((m + 2 * k) & word_m1);
            }
            for(; m <= kmax; m += (6 * k + 1)){
                Prime5mod6[m >> word_exp] |= 1ull << (m & word_m1);
            }
        }
    }
    for(unsigned long long int k = 1; k <= kmax; k++){
        if(((Prime5mod6[k >> word_exp] & (1ull << (k & word_m1))) == 0))
            count_p++;
        if(((Prime1mod6[k >> word_exp] & (1ull << (k & word_m1))) == 0))
            count_p++;
    }
    if(((Prime1mod6[kmax >> word_exp] & (1ull << (kmax & word_m1))) == 0) && n % 6 <= 1)
        count_p--;
}
return count_p;
}


int main()
{
    unsigned long long int n = 1000000000;

    auto ti = std::chrono::system_clock::now();
    std::cout << " found " << sieve_prime_wheel_6(n) << " primes < " << n << std::endl;
    auto tf = std::chrono::system_clock::now();

    std::chrono::duration<double> delta_temp = tf - ti;
    std::cout << " \n" << "t: " << delta_temp.count() << " s" << std::endl;

    return 0;
}
```

References

[1] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

[2] https://en.wikipedia.org/wiki/Wheel_factorization

[3] https://en.wikipedia.org/wiki/Modular_arithmetic

[4] https://en.wikipedia.org/wiki/Euler%27s_totient_function