Implementation of a Core(c) Number Sieve.

Helmut Preininger
1200 Vienna
Austria
August 28, 2017
mailto: helmut.preininger@chello.at
hosted at: ww.vixra.org

Abstract

In this paper we give an implementation of a Core(c) Number Sieve (for a given c=1,2,3,.. we sift out numbers that have in there factorization a prime with a power ¿= c). For c=2 we have a squarefree number sieve. (Note, that, for c=1, our implementation compute the usual prime number sieve.) Our goal is to use only one codebase and avoid extra algorithms for every c.
We use some well known algorithms and adopt it for our purpose.

1 The Sieve

Let $\mathbb{P}$ be the set of all prime numbers $p_i \in \mathbb{N}$. Every natural number $m \in \mathbb{N}$ can be expressed as $m = \Pi_i p_i^{\alpha_i}$, with $p_i \in \mathbb{P}$.
Let $c \in \mathbb{N}$, $c > 1$. Every $m \in \mathbb{N}$ can be decomposed into $m = a \cdot b$, with

$$a = \Pi_i p_i^{\alpha_i} \,, \alpha_i < c$$

and

$$b = \Pi_j p_j^{\alpha_j} \,, \alpha_j = n_j \cdot c \,, n_j = 1, 2, \ldots$$

Let $S(c)$ be a sieve where all numbers $m \in \mathbb{N}$ (less than an upper bound), of the form $m = \Pi_i p_i^{\alpha_i}$, $\alpha_i < c$, are marked.
In the special case $S(1)$ our implementation computes the usual prime number sieve.

1.1 Prerequisite: The programming language.

Every programming language[1] that admits bitwise boolean operations, bitwise shift operations and bytewise memory copies is fine.

[1]We use PureBasic, a small procedural programming language. It is bundled with a compiler, an IDE, a debugger and runs on all three major platforms.

1.2 The Data

Let $UB \in \mathbb{N}$ be the upper bound of $S(c)$. For every number $m = 0, .., UB$ we only store the information: $m \in S(c)$ or $m \notin S(c)$. If the i-th bit is 0 it means $i \in S(c)$. Therefore we allocate a $UB + 1$ bit memory block. For large UB this approach is not usable.
We implement a *segmented* sieve, i.e. only a small portion of the sieve is present in the memory at one time (for more details to segmented sieves see for example [RICHJ]).

1.3 Bitwise access

It is not possible to access one bit of a memory block directly. Therefore we have to use bitwise shift and bitwise boolean operations. Fortunately this operations are very fast. To avoid a function call we realize it as macros.
If the memory is organized in 64 bit pieces, the macros are (note, that the syntax is related to PureBasic. It can be easily adopted to every other programming language which provide similar operations.):

```
Macro BitSet(_var_,_pos_)
  _var_\i[(_pos_) >> 6] | (1 << ((_pos_) & 63))
EndMacro

Macro BitRead(_var_,_pos_)
  _var_\i[(_pos_) >> 6] & (1 << ((_pos_) & 63))
EndMacro
```

Remark 1. *.*

- `_var_\i` *refer to the 64 bit piece with index i of a memory block (the first piece has index 0)*
- `_pos_` *is the* `_pos_`*-th bit of a memory block and is associated with the number* $m =$ `_pos_`
- `a >> 6` *is equivalent to* $a/64$
- `1 << k` *is equivalent to* 2^k
- `&` *.. bitwise boolean AND (Note,* `_pos_ & 63 = _pos_ MOD 64`*.)*
- `|` *.. bitwise boolean OR*
- `a | b` *is a short version of* `a = a | b`

1.4 The Segment

A 4-tupel $[LB, UB, c, Seg]$ is called *segment* where LB is the lower bound, UB the upper bound, c is the core level and Seg is a memory block of at least $UB - LB + 1$ bits. This segment include all numbers m in the interval $[LB, UB]$ and the i-th bit of the memory block Seg is associated with the number $m = LB + i$. Note, the first bit in the block is the 0-th bit.

1.5 The Basic Algorithm

Note, we mark all numbers m that are not in $S(c)$. (i.e. if $m \notin S(c)$ the m-th bit is set to 1)

The basic algorithm:
For all primes p_i with $p_i^{\max(2,c)} \leq UB$ (since for $c = 1$ we have p_i^2) we process the following procedure:
First we calculate the $offset$ (starting value), where we begin to mark. This value $offset$ is a function of c and LB. Now we marked every number m, $offset \leq m \leq UB$ with m is a multiple of p_i^c as $m \notin S(c)$.

Unfortunately the algorithm is slow and needs some improvements. We observe, that for all primes p_i, the mark pattern of p_i^c is periodic with length k_i. All our improvements rely on this fact. Note, that this improvements are well known (see, for example, [RICHJ]).

1.5.1 Helper: CopySieveBytes

The helper function `CopySieveBytes` has three parameters:
`Sieve` .. The address of the `memoryblock + offset`
`n` .. The length, in bytes, of the smallest period.
`MaxBytes` .. The length of the memoryblock (in bytes) `= SegmentBytes - offset`

The algorithm: `CopySieveBytes(Sieve,n,MaxBytes)`

```
1. kNow = n
2. kEnd = kNow * 2
   IF kEnd >= MaxBytes
      Goto Step 4
   ENDIF
3. CopyMemory(Sieve,Sieve + kNow,kNow)
   kNow = kEnd
   Goto Step 2
4. IF kNow < MaxBytes
      CopyMemory(Sieve,Sieve + kNow,MaxBytes - kNow)
```

```
ENDIF
```

How does it work? We assume, that the first `n` bytes of the memory block are correctly marked (Step 1). We want to fill the memory block with this pattern. Each copy doubles the size of the correct pattern (`kNow`) (Step 2 - Step 3). Therefore we have only $O(ln_2(k))$ copy operations, where k = (length of the memory block in bytes) / n. In Step 4 we fill the rest.

1.5.2 Helper: GetStart

The helper function `GetStart` has three parameters:
`c` .. The "core level"
`SegNr` .. The segment number (the first segment has $SegNr = 1$)
`Prime` .. The prime number.
and returns the value of the first marked number, in relation to `c,SegNr,Prime`

The algorithm: `GetStart(c,SegNr,Prime)`

```
1. IF c = 1 AND SegNr = 1
      kNum = Prime^2
   ELSE
      kNum = Prime^c
   ENDIF
2. IF SegNr = 1
      Start = knum
   ELSE
      Start = mod((LB - 1), kNum)
      IF Start > 0
         Start = kNum - Start - 1
      ENDIF
   ENDIF
3. RETURN Start
```

How does it work? We estimate Start (the starting value) where we begin to mark numbers. In Step 1 we handle the special case $c = 1$ and $SegNr = 1$ and set the temporary variable `kNum`. In Step 2 we compute the offset corresponding with the lower bound LB of the segment.

1.6 Improvement: The Even Prime

Let $SegNr$ be the index of the current segment. The first segment has $SegNr = 1$. We mark all even numbers m of the segment (except for $c = 1$ and $SegNr = 1$ the number 2).

The Function `EvenPrime` has two parameters:
`c` .. The "core level"
`MaxBytes` .. The length of the Segment in Bytes.

The algorithm: `EvenPrime(c,MaxBytes)`

```
1. offset = 0
   kBytes = 1
2. IF SegNr = 1
      SELECT c
         CASE 1
            BitSet(0,1,4,6,8,10,12,14)
            offset = 1
         CASE 2
            BitSet(0,4)
         OTHERWISE
            BitSet(0)
            kBytes = ⌊2^c/8⌋
      ENDSELECT
      GOTO 4
   ENDIF
3. SELECT c
      CASE 1
         BitSet(0,2,4,6)
      CASE 2
         BitSet(0,4)
      CASE 3
         BitSet(0)
      OTHERWISE
         offset = ⌊Rem(LB,2^c)/8⌋
         IF offset > 0
            offset = p^c/8 - offset
         ENDIF
         BitSet(offset * 8)
         kBytes = p^c/8
    ENDSELECT
4. CopySieveBytes(Sieve + offset, kBytes, SegmentBytes - offset)
```

How does it work? First we compute the `offset` (starting value) in bytes and the period `kBytes`. The `offset` depends on c and $SegNr$. In Step 2 we handle $SegNr = 1$. Note, if $c = 1$, then the number 2 is not marked. In Step 3 we handle the other segments. Only if $2^c > 8$ then `kBytes > 1` and therefore the `offset` depends on the lower bound LB of the segment.

1.7 Improvement: The Small Odd Primes

The Function `SmallOddPrimes` has two parameters:
`c` .. The "core level" i.e. p^c
`MaxBytes` .. The length of the segment in bytes.
and return the smallest prime which is not processed.

The algorithm: `SmallOddPrimes(c,MaxBytes)`

```
1. kProd =  2^c
   IF kProd < 8
      kProd = 8
   ENDIF ;
   Prime = 3
   kProd = kProd * Prime^c
2. IF kProd > (SegmentLength / 4)
      GOTO Step 5
   ENDIF
   BeginPeriod = GetStart(c,SegNr,Prime)
   EndPeriod = BeginPeriod + kProd
   k = 0
3. IF (k * Prime^c) + BeginPeriod) >= EndPeriod
      GOTO Step 4
   ENDIF
   BitSet(BeginPeriod + k * Prime^c)
   k = k + 1
   GOTO Step 3
4. BeginByte =  ⌊BeginPeriod/8⌋
   CopySieveBytes(Sieve + BeginBytes, ⌊EndPeriod/8⌋ , MaxBytes - BeginBytes)
   Prime = Nextprime()
   kProd = kProd *  Prime^c
   GOTO Step 2
5. RETURN Prime
```

How does it work? For example, the period (in bytes) of $3^c \cdot 5^c$ is $kBytes = 15^c \cdot \max(8, 2^c)$ (Since we copy bytes we have $\max(8, 2^c)$). In general we have $kBytes = \max(8, 2^c)\Pi_i p_i^c$ with $i > 1$. Up to an appropriate limit of kBytes we can also use the CopySieveBytes method. In Step 1 we set the period (we start with $Prime = 3$). In Step 2 we test the terminate condition and set the bounds of the period. In Step 3 we mark only the period. In Step 4 we copy the pattern of the period to the rest of the memory block and choose the next prime

1.8 Improvement: The Wheel

Our implementation of the wheel is in some sense generic, i.e. it depends on c (the core).

The constants are:
`c` .. The "core level"
`UBPrime` .. An upper bound for the maximal $wheel = \Pi_i p_i^c$ with $p_i < UBPrime$.
`SegLength` .. The length of the segment.
`UpperBound` .. The maximum of the segment.
`LowerBound` .. The minimum of the segment.
`kSqrt` .. Process only primes p with $p \leq \sqrt[\max(2,c)]{UpperBound}$

The variables are:
`From` .. The actual number that is performed.
`FullStep` .. The length of the actual wheel.
`NumList(p)` .. A list of all $p_i \leq p$.
`StepList()` .. A list of numbers $1 \leq m \leq FullStep$ with $m \nmid FullStep$.
`WList(p)` .. A list of numbers $w_i = m_i \cdot p^c$ for all $m_i \in StepList()$, and p is a given prime.

The algorithm: `SingleWheel(Prime)`

```
1. WList(Prime) /* Fill WList() with all primes p_i <= Prime */
   Adder = FullStep * Prime^c
   Limit = SegLength - Adder
2. IF From > Limit
      GOTO Step 3
   ENDIF
   FOR EACH w IN WList()
      BitSet(From + w)
   ENDFOR
   From = From + Adder
   GOTO Step 2
3. /*mark the rest, if any*/
   FOR EACH w IN WList()
      IF (From + w) >= Limit
         GOTO Step 4
      ENDIF
      BitSet(From + w)
   ENDFOR
4. Terminate
```

How it works? In Step 1 we fill some variables and the WList(). Note, the StepList is already filled. In Step 2 we mark all elements of WList added with an offset. In Step 3 we mark the rest.

The algorithm: `AllWheels(StartPrime)`

```
1. kPrime = StartPrime
2. SetNumList(kPrime)
   FullStep = Π_{p∈NumList()} p^c
   SetStepList(FullStep)
3. IF kPrime > kSqrt
      GOTO Step 7
   ENDIF
   From = GetStart(kPrime)
4. IF ((GCD(From,FullStep)= 1) OR (From > UpperBound))
      GOTO Step 6
   ENDIF
5. BitSet(From - LowerBound)
   From = From + kPrime^c
   GOTO Step 4
6. IF From < SegLength
      Wheel()
   ENDIF
   kPrime = NextPrime(kPrime)
   IF kPrime < UBPrime
      GOTO Step 2
   ELSE
      GOTO Step 3
   ENDIF
7. Terminate
```

How it works? Let $c = 1$ and let `NumList = {2,3}`, i.e. all numbers $m \in Segment$ with $\gcd(m, 2 \cdot 3) > 1$ are marked. Thus, the mark schema has the period $2 \cdot 3$, i.e. only the numbers of the form $6k + 1$ and $6k + 5$, $k = 0, 1, ..$, are not marked. Therefore we have `StepList = {1,5}`.
Now we handle a prime $p > 3$. The variable `FullStep` is `FullStep = 6*p`. First we mark all numbers up to the smallest number of the segment with $\gcd(From, FullStep) = FullStep$ in the usual way (in function `AllWheels()`). Then we mark all numbers m of the form $(FullStep \cdot k + 1 \cdot p) + From$ and $(FullStep \cdot k + 5 \cdot p) + From$, with $k = 0, 1, ..$ (in function `SingleWheel()`).

1.9 The Main procedure: SegSieve

Finally the Main procedure, that puts everything together, is given by:

`Sieve` .. A global variable, the address of the memory block
`c` .. The "core level"
`MaxBytes` .. The length of the segment in bytes

The main procedure `SegSieve` has one parameter:
`Sieve` .. The address of the `memoryblock`
`Item` .. The number *Item* has to be in the computed segment.

The algorithm: `SegSieve(Item)`

```
1. SegNr = (Item / SegLength) - 1
   LowerBound = (SegNr - 1) * SegLength
   UpperBound = LowerBound + SegLength + 1
2. EvenPrime(c,MaxBytes)
   Prime = SmallOddPrimes(c,MaxBytes)
   AllWheels(Prime)
```

How does it work? In Step 1 we set appropriate constants. In Step 2 we call the, above defined, procedures and compute the segment.

References

[RICHJ] J. Richstein, *Segmentierung und Optimierung von Algorithmen zu Problemen aus der Zahlentheorie*, 1999, Diss.