Segmented Eratosthenes Sieve Algorithm

Author: Ricson Cheng
11th Grade
ricsoncheng@gmail.com

Advisor : Elizabeth Mabrey
Director of Storming Robots
emabrey@stormingrobots.com

Date of Completion: May, 2014

I. Abstract

The Eratosthenes Sieve algorithm stores an array to represent the primality of each integer. In order to optimize the sieve, the primality of each integer can be stored as a single bit. Also, reusing a smaller array to represent segments of the numbers to be sieved reduces memory.

II. Introduction

The Eratosthenes Sieve creates an array of numbers from 2 to N. Iterating through the list, each number, if not already marked composite, is marked as a prime. Then, all multiples of that number are marked composite. Normally, the algorithm is limited by the available memory. However, much larger sieves can be run if the algorithm is segmented.

III. Design of algorithms

The sieve can be optimized by excluding even numbers since all even numbers excepting 2 are composite. The first item in the array represents 3, the second, 5 and the third, 7. This reduces memory by a half because the array representing the numbers will be size N/2 instead of N.

If composite C has factors A and B, at least one of A or B must be at least the square root of C. Otherwise, since sqrt(C) > A and sqrt(C) > B, then sqrt(C)*sqrt(C) > A*B, then C > A*B, a contradiction. Therefore, sieving with primes up to sqrt(N) as opposed to N is sufficient.

Each byte is composed of eight bits. Typically, an integer takes 4 bytes, or 32 bits, and a char, 1 byte, or 8 bits. By storing a boolean value in each bit of an integer, we can use one integer to store whether 32 integers are prime or not prime.

Even with these optimizations, for large N, it may be impossible to fit the entire array into memory. A segmented sieve allows for this.

In the segmented sieve algorithm, the array of primes that serve as the sieve are created with the traditional sieve. Another integer array is created; this is used to represent each segment. Finally, since the prime 3 will sieve out elements 1, 4, 7, … in the array [1, 3, 5, 7, 9, 11 …] but the elements 2, 5, 8, ... in the array [101, 103, 105, 107, 109, 111 …], a third array is required to store the position of the first multiple of a given prime in the segment.

Each bit in each integer in the segment represents an odd number. With delta as the size of a segment, a loop is used to sieve from 0 to delta-1, then delta to 2*delta-1, then 2*delta to 3*delta-1 until reaching k*delta-1 such that k*delta-1 >=N.

For each iteration, the list of primes is looped over. The initial position of each prime is set by the marker array, then the prime is added over and over to find multiples, which are marked as composite. When next multiple of the prime is greater than delta, the loop terminates and the next prime is used to sieve. After each segment is sieved, the number of primes in the array is counted and then the array is cleared.

Since delta may not evenly divide N, during the last segment, any bits past N are cleared before the primes are counted.

IV. Phases of Development

```
define np(list, len)
        count = 1
        for x from 0 to len-1
                if list[x] is true
                        count += 1
        return count

define sieve(n)
        isprime = array[n*sizeof(char)]
        for x from 0 to n-1
                if x is even
                        isprime[x] = false
                else
                        isprime[x] = true
        isprime[0] = false
        isprime[1] = false
        for x from 3 to n-1 by two
                cutoff = n/x-x
                for i from 0 to cutoff-1 by two
                        isprime[x*(x+i)] = false
        return isprime

define nsieve(n)
        return np(sieve(n),n)
```

First start with the conventional implementation of the sieve.

The sieve function first creates an array. After all the even numbers and 0 and 1 are set to false, a loop is used to iterate through. Then, the np function counts the number of primes in the array.

```
define endremove(limindex, delta, isprime)
        for m from limindex to delta/2
                isprime[m] = 0
```

```
bpi = size(integer)*8

define clearbit(arr, num)
        index = num/bpi
```

```
define segment(numprimes, pplace, delta, isprime, ps)
        for j from 0 to numprimes-1
                marker = pplace[j]
                while marker < delta/2
                        isprime[marker] = 0
                        marker += ps[j]
                marker -= delta/2
                pplace[j] = marker

define pcount(isprime, delta, count)
        for k from 0 to delta
                if isprime[k]
                        count +=1
                        isprime[k] = 0

define segsieve(max)
        count = 0
        sqrtmax = sqrt(max)
        delta = (1.5*sqrtmax/2)*2
        j = sieve(sqrtmax)
        numprimes = np(j,sqrtmax)
        ps = array[(numprimes-1)*sizeof(int)]
        d = 0
        c = 0
        while d < numprimes -1
                if j[c] is True
                        ps[d] = c
                        d += 1
                c += 1
        pplace = array[(numprimes-1)*sizeof(int)]
        for i from 0 to numprimes-2
                pplace[i] = 3*ps[i]/2
        delete(j)
        isprime = array[delta/2]
        set delta/2*size(integer) bytes of isprime to -1
        limindex = ((max%delta)/2)/bpi+1
        mloop = (max+1)/delta+1
        for i from 0 to mloop-1
                segment(numprimes, pplace, delta, isprime,
ps)
                if i == mloop-1
                        endremove(limindex, delta,
isprime)
                pcount(isprime, delta, count)
        return count

define primesieve(n)
        if n < 10000
                return nsieve(n)
        else
                return segsieve(n)
```

```
        bitnum = num%bpi
        arr[index] &= ~(1 << bitnum)

define endremove(limbit, limindex, delta, isprime)
        for m from limindex to delta/bpi/2
                isprime[m] = 0
        for n from limbit to bpi
                isprime[limindex-1] &= ~(1 << n);

define segment(numprimes, pplace, delta, isprime, ps)
        for j from 0 to numprimes-1
                marker = pplace[j]
                while marker < delta/2
                        clearbit(isprime, marker)
                        marker += ps[j]
                marker -= delta/2
                pplace[j] = marker

define pcount(isprime, delta, count)
        for k from 0 to delta/bpi/2
                for m from 0 to bpi
                        if isprime[k] & (1 << m)
                                count += 1
                isprime[k] = -1

define segsieve(max)
        count = 0
        sqrtmax = sqrt(max)
        delta = (1.5*sqrtmax/bpi/2)*bpi*2
        j = sieve(sqrtmax)
        numprimes = np(j,sqrtmax)
        ps = array[(numprimes-1)*sizeof(int)]
        d = 0
        c = 0
        while d < numprimes -1
                if j[c] is True
                        ps[d] = c
                        d += 1
                c += 1
        pplace = array[(numprimes-1)*sizeof(int)]
        for i from 0 to numprimes-2
                pplace[i] = 3*ps[i]/2
        delete(j)
        isprime = array[delta/2/bpi*sizeof(int)]
        set delta/16 bytes of isprime to -1
        limbit = ((max%delta)/2)%bpi+1
        limindex = ((max%delta)/2)/bpi+1
        mloop = (max+1)/delta+1
        for i from 0 to mloop-1
                segment(numprimes, pplace, delta, isprime,
ps)
                if i == mloop-1
                        endremove(limbit, limindex, delta,
isprime)
                pcount(isprime, delta, count)
        return count

define primesieve(n)
        if n < 10000
                return nsieve(n)
        else
                return segsieve(n)
```

Because all the nonprimes less than n must have at least one factor equal to or less than sqrt(n), the segsieve function uses the sieve function to get the primes from 1 to sqrt(n) and stores them in an array.

The delta variable is used to determine the size of each segment. Since even numbers are to be skipped, delta is made divisible by two. If delta is

Define bpi to be the number of bits in each integer.

Because we use every single bits in each integer, we must also let delta be divisible by the number of bits.

In additional to limindex, which marked the index boundary of the sieve, limbit is used the mark the bit position of that boundary.

made too small, extra looping reduces performance, so delta is made to be 50% larger than the largest prime used to sieve. In the case that n is not divisible by delta, we compute limindex to be the index after which the counting should be truncated. The pplace variable is used to keep track of the offsets of the primes. When the segment represents the integers from 0 to 7, the prime three would start at index 0. When the segment represents the integers from 8 to 15, the prime three would start at index 1 since 9 is divisible by 3 The main loop calls segment. Each prime is looped over the segment starting from the marker stored in pplace, then those integers in the array are set to 0. pcount is used to add up all the primes in the segment. If the loop is on it's last iteration, endremove is used to truncate all values beyond the boundary of the sieve. Because the overhead involved makes a segmented sieve slower at small numbers, below 10000, a normal sieve is used instead.	In segment, the function clearbit is called to set the specific bit to zero. The pcount function loops over both the integers in the segment, and the bits in each integer. After each inner loop, the integer is set to -1 which sets all the bits in the integer to 0.

V-Complexity

It may be possible to increase performance of the sieve by changing the segment array type from integer to short, long, long long, or char.

It is not important to optimize the traditional sieve that generates the original primes because it only sieves up to sqrt(N). For any large N, the first sieve is an insignificant portion of the running time.

The size of each segment, delta, can significantly affect performance. Very small values of delta, like 32 or 64, use less memory but reduce performance because large amounts of extraneous looping must be done. Large values of delta reduce the amount of computations needed when keeping track of the prime multiples but require more memory. A typical delta used might be several times sqrt(N).

VI-Performance

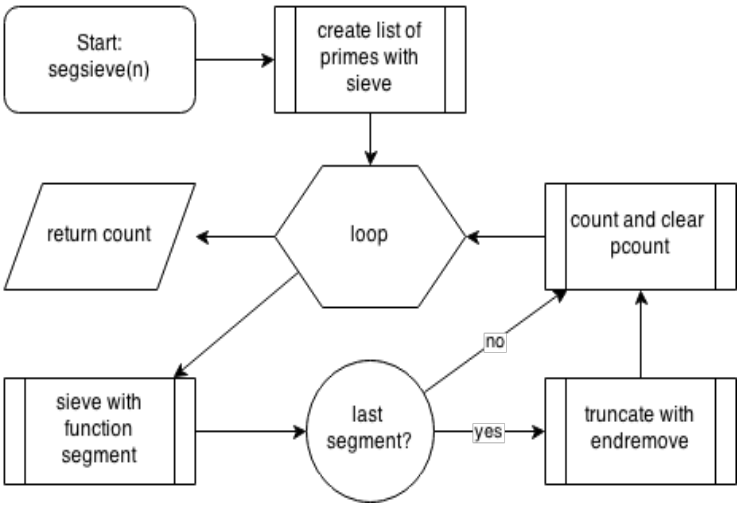
Compiler: Visual Studio Express 2012 /O2

Machine: i7-3632QM at 3.2 GHz 1333MHz, DDR3

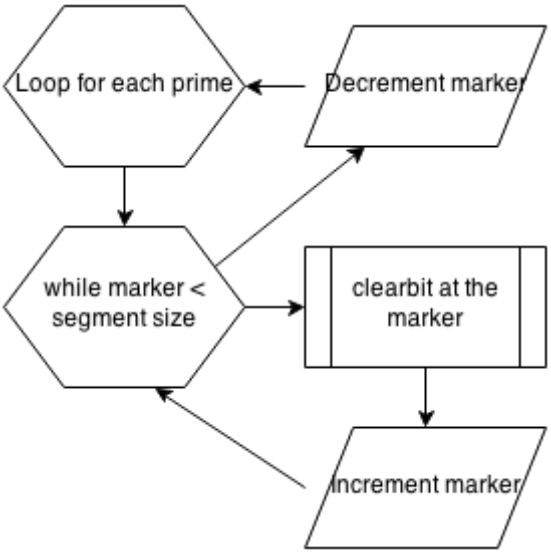
	segmented	*segmented*	*unsegmented*	*unsegmented*
N	*min (s)*	*max (s)*	*min(s)*	*max(s)*
1E5	*0.000*	*0.001*		
1E6	*0.003*	*0.004*		
1E7	*0.035*	*0.036*		
1E8	*0.354*	*0.371*		
1E9	*3.526*	*3.605*		
1E10	*35.895*	*36.475*		
1E11	*362.267*	*377.089*		
1E12	*3763.015*	*3769.901*		
1E13	*38564.199*	*40154.638*		

VII-Diagrams and Tables

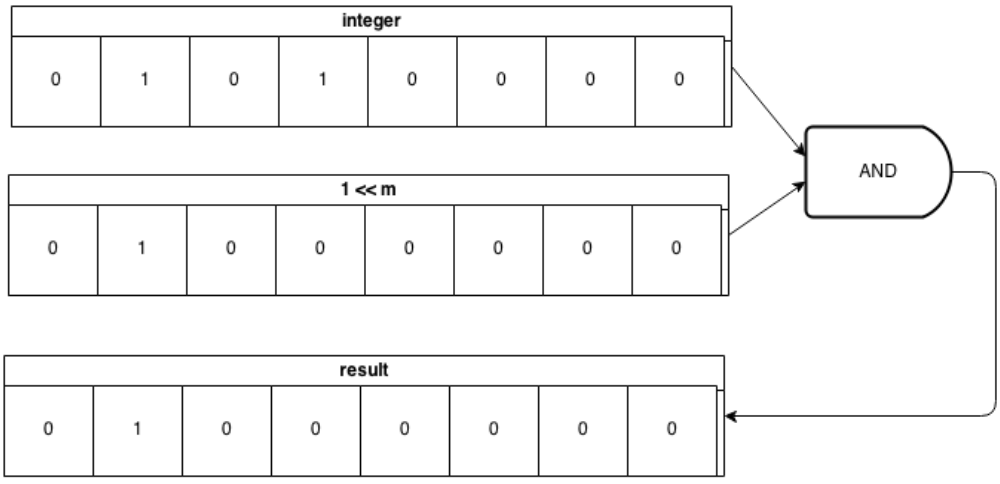
Segsieve function



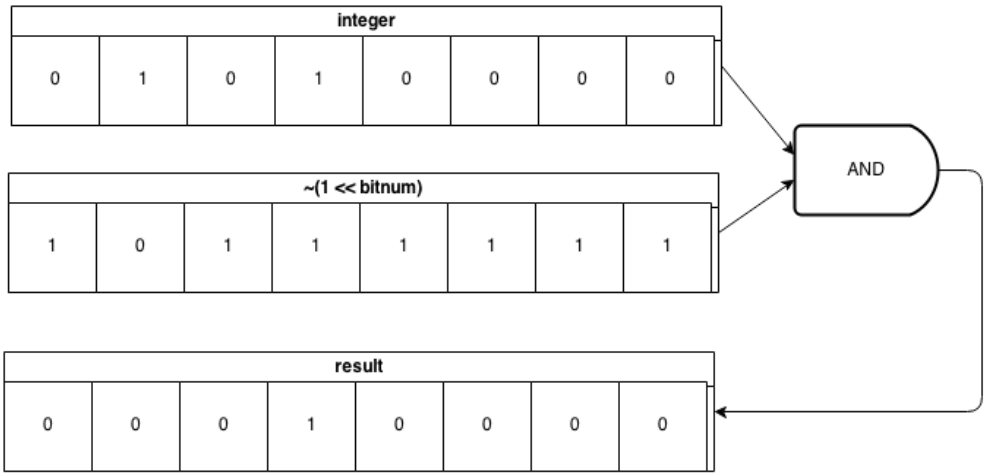
Segment function



Checking a bit



Clearing a bit



VII-Conclusion

By utilizing every bit in an integer, and by segmenting the numbers to be sieved, the eratosthenes sieve can be extended to much larger numbers while using less memory. Future work may further optimize the sieve by setting a variable delta which adjusts to memory constraints and prime size. The segmented sieve is embarrassingly parallel, and multiprocessing should be able to improve performance.

IX-References

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes