Problem 10. Sum all primes below N million

10.1 The natural looping algorithm

If you have a reasonably fast prime testing algorithm, like the one from the overview for problem 7, you could simply loop through all promising candidates (odd numbers or numbers not divisible by either two or three for example) and sum those that turn out to be prime. The code would then resemble

```
limit := 2000000
sum := 5    // we know that 2 and 3 are prime
n := 5
while n <= limit
    if isPrime(n) then sum := sum+n
    n := n+2
    if n <= limit and isPrime(n) then sum := sum+n
    n := n+4
end while
output sum
```

and run in a couple of seconds if the limit is one to five million.

Although that prime testing algorithm is already a vast improvement over dividing by all numbers and still a lot better than dividing by all odd numbers (and in a different league from algorithms which don't stop dividing at $\sqrt{n}$), it is too slow if you have to perform many tests. To find all primes up to one million, testing only numbers not divisible by 2 or 3, over 20 million divisions are carried out, finding the primes up to five million takes over 200 million divisions.

That is an awful lot of work, how can we substantially reduce it?
Apart from some rather advanced new methods, the best answer to that question is over 2000 years old. When you want to find all primes below some limit, use the sieve of Eratosthenes. When you want to test just a handful of not too large numbers, trial division is adequate. However, suppose you'd want to test a few hundred numbers between 10^9 and 10^{10}. Then a mixed strategy is good. First use a sieve to find the primes up to 10^5, then perform trial division by these primes.

10.2 The sieve of Eratosthenes

The basic idea behind this ancient method is that instead of looking for divisors d of n, we mark multiples of d as composites. Since every composite has a prime divisor, the marking of multpiples need only be done for primes. The classical algorithm is

© Project Euler, further distribution without the consent of the author(s) prohibited

Author: daniel.is.fischer

1. Make a list of all numbers from 2 to N.
2. Find the next number p not yet crossed out. This is a prime.
   If it is greater than $\sqrt{N}$, go to 5.
3. Cross out all multiples of p which are not yet crossed out.
4. Go to 2.
5. The numbers not crossed out are the primes not exceeding N.

You only need to start crossing out multiples at p^2, because any smaller multiple of p has a prime divisor less than p and has already been crossed out as a multiple of that. This is also the reason why we can stop after we've reached $\sqrt{N}$.

10.2.1 A first implementation

We begin with an implementation close to the description of the algorithm, which we will improve somewhat after having discussed it. For the list of numbers, we use a boolean array `sieve`, indicating which numbers have been crossed out.

```
limit := 2000000
crosslimit := $\lfloor\sqrt{\texttt{limit}}\rfloor$
sieve := new boolean array [2 .. limit] false
for n := 4 to limit with step 2     // mark even numbers > 2
    sieve[n] := true
end for
for n := 3 to crosslimit with step 2
    if not sieve[n] then    // n not marked, hence prime
        for m := n*n to limit with step 2*n
            sieve[m] := true
        end for
    end if
end for
sum := 0
for n := 2 to limit
    if not sieve[n] then
        sum := sum+n
    end if
end for
output sum
```

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

First, we initialise the array to false because we have not yet crossed out any numbers. In crossing out, we discriminate between even and odd numbers, slightly deviating from the description. After all even numbers > 2 have been crossed out as multiples of 2, there is no point in crossing them out again as multiples of their odd divisors. So when crossing out multiples of odd primes p, we can proceed in steps of $2p$ instead of p, sparing the even multiples. As noted above, all multiples of p less than p^2 will already have been crossed out as multiples of smaller primes, so we need not revisit them. Also, if n is an odd composite, when the loop reaches n, that number and its multiples have already been crossed out, so we need not enter the inner loop then.

10.2.2 Optimising the sieve

An easy optimisation of the sieve is suggested by the discrimination of odd and even numbers. Apart from 2, all that the even numbers do is being crossed out once and occupying space. If we get rid of the even numbers, we save ${}^{(limit-2)}\!/_2$ crossings out and, more importantly, we use only half the memory. That allows sieving larger ranges and reduces cache misses, thus improving performance.

Sieving only odd numbers requires a bit of—simple—index-arithmetics. We let the i^{th} element of the array correspond to the odd number $2i + 1$. Thus, if $p = 2i + 1$, we find that $p^2 = 4i^2 + 4i + 1$ has the index $2i(i + 1)$ and if $m = k \cdot p$ corresponds to j, then $m + 2p$ corresponds to $j + p$. So if the i^{th} index is not yet crossed out, the inner loop starts at $2i(i + 1)$ and proceeds with step $2i + 1$.

The code becomes

```
sievebound := (limit-1) div 2  // last index of sieve
sieve := new boolean array [1 .. sievebound] false
crosslimit := ($\lfloor\sqrt{\texttt{limit}}\rfloor$-1) div 2
for i := 1 to crosslimit
    if not sieve[i] then    // 2*i+1 is prime, mark multiples
        for j:= 2*i*(i+1) to sievebound with step 2*i+1
            sieve[j] := true
        end for
    end if
end for
sum := 2    // 2 is prime
for i := 1 to sievebound
    if not sieve[i] then sum := sum+(2*i+1)
end for
output sum
```

and solves the problem in much less than a second.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer