Eratosthenes sieve in language C

Generally about the sieve of Eratosthenes

The algorithm, developed by Eratosthenes, is used to search for prime numbers (numbers with exactly two divisors). The most natural method of determining whether a number is prime consists in checking its divisibility by all natural numbers smaller than it. Eratosthenes noted that you can limit the number of divisors. He came to the conclusion that having a given natural number N, in order to determine all the prime numbers smaller than N, it is necessary to find the largest prime number that meets the condition $p^2<=N$ (where p - the prime number), and all larger than it to check for divisibility by prime numbers smaller or equal to p.

For example for N=100:
p=7 (because 72=49, and 112=121)
Prime numbers smaller than p:
p1=2
p2=3
p3=5

We check the numbers from 8 to 100. If division gives the remainder other than 0, for each of the above prime numbers, used as divisors, the number is the prime. In addition, from the set of numbers obtained, you have to reject 1, which has only one divisor.
It is easy to notice that the largest of the prime numbers obtained as a result of these actions meets the condition $p_{highest}^2=N^2$. This means that with the results of previous calculations, we can find infinitely many prime numbers.

Adapting the algorithm to the source code

Let's save the above example in the form of source code in C language. The code presents one of the simplest possible solutions.

```
--------------SOURCE 1--------------
   #include<stdio.h>
   main()
   {
   int i;
   int liczp[99];       // array for prime numbers to be found
   int illiczb,nilliczb;
   // variables counting the number of numbers on the array liczp[]

   int liczba;          // a variable to store the number to be checked

   liczp[1]=2;
   // enter the prime numbers at the beginning of the array of numbers

   liczp[2]=3;  //for N=100
   liczp[3]=5;
   liczp[4]=7;
   illiczb=4;
   // enter the current quantity of numbers into the variable

   nilliczb=illiczb;
```

```
for(liczba=8; liczba<=100; liczba++){

if((liczba%liczp[1]!=0) && (liczba%liczp[2]!=0) && (liczba%liczp[3]!=0)
&& (liczba%liczp[4]!=0)){
nilliczb++;
liczp[nilliczb]=liczba;  // if the number to be checked is not divisible
                         //by those initially entered to the array
                         //numbers prime we write it to the array
                         //and increase the variable nilliczb; variable
                         //number (number of divisors) does not change
}
}

for(i=1; i<=nilliczb; i++){
printf("%i) %i\n",i,liczp[i]);
// after the screening is finished, we display the contents of the array
}

getchar();
}
```

Code optimization - a prelude to the search of subsequent ranges

Let's modify the above code so that it can be used to check numbers greater than 100 based on the results of previous ranges.
For N=100 we have only four divisors, so we can afford to list each condition separately. However, the number of divisors increases after checking each successive range. So we have to modify this fragment. Thanks to variable `illiczb` and `nilliczb` we know how many numbers there are on the array. We can use fork loops to divide the prime numbers. This will cause, however, that the division will always be performed as many times as many numbers we have on the array, despite the fact that the checked number will turn out to be divided by one of them. If dividing by 3 or 5 gives the rest of 0 then further dividing is not justified. So let's use while loops. Let's add a variable `flaga` that will regulate the operation of the loop. Let's specify that the loop will be executed when the `flaga!=0`.

Value 0 should be assigned to the `flaga` variable when:

- as a result of dividing we get the rest of 0;
- we divide as many times as many numbers we had on the array before the screening starts;

To enable checking the second of the above conditions, let's enter the variable „`j`”, which will count the loop executions. The number to be checked will be prime, if we make loops as many times as we have on the array of numbers and we do not get the rest of 0 in the last division. Enter the `flagapoz` variable, whose task will be to count the repetitions of loops, in which we have achieved the rest other than 0.
Let's also add a `pauza` function, which will stop the program after displaying each 23 lines of results and allow you to display more results or exit the program.

```
--------------SOURCE 2 --------------
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

long int liczp[9999999];      // increase the size and scope of the array
long int illiczb,nilliczb;
long int liczba;
long int minz, maxz;
//add “minz” and “maxz” variables to operate the "for" loop
// in the "szukaj" function.

short int ilwierszy;
//add the variable "ilwierszy" for the function "pauza".
void init(){

liczp[1]=2;
liczp[2]=3;
liczp[3]=5;
liczp[4]=7;
illiczb=4;
nilliczb=illiczb;
printf("%i\n%i\n%i\n%i\n",liczp[1],liczp[2],liczp[3],liczp[4]);
ilwierszy=4;
// write into the array the initial prime numbers for N=100
// and print them out on the screen
}

void pauza()
{
char znak;
if(ilwierszy==23){
printf("[Enter] - next page of results; [k] - end of program ");

if((znak=getchar())!='k'){

ilwierszy=0;
}
else{
exit(0);
}

}
}

void szukaj()
{
short int flaga;
long int flagapoz, j;

illiczb=nilliczb;

for(liczba=minz; liczba<=maxz; liczba+=2)
{
flaga=1;
flagapoz=0;
j=0;

while(flaga!=0)
{
```

```
j++;
if(j>illiczb || liczba%liczp[j]==0){
flaga=0;
}
else{
flagapoz++;
}
}

if(flagapoz==illiczb){
nilliczb++;
liczp[nilliczb]=liczba;
printf("%i\n",liczba);
ilwierszy++;

pauza();
}
}
}

main()
{

init();
minz=9;
maxz=100;
szukaj();
minz=101;
maxz=10000;
szukaj();
minz=10001;
maxz=100000000;
szukaj();

getchar();
}
```

Even higher ranges

Some numbers from the set from 100000001 to 10^{16} go beyond the range of variables in which the values to be checked are stored. The commands will be executed 10^8*10^8, i.e. 10^{16} times. To search the range of numbers up to 10^{32} you will need to add more loops.

The problem of performing calculations on the checked number requires a more complex solution. Let's use the method used in banking to check the correctness of bank account numbers. Following this method, a modulo from a given number can be calculated by dividing the number to be checked into parts. When we do this, we calculate the modulo from the first part, and the result is added at the beginning of the next part of the number. Calculate the modulo from the received number and place the result again at the beginning of the next part of the number.

Example:

We are looking for the result of the following action: 21357648 % 7
Let's divide the number into parts:
21 – 35 – 76 – 48
Let's calculate the modulo from the first fragment of the number:
21 % 7 = 0
Add the result at the beginning of the second fragment and calculate the modulo:
035 % 7 = 0 (count as 35 % 7)
Let's repeat the above steps until the final result is obtained:
076 % 7 = 6
648 % 7 = 4

We finally got 4, so the rest of dividing 21357648 by 7 is 4.

This method can be used for the needs of the program by modifying it accordingly. The problem, apart from the form of recording numbers, is also the place for their recording. The number of first numbers to be found from the range up to 10^{16} is so large that you should consider saving them outside the program (for example in an external file).
The largest number discovered so far consists of more than 7 million digits (the previous one contained more than 6 million digits, and the previous one more than 4 million). For the first person (or team) who finds the first number longer than 10 million digits, there is a prize of $100,000 (the prize for the first person who found the first number more than a million digits was $50000).

Currently, the largest first number consists of 22 338 618 digits and is more than 5 million digits higher than the previous one. It is the so-called Mersen number - that is, it has a form:

$$2^{74207281} - 1$$