Problem Set 3 Exercise #11: Sieve of Eratosthenes

Reference: Week 7 Lecture notes

Learning objective: One-dimensional array

Estimated completion time: 45 minutes

Problem statement:

In mathematics, the Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer. It works efficiently for the small primes (below 10 million). It was created by Eratosthenes, an ancient Greek mathematician (see Wikipedia: Sieve of Eratosthenes for stories and animation).

The idea behind this method is to use a pre-processing step to first identify all prime numbers up to a number *n*, after which prime queries become a simple matter of array lookup. For the interest of memory, the maximum number you would check in your program is 10,000 (otherwise you might run out of memory). The algorithm works as follows:

i. Step 1 (declare): create an integer array `arr` of size 10,001.

ii. Step 2 (initialize): for each index `m` from 0 to 10,000:
```
    arr[m] = 1
```

iii. Step 3 (pre-process): for each index `m` from 2 to the square root of 10,000:
```
    if arr[m] is 1, then
        for each index n from 2*m to k*m, where k*m is the largest number ≤ 10,000
                arr[n] = 0
```

iv. When the above pre-processing finish, we can tell whether a positive integer `m` (`m` ≥ 2) is a prime or not through the following checking:
```
    if arr[m] is 1
        then m is prime
    else arr[m] must be 0
        which means m is not a prime
```

While the algorithm looks complicated, the idea is quite simple:

1. We start by assuming all numbers as prime, then we look at position 2, we know that 2 is prime, so in Step 3 we set all multiples of 2 (**k***2) as not prime by assigning `arr[k*2]` as 0.
2. We then look at the next prime in the array, which in this case is 3. So we set all multiples of 3 as not prime (`arr[k*3]` = 0).

3. Similarly, we look for the next prime 5 (since 4 has been marked as not prime already), and again set all multiples of 5 as not prime.
4. We keep repeating the above steps until the next prime is equal or greater than the square root of 10,000 - while we could continue repeating till 10,000, mathematically there is no need to do so (Why?).
5. Once the pre-processing is complete, we simply look at **`arr[m]`** to find out if **`m`** is prime - if **`arr[m]`** is 1, then **`m`** is prime, otherwise not.

Write a program **sieve.c** to implement the above algorithm. Your program will keep reading positive integers smaller than 10,000 and tell whether each input number is a prime or not. Your program should stop when 0 is entered by user.

Sample run #1:

```
Pre-processing ready, enter values (0 to stop):
92
not prime
93
not prime
94
not prime
95
not prime
96
not prime
97
prime
98
not prime
99
not prime
100
not prime
101
prime
102
not prime
0
```

Sample run #2:

```
Pre-processing ready, enter values (0 to stop):
991
prime
0
```