Sieve of Eratosthenes and Implementation with Python

Cryptography and Information Security-103A-CSCSN-ISA-ECRYPT, 2022L

Andreas Kalavas *K-6669*, and Anastasios Ektoroglou *K-6667*

***Abstract*—As far as we know today, an ancient Greek introduced the first algorithm for the computation of primes: Eratosthenes of Kyrene (276-194 BC). He was a high-ranking scholar in Alexandria and a director of its famous library, containing the complete knowledge of ancient mankind. He and others studied the essential astronomical, geographical, and mathematical questions of their time: What is the perimeter of the Earth? Where does the Nile come from? How can one construct a cube containing twice the volume of another given one?**

***Index Terms*—prime number, algorithm, sieve, insert**

I. Introduction

The Sieve of Eratosthenes is a method for finding all primes up to a given natural n. This method works well when n is relatively small, allowing us to determine whether any natural number less than or equal to n is prime or composite. A prime number is a natural number that has exactly two distinct natural number divisors: the number 1 and itself.

Prime numbers are critical for the study of number theory. Nearly all theorems in number theory involve prime numbers or can be traced back to prime numbers in some way. Prime numbers are also important for the study of cryptography.

A table up to n is a list of all primes between 1 and n. It begins as follows:

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...]$$

II. The algorithm

To see why this algorithm is called a "sieve," imagine numbers are rocks, where the shape of each rock is determined by its factors. Even numbers, which are multiples of 2, have a certain distinctive shape, multiples of 3 have a slightly different shape, and so on. Now suppose we have a bowl with adjustable holes in the bottom. To figure out which numbers are prime, put a bunch of rocks in the bowl and adjust the holes to match the shape for multiples of 2. When we shake the bowl, all the rocks that are multiples of 2 will fall through the holes. Next, adjust the holes so they match the shape for multiples of 3, and shake again so the multiples of 3 fall out. If the goal is to have only prime numbers in the bowl we need to keep repeating the adjusting and shaking steps until all the composite numbers have fallen out.

As a result of taking a more systematic approach, we are guaranteed that when we're done we will have every prime number within a specified range.

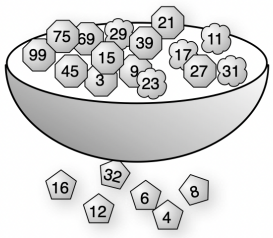

Fig. 1. The first time the bowl is shaken, even numbers (multiples of 2) fall out.

III. Algorithm analysis

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1) Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Fig. 2. Example with n=100.

2) Initially, let p equal 2, the smallest prime number.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Fig. 3. First prime number.

3) Enumerate the multiples of p by counting in increments of p from $2p$ to n, and mark them in the list (these will be $2p$, $3p$, $4p$, ...; the p itself should not be marked).

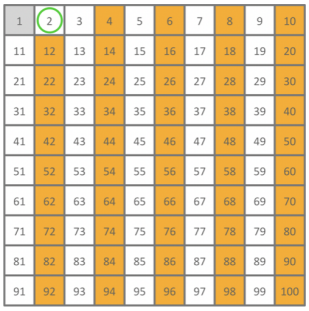

Fig. 4. Counting increments of 2.

4) Find the smallest number in the list greater than p that is not marked. if there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

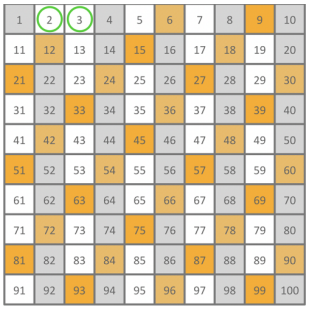

Fig. 5. The smallest number greater than 2 that is not marked.

5) When the algorithm terminates, the numbers remaining not marked on the list are all the primes below n.

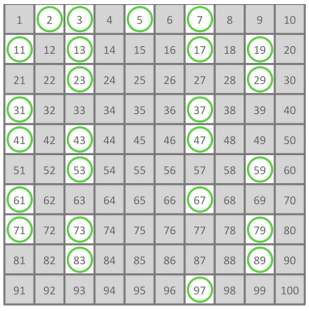

Fig. 6. We could keep going like this all the way to 100.

The main idea here is that every value given to p will be prime because if it were composite it would be marked as a multiple of some other, smaller prime. The algorithm can be further optimised if we consider the fact that a number's divisors go in pairs (except its square root if it is a whole number). This means that in order to find the prime numbers up to n, we need to execute the inner loop of the algorithm only for the prime numbers that are less or equal to the square root of n.

IV. Time complexity of the algorithm

The main part of the algorithm consists of two nested loops; The outer one is executed n times and the inner one n/p times, for each value of p. Thus, an upper bound for the running time of the algorithm is the harmonic sum:

$$\sum_{p=2}^{n} = n/2 + n/3 + n/4 + ... + n/n = O(nlogn)$$

In fact, the algorithm is more efficient, because the inner loop will be executed only for prime numbers p, smaller or equal to the square root of n. It can be shown that the running time of the algorithm is only $O(nloglogn)$, a complexity very near to $O(n)$.

V. Python Implementation

```python
import numpy as np
import math
import time

print("_____\n")
print("\t\t\tWe will find all the prime numbers less than N\n")
time.sleep(1)

                ###.     Assign values in our tools    .###

n = int(input("Give the number n: "))
sq = math.floor(math.sqrt(n))
marked_numbers = np.zeros(n+1)
primes = []

                ###.     Processing Part    .###

for i in range(n + 1):
    if(i < 2):
        continue

    if(marked_numbers[i] == 0):
        primes.append(i)

        if i <= sq + 1:
            num = 2 * i

            while num <= n:
                marked_numbers[num] = 1
                num = num + i

                ###.     Results    .###

time.sleep(1)
print(f"\nPrimes that are smaller or equal than n: {primes}\n")
print("_____\n")
```

We wrote an algorithm that checks for all numbers between 2 and n.

A. Imports

```python
import numpy as np
import math
import time
```

- First we import the library ***numpy*** which is the universal standard for working with numerical data in Python, that will help us because we need to use an array to mark our numbers that are not prime.
- Second import is the library ***math*** which provides access to the mathematical functions and will help us to compute the square root of our given number.
- Finally, the last library named ***time*** which is a module to handle time-related tasks, and will help us to represent our data with a delay (only for a nice presentation of our data).

B. Values assignment

```
9   n = int(input("Give the number n: "))
10  sq = math.floor(math.sqrt(n))
11  marked_numbers = np.zeros(n+1)
12  primes = []
```

- In the first place, we ask the user to give the number text n that will be the maximum value for our calculations.
- Secondly, we need to calculate the square root of the given number n and we assign it to the variable sq.
- With usage of ***numpy*** we make an array of zeros and we assign it to the variable $marked\_numbers$.
- Finally, we declare an empty array that we will add every prime number inside.

C. Calculations

```
18  for i in range(n + 1):
19      if(i < 2):
20          continue
21
22      if(marked_numbers[i] == 0):
23          primes.append(i)
24
25          if i <= sq + 1:
26              num = 2 * i
27
28              while num <= n:
29                  marked_numbers[num] = 1
30                  num = num + i
```

Fig. 7. Algorithm part.

```
18  for i in range(n + 1):
19      if(i < 2):
20          continue
```

1) In the first place, we start with a loop which starts from 0 in range $n+1$ and we need to start from number two which is the first prime number, so we put a ***for*** function to check it.

```
22      if(marked_numbers[i] == 0):
23          primes.append(i)
```

2) We check if we have a marked number (we mark a number in the table with 1) in the array of zeros. If the number is not marked, then we add it to the variable ***primes*** which includes all the prime numbers for our program.

```
25          if i <= sq + 1:
26              num = 2 * i
```

3) After that, we need to check if the number is smaller or equal to the square root of the given number n, if it's $True$ then the variable num takes the value of the counter i multiplied by 2 and go forward for while loop...

```
28              while num <= n:
29                  marked_numbers[num] = 1
30                  num = num + i
```

4) Because we have to check every number smaller or equal to n, while the pointer variable num is smaller or equal to the given number, we mark every number that is multiplied by the number that we found in the first place with ***if*** function, in our code we do it by adding num with the pointer that we stacked i.

D. Printing the results

```
34  time.sleep(1)
35  print(f"\nPrimes that are smaller or equal than n: {primes}\n")
36  print("_____________________________________________________________________________________\n")
37
```

- We are using a delay of 1 second...
- And then we print out the elements of ***primes***, which include all the prime numbers that we find between 2 and n.

VI. Testing our Code

We tested our code by giving $n = 100$ as the examples above and we took the results:

```
~/Desktop/IMMI/PLs/Python/Sch_pjs ❯ python3 ecryp_project.py
_____________________________________________________________________________________

                    We will find all the prime numbers less than N

Give the number n: 100

Primes that are smaller or equal than n: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

_____________________________________________________________________________________
~/Desktop/IMMI/PLs/Python/Sch_pjs ❯                                                 7s
```

REFERENCES

[1] https://www.researchgate.net/publication/230595538$_{T}he_{S}ieve_{o}f_{E}ratosthenes--_{H}ow_{F}ast_{C}an_{W}e_{C}ompute_{a}_{P}rime_{N}umber_{T}able$

[2] https://study.com/academy/lesson/the-sieve-of-eratosthenes-lesson-quiz.html

[3] https://ix.cs.uoregon.edu/ conery/eic/python/EiCPP$_{c}hapter3.pdf$

[4] https://numpy.org/doc/stable/user/absolute$_{b}eginners.html$

[5] https://www.programiz.com/python-programming/time

[6] https://www.w3schools.com/python/python$_{m}ath.asp$

[7] https://en.wikipedia.org/wiki/Sieve$_{o}f_{E}ratosthenes/$

[8] https://www.geeksforgeeks.org/sieve-of-eratosthenes/

[9] https://brilliant.org/wiki/sieve-of-eratosthenes/

[10] https://brilliant.org/wiki/prime-numbers/

[11] https://gofiguremath.org/number-theory/prime-numbers/

[12] Laaksonen Competitive Programmer's Handbook