Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 3:
Study and Empirical Analysis of Different Implementations for Eratosthenes Sieve

Elaborated:
st. gr. FAF-21X Cristian Brinza

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

Algorithm Analysis

Objective

Analyze different implementations for Eratosthenes Sieve, and make conclusions on the obtained results based on empirical analysis.

Task

Implement 5 algorithms for Eratosthenes Sieve. Decide an input format for all these algorithms and decide a comparison metric. Analyze empirically the algorithms and present the results using tables or graphs. Compare results and deduce conclusions based on them.

Theoretical Notes

the Sieve of Eratosthenes is an efficient and well-known algorithm for identifying all prime numbers within a given limit. The process involves generating a list of numbers from 2 up to the limit and then systematically eliminating all multiples of each prime number, beginning with 2. Any unmarked number that remains on the list after all iterations is confirmed to be prime. This approach is a useful tool for researchers, mathematicians, and programmers who require an accurate and quick way to identify prime numbers.

Comparison Metric

Using the empirical approach, we will analyze and compare the algorithms by their execution runtime in seconds, for different values of ***n***.

Input Format

In this laboratory work, we will analyze 5 different implementations of Eratosthenes Sieve. For input format I decided to use 3 values for n: 1000, 10000 and 20000. The smallest number is 1000 because for smaller values the algorithms are very fast and runtime is close to 0 seconds. The biggest number is 20000 because for bigger values I run out of memory.

Implementation

All the 5 algorithms were implemented using Python.

The execution runtime for each experiment was measured using the time library in python. Because the computer has various tasks besides our code compilation, the execution time may differ from time to time.

Code link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab3

<u>Algorithm 1</u>

This algorithm works by marking all multiples of a prime number (starting from this number multiplied by 2 up to n) as non primes.

Multiples are computed by adding the prime to the last calculated multiple.

This algorithm is described by the following pseudocode:

```
def eratosthenes_sieve_1(n):
    # Create a list c that will hold boolean values indicating if each number up to n
is prime
    c = [True for i in range(n+1)]
    # 1 is not prime, so set it to False
    c[1] = False
    # Start at 2, the first prime number
    i = 2
    # Iterate through each number up to n
    while i <= n:
        # If the current number i is prime, mark all multiples of i as not prime
        if c[i] == True:
            j = 2 * i
            while j <= n:
                c[j] = False
                j += i
            # Move to the next number
        i += 1
    # Create a list primes to hold all prime numbers found
    primes = []

    # Iterate through each number up to n
    for i in range(2, n+1):
        # If the current number is prime, add it to the list of primes
        if c[i] == True:
            primes.append(i)

    # Return the list of primes
    return primes
```

To compute the time complexity, we firstly construct the table:

Number of iterations

1

1

n – 1

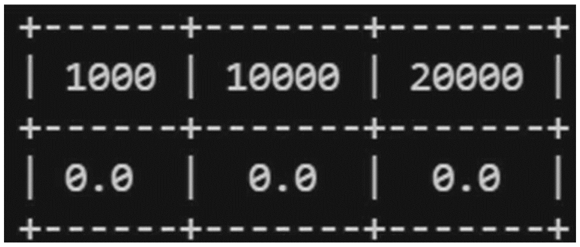
n - 1

n / log n (Number of primes up to n) n/2 + n/3 + ... + n/(n/2) < n log n n log n

n log n

n - 1

This algorithm has a time complexity of O(n log n). Thus, we have this runtime for n = 1000, 10000 and 20000:



```
+------+-------+-------+
| 1000 | 10000 | 20000 |
+------+-------+-------+
| 0.0  |  0.0  |  0.0  |
+------+-------+-------+
```

The output is 0 seconds for all these 3 values of n, because the algorithm is very fast, and it takes less than 0.00000000000000001 seconds of runtime.

Algorithm 2

This algorithm works in the same principle as the previous one, except the fact that we mark as non primes the multiples of all numbers in the array, and not only for the prime ones.

This algorithm is described by the following pseudo code:

```
# This is an implementation of the Sieve of Eratosthenes algorithm to find all prime
numbers up to n.
def eratosthenes_sieve_2(n):
    # create a boolean array c, where c[i] is True if i is prime, and False otherwise
    c = [True for i in range(n+1)]
    # 1 is not considered prime, so set it to False
    c[1] = False
    # start at 2, the first prime number
    i = 2
    while i <= n:
        # mark all multiples of i as composite (not prime)
        j = 2 * i
        while j <= n:
            c[j] = False
            j += i
        # move to the next number
        i += 1
    # initialize empty list to store prime numbers
    primes = []
    for i in range(2, n+1):
        if c[i] == True: # if c[i] is True, then i is prime, so append it to the list
of primes
            primes.append(i)
    # return list of prime numbers up to n
    return primes
```

To compute the time complexity, we firstly construct the table:

Number of iterations

1

1

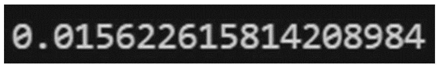
n – 1

n – 1

n/2 + n/3 + ... + n/(n/2) < n log n n log n

n log n n - 1

The time complexity for this algorithm is O(n logn). Thus, we have this runtime for n = 1000, 10000 and 20000:



```
0.015622615814208984
```

Algorithm 3

This algorithm works by finding all the multiples by a less optimized mehod. For each prime, we check all the numbers (starting from prime+1 and up to n) if they are divisible by the current prime, and if so we mark them as non primes.

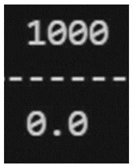
The logic of the algorithm is represented by the following pseudocode:

```
# This is a function that uses the Sieve of Eratosthenes algorithm to find all prime
numbers up to n
def eratosthenes_sieve_3(n):
    # Create a list of booleans to represent whether a number is prime or not
    c = [True for i in range(n+1)]
    # Set 1 to be a non-prime number
    c[1] = False
    # Start with the first prime number, 2
    i = 2
    # Loop through all numbers up to n
    while i <= n:
        # If the current number is prime
        if c[i] == True:
            # Mark all multiples of i as non-prime
            j = i + 1
            while j <= n:
                if j % i == 0:
                    c[j] = False
                j += 1
        i += 1
    # Create a list of all prime numbers up to n
    primes = []
    for i in range(2, n+1):
        if c[i] == True:
            primes.append(i)
    # Return the list of prime numbers
    return primes
```

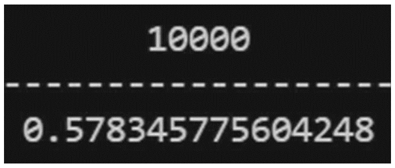
To compute the time complexity, we firstly construct the table:

Number of iterations	
1	
1	
n – 1	
n - 1	
n / log n	(Number of primes up to n)
(n – 3) *	n / log n (Worst case when j = 3)
(n – 3) *	n / log n (Worst case when j = 3)
(n – 3) *	n / (log n * i)
(n – 3) *	n / log n (Worst case when j = 3)
n – 1	

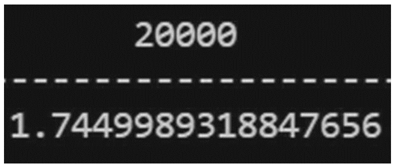
The time complexity for this algorithm is O(n^2/log n), which makes it a less efficient algorithm. Thus, we have this runtime for n = 1000, 10000 and 20000:



```
1000
------
0.0
```



```
      10000
------------------
0.578345775604248
```

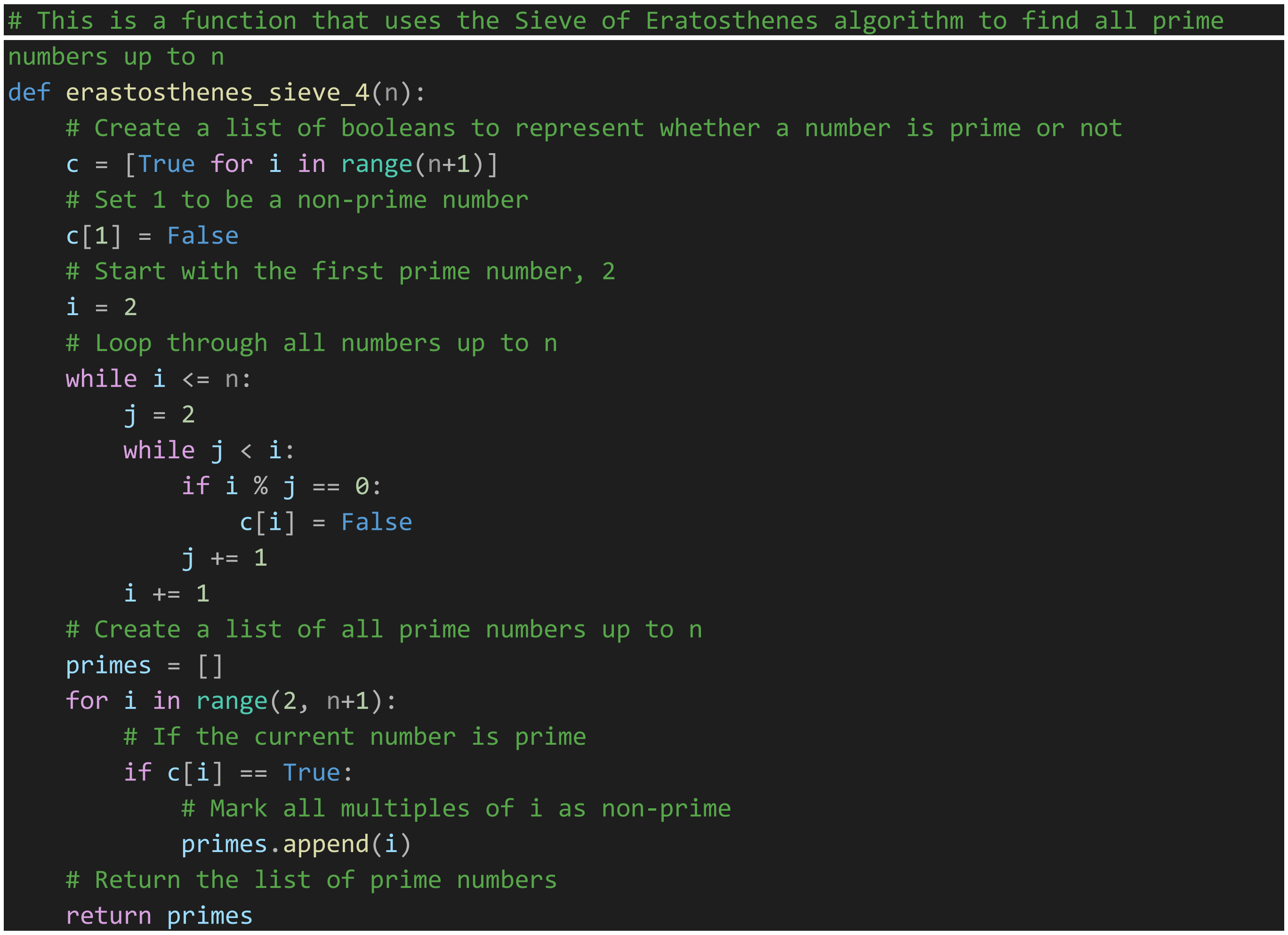


```
      20000
------------------
1.7449989318847656
```

<u>Algorithm 4</u>

This agorithm works by dividing each number to all values starting from 2 and up to the predecessor of that number. If this number is found as a multiple of a value smaller than it and bigger than 2, then this number is marked as a non prime.

The pseudocode describing the algorithm:



```python
# This is a function that uses the Sieve of Eratosthenes algorithm to find all prime
numbers up to n
def erastosthenes_sieve_4(n):
    # Create a list of booleans to represent whether a number is prime or not
    c = [True for i in range(n+1)]
    # Set 1 to be a non-prime number
    c[1] = False
    # Start with the first prime number, 2
    i = 2
    # Loop through all numbers up to n
    while i <= n:
        j = 2
        while j < i:
            if i % j == 0:
                c[i] = False
            j += 1
        i += 1
    # Create a list of all prime numbers up to n
    primes = []
    for i in range(2, n+1):
        # If the current number is prime
        if c[i] == True:
            # Mark all multiples of i as non-prime
            primes.append(i)
    # Return the list of prime numbers
    return primes
```

To compute the time complexity, we firstly construct the table:

Number of   iterations

1

1

n - 1

n – 1

(n – 1) * (n-2) (Worst case when i == n)

(n – 1) * (n-2) (Worst case when i == n)

(n – 1) * (n-2) (Worst case. Best case: 0)

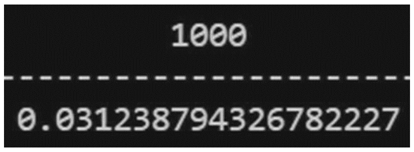
(n – 1) * (n-2) (Worst case when i == n)

n - 1

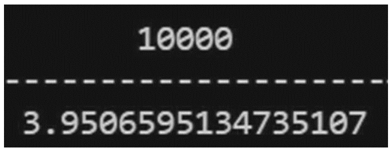
We can see that the inner loop runs ***n - 1*** times for worst case when **i = n**. The best case is 0 iterations when

i = n and so ***j < i*** is false. Thus, for ***tau(n)*** for inner loop we have ***0 <= tau(n) <= n – 2.***

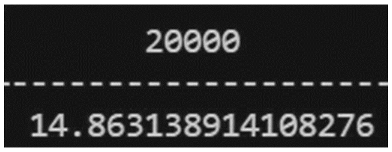
The time complexity for this algorithm is O(n^2), which makes it a less efficient algorithm, like the



```
1000
--------------------
0.031238794326782227
```



```
10000
------------------
3.9506595134735107
```



```
20000
------------------
14.863138914108276
```

previous algorithm. Thus, we have this runtime for n = 1000, 10000 and 20000:

<u>Algorithm 5</u>

This algorithm works in the same way as the previous one, except a single line of code:

while j <= sqrt(i)

while in Algorithm 4 we have:

while j <= i

That little optimization affects the time optimization in a positive way. If ***i*** is a non prime, then it must have a factor ***j*** bigger than 2 and smaller than square root of ***i***. Suppose ***i*** is not a prime number and can be written as ***i* = *a* * *b***, where ***a*** and ***b*** are both integers greater than 1. If both ***a*** and ***b*** were greater than the square root of ***i***, then their product ***i*** would be greater than ***i***. Therefore, at least one of ***a*** and ***b*** must be less than or equal to the square root of ***i***.

The pseudocode describing this algorithm:

```
#This is an implementation of the Sieve of Eratosthenes algorithm to find prime numbers
up to n
def eratosthenes_sieve_5(n):
    # Create a boolean list to keep track of which numbers are prime
    c = [True for i in range(n+1)]
    # 1 is not a prime number, so mark it as False
    c[1] = False

    # Loop over all numbers from 2 to n
    i = 2
    while i <= n:
        # Loop over all numbers from 2 to the square root of i
        j = 2
        while j <= i**0.5:
            # If i is divisible by j, mark it as not prime
            if i % j == 0:
                c[i] = False
            j += 1
        i += 1

    # Collect all prime numbers
    primes = []
    for i in range(2, n+1):
        if c[i] == True:
            primes.append(i)

    return primes
```

To compute the time complexity, we firstly construct the table:

Number of iterations

1

1

n - 1

n - 1

(n – 1) * $\sqrt{n}$ (Worst case when i == n)

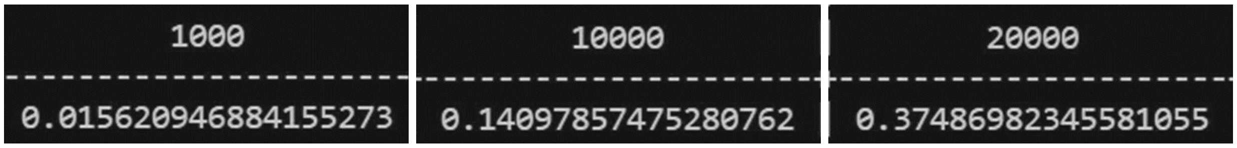
(n – 1) * $\sqrt{n}$ (Worst case when i == n)

(n – 1) * $\sqrt{n}$ (Worst case)

(n – 1) * $\sqrt{n}$ (Worst case when i == n)

n - 1

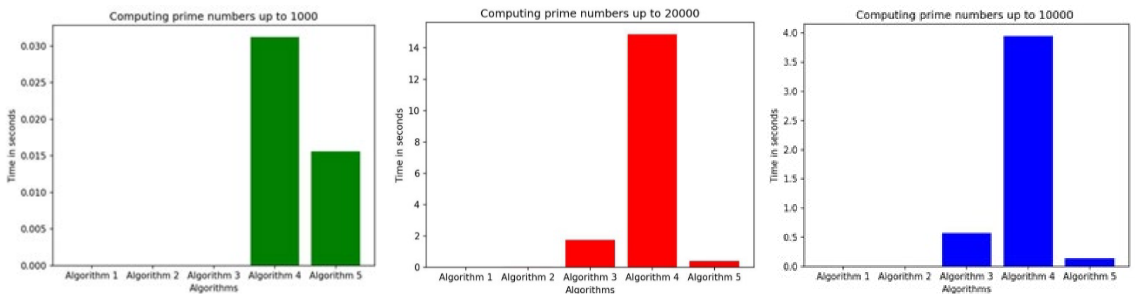
The time complexity for this algorithm is O(n$\sqrt{n}$), which makes it more efficient than the previous algorithm because of a little optimization trick. Thus, we have this runtime for n = 1000, 10000 and 20000:



```
        1000                       10000                      20000
-----------------------  -----------------------  -----------------------
0.015620946884155273     0.14097857475280762      0.37486982345581055
```

Results

The results for runtime give a predictable result based on computed time complexity for each algorithm:



Even though Algorithm 3 and Algorithm 4 have a time complexity of O(n^2), the 4th Algoritm has a much longer runtime. Also, we can see how the little optikization trick for Algorithm 5 makes it much faster in terms of runtime than the Algorithm 4.

Conclusions

We analyzed 5 implementations of the Eratosthenes Sieve algorithm and compared their runtime for 3 values of n and through Empirical Analysis were compared based on the execution time for 3 different values of ***n***.

The fastest algorithms in terms of runtime are Algorithm 1 and 2, having time complexities of O(n logn). Algorithm 3 and 4 are the slowest, having time complexities of O(n^2⁄log n) and O(n^2). The Algorithm 5 represents an optimized version of Algorithm 4, and gives a much faster runtime results, and a better time complexity of O(n$\sqrt{n}$).

Algorithms 1 and 2 were the fastest with time complexity O(n logn), while Algorithms 3 and 4 were slower with time complexity O($n2$⁄log n) and O($n2$). Algorithm 5 was an optimized version of Algorithm 4 with faster runtime and better time complexity of O(n$\sqrt{n}$). Empirical analysis is important for comparing algorithm efficiency and optimization can significantly improve performance.

Code link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab3