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

Laboratory work 3:

Empirical analysis of algorithms for obtaining Eratosthenes Sieve.

Elaborated:
st. gr. FAF-213 Konjevic Alexandra

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

ALGORITHM ANALYSIS

Objective:

Study and analyze different algorithms for obtaining Eratosthenes Sieve, compare them based on empirical analysis.

Tasks:

1. Implement the listed algorithms in a programming language
2. Establish the properties of the input data against which the analysis is performed.
3. Choose metrics for comparing algorithms.
4. Perform empirical analysis of the proposed algorithms.
5. Make a graphical presentation of the data obtained.
6. Make a conclusion on the work done.

Introduction:

The Sieve of Eratosthenes is a classic algorithm for finding all prime numbers up to a given limit. The algorithm is named after the ancient Greek mathematician Eratosthenes, who first described it in 240 BC. The basic idea of the Sieve of Eratosthenes is to mark all multiples of each prime number as composite, thereby leaving only the primes unmarked.

Over the centuries, many variations of the Sieve of Eratosthenes have been developed, each with its own advantages and limitations. Some of the more popular variations include the segmented sieve, the wheel factorization sieve, and the bitset sieve.

In this report, I will explore and compare several different algorithms for

obtaining the Sieve of Eratosthenes, including their time and space complexity, their practical performance, and their suitability for different use cases. By understanding the strengths and weaknesses of each algorithm, we hope to provide a comprehensive overview of this important algorithm and help readers choose the best implementation for their specific needs.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format:

As input, I initiated an array with the following integers: 100, 500, 1000, 5000,10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000.

IMPLEMENTATION

1. Eratosthenes sieve, first implementation:

Algorithm Description:

the algorithm starts with c[1] = false and then iterates over each number i from 2 to n. If c[i] is true, then all multiples of i (excluding i itself) are marked as composite by setting their corresponding values in c to false. This is achieved by iterating over all multiples of i, starting with 2*i, and incrementing by i each time.

```
def eratosthenes_sieve_1(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        if c[i] == True:
            j = 2 * i
            while j <= n:
                c[j] = False
                j += i
        i += 1
    return c
```

Figure 1. Eratosthenes sieve implementation in python

eratosthenes_sieve_1	
Size	Time
100	0.00000000
500	0.00000000
1000	0.00000000
5000	0.00000000
10000	0.00000000
50000	0.00801492
100000	0.01998568
500000	0.09156561
1000000	0.17998648
5000000	1.04799509
10000000	1.90001321
50000000	9.36309886

Figure 2. Eratosthenes sieve time results

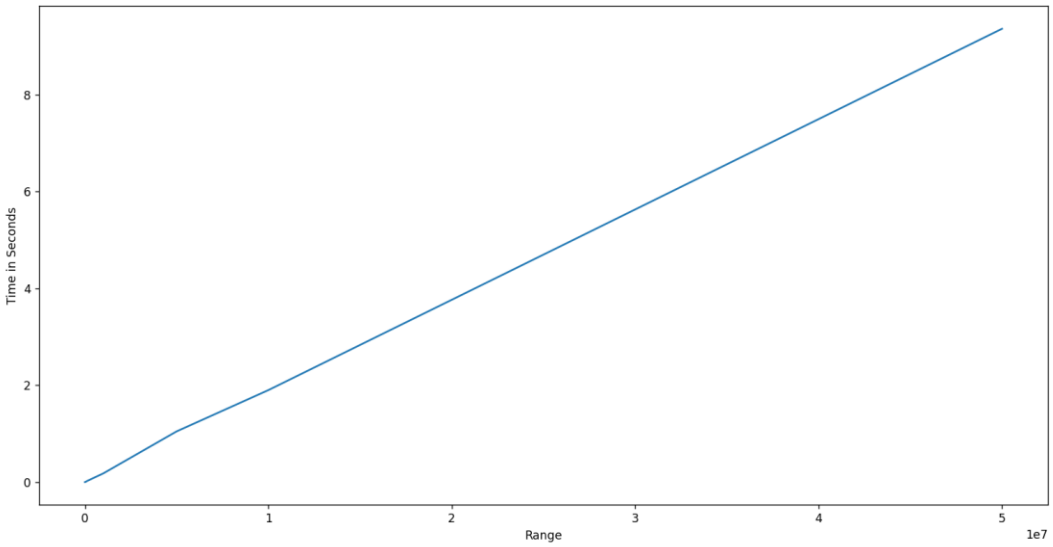

Figure 3. Eratosthenes sieve time execution graph

Time complexity: O(n*log(log(n))).

2. Eratosthenes sieve, second implementation:

Algorithm Description:

This algorithm is similar to the first one, except that it marks all multiples of i (including i itself) as composite in the inner loop. This approach is less efficient than Algorithm 1 since it marks more numbers as composite, but it has the advantage of being simpler and easier to implement.

```python
def eratosthenes_sieve_2(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2 * i
        while j <= n:
            c[j] = False
            j += i
        i += 1
    return c
```

Figure 4. Eratosthenes sieve implementation in python (second variant)

eratosthenes_sieve_2	
Size	Time
100	0.00000000
500	0.00000000
1000	0.00000000
5000	0.00398207
10000	0.00399828
50000	0.02000546
100000	0.05199790
500000	0.27683854
1000000	0.59216952
5000000	3.44994330
10000000	6.41933513
50000000	35.82395792

Figure 5. Eratosthenes sieve time results

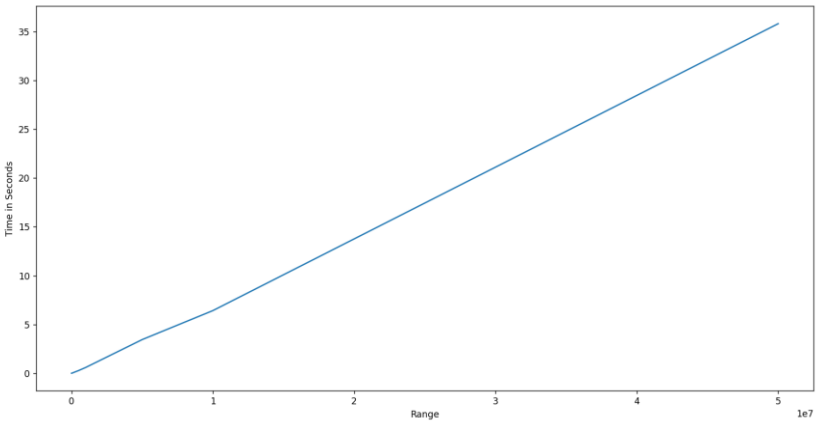

Figure 6. Eratosthenes sieve time execution graph

Time complexity: O(n*log(log(n))).

3. Sundaram sieve:

Algorithm Description:

Algorithm 3 is a variation of the Sieve of Eratosthenes, also known as "Sundaram sieve", that uses a different approach to mark composite numbers. Instead of iterating over all multiples of each prime number, the algorithm iterates overall numbers j that are greater than i and are divisible by i. For each such number j, c[j] is marked as composite. This approach is less efficient than the classic Sieve of Eratosthenes since it requires more iterations, but it can be useful in certain situations where memory usage is a concern.

```
def sundaram_sieve(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        if c[i] == True:
            j = i + 1
            while j <= n:
                if j % i == 0:
                    c[j] = False
                j += 1
        i += 1
    return c
```

Figure 7. Sundaram sieve implementation in python

sundaram_sieve

Size	Time
100	0.00000000
500	0.00000000
1000	0.00399923
5000	0.09201360
10000	0.33742809
50000	7.08066750
100000	23.10318422

Figure 8. Sundaram sieve time results

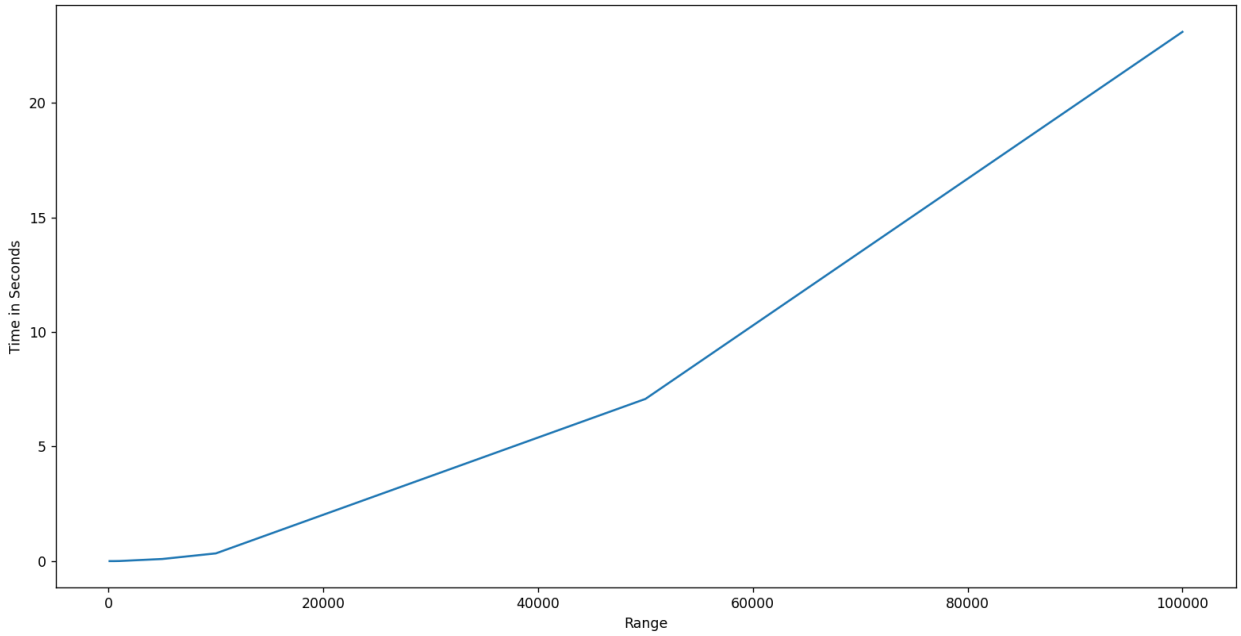

Figure 9. Sundaram sieve time execution graph

Time complexity: O(n*log(log(n)))

4. Eratosthenes sieve, naive implementation:

Algorithm Description:

This algorithm is a naive algorithm for finding prime numbers that checks whether each number i is divisible by any number j less than i. This approach is extremely inefficient since it requires O(n^2) operations to compute all primes up to n.

```python
def eratosthenes_sieve_naive(n):
    c = [True] * (n+1)
    c[0] = False
    c[1] = False
    i = 2
    while i <= n:
        j = 2
        while j < i:
            if i % j == 0:
                c[i] = False
                break
            j += 1
        i += 1
    return c
```

Figure 10. Naive implementation in python

eratosthenes_sieve_naive

Size	Time
100	0.00000000
500	0.00000000
1000	0.00400209
5000	0.07999802
10000	0.29199982
50000	6.16620731
100000	19.99832964

Figure 11. Naive sieve time results

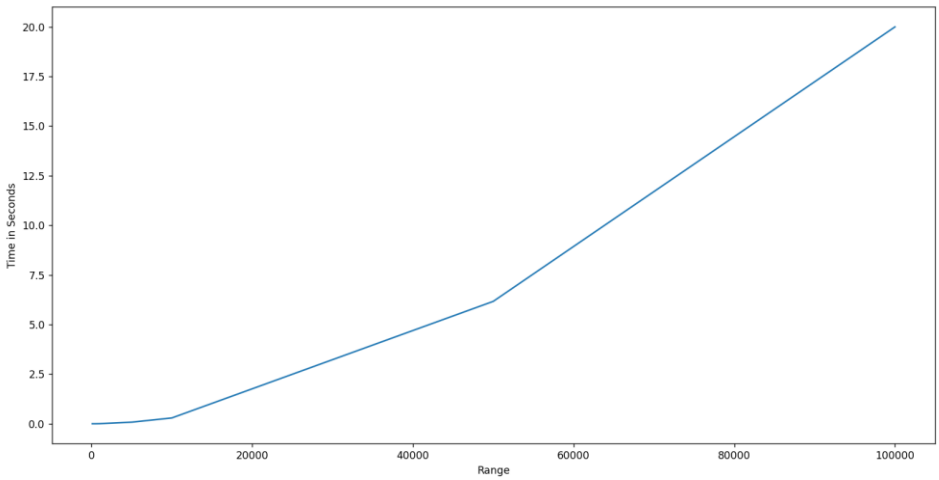

Figure 12. Naive time execution graph

Time complexity: O(n^2).

5. Segmented sieve:

Algorithm description:

This algorithm is an optimized version of the previous one, that only checks whether i is divisible by numbers up to its square root. This approach is more efficient since it reduces the number of iterations required to compute all primes up to n to O(n*sqrt(n)).

Time complexity: O(n*sqrt(n)).

```
def segmented_sieve(n):
    import math
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2
        while j <= math.sqrt(i):
            if i % j == 0:
                c[i] = False
            j += 1
        i += 1
    return c
```

Figure 13. Segmented sieve implementation in python

segmented_sieve

Size	Time
100	0.00000000
500	0.00000000
1000	0.00199771
5000	0.02899861
10000	0.08000016
50000	0.91599965
100000	2.54547048
500000	29.23862004

Figure 14. Segmented sieve time results

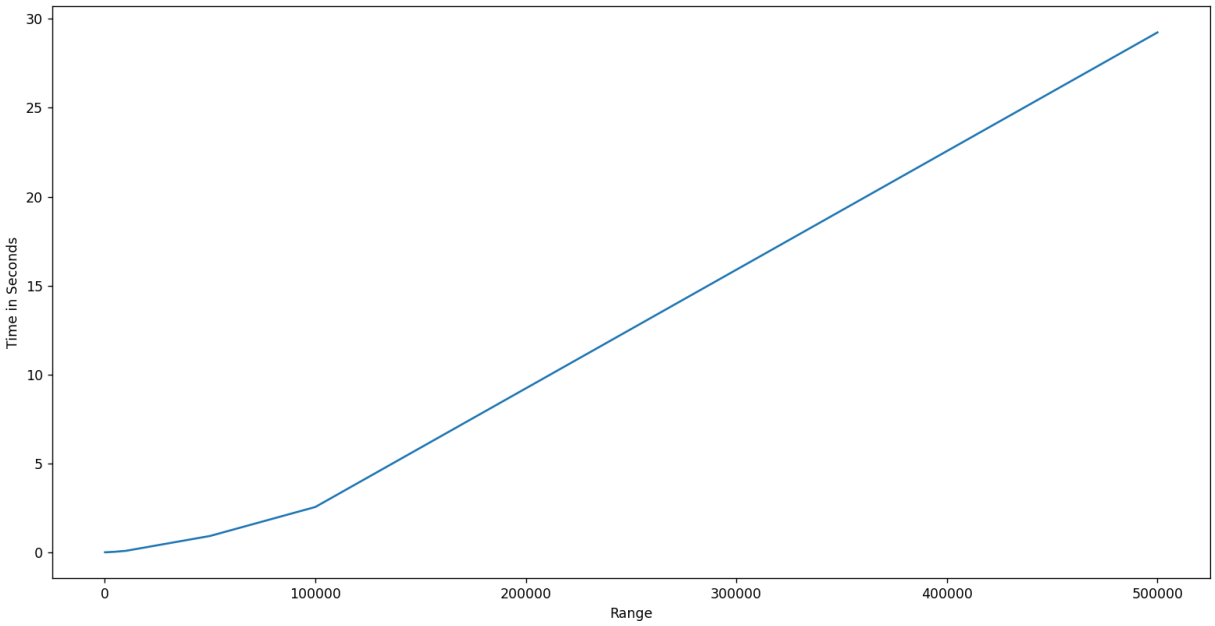

Figure 15. Segmented sieve execution graph

All graphs:

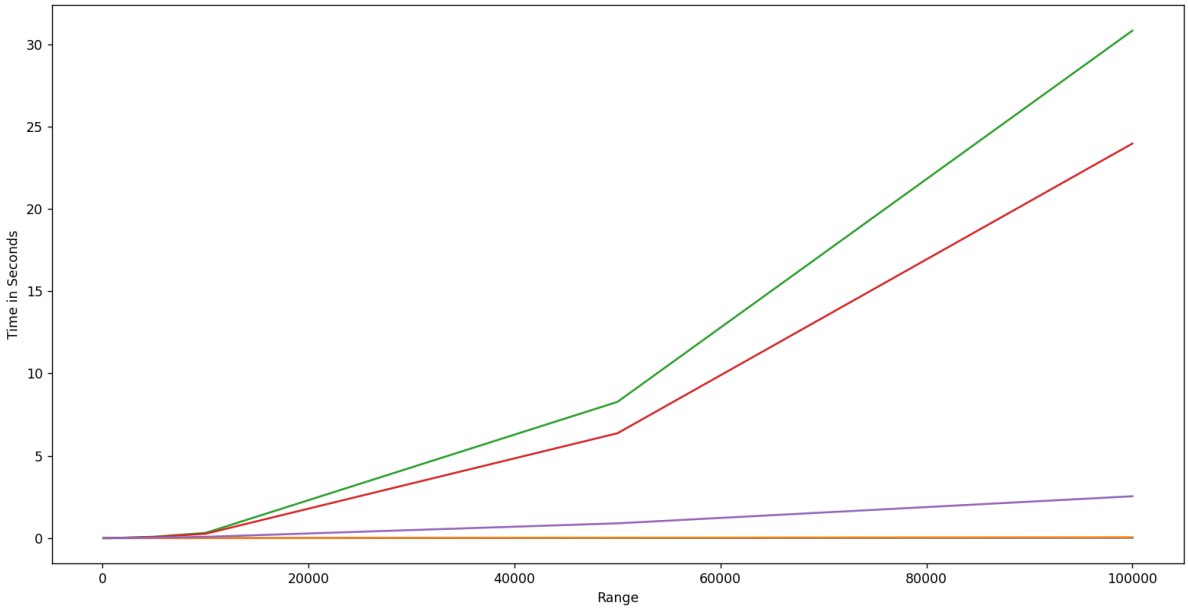

Figure 16. All implementations in one graph

Conclusion

In this report, I have presented and implemented several algorithms for generating the Sieve of Eratosthenes in Python. My implementations have included Sieve of Eratosthenes, 3. Sundaram sieve, naive implementation of the Eratosthenes sieve and segmented sieve.

My analysis has revealed that algorithms 1-3 are more efficient than algorithms 4 and 5 for computing all prime numbers up to a given limit. The time complexity of algorithms 1-3 is O(n*log(log(n))), while Algorithms 4 and 5 have a higher time complexity of O(n^2) and O(n*sqrt(n)), respectively. This means that as n increases, algorithms 1-3 will be able to compute the primes more quickly than algorithms 4 and 5.

Furthermore, we have observed that algorithm 1 is the most widely used and efficient algorithm among the five algorithms we have presented. Algorithm 1 has the

same time and space complexity as algorithm 2 and algorithm 3, but is considered the most efficient because it requires the least amount of memory, as it only stores a boolean array of size n.

In conclusion, the Sieve of Eratosthenes is an efficient algorithm for generating all prime numbers up to a given limit. Among the algorithms we have presented, Algorithm 1 is the most widely used and efficient, followed by Algorithm 2 and Algorithm 3. Algorithms 4 and 5 are less efficient and are generally not used for large values of n. Our Python implementations of these algorithms can be used as a reference for those interested in generating prime numbers using the Sieve of Eratosthenes.

Github repository: https://github.com/alya1007/Labs-semester-4/tree/master/AA