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

REPORT

Laboratory work no.3
Study and empirical analysis of Sieve of Eratosthenes algorithms.

Elaborated:
st. gr. FAF-213 Gutu Dinu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

1. Analysis and study of the given Sieve of Eratosthenes algorithms.
2. Empirical analysis of the aforementioned algorithms.

Tasks

1. Implement the algorithms listed above 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.

Theoretical notes

Empirical analysis is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction

The ancient procedure known as the **sieve of Eratosthenes** in mathematics can be used to locate all prime integers up to a certain limit. It accomplishes this by repeatedly labeling as composite (i.e., not prime) the multiples of each prime, beginning with the first prime number, 2. The series of numbers used to construct the multiples of a particular prime are those that begin with that prime and have a constant difference between them equal to that prime. This is the main difference between the sieve and the sequential trial division test for each candidate number's prime divisibility. The remaining unmarked integers are primes once all the multiples of each identified prime have been marked as composites.

The earliest known reference to the sieve is in Nicomachus of Gerasa's *Introduction to Arithmetic*, an early 2nd cent. CE book which attributes it to Eratosthenes of Cyrene, a 3rd cent. BCE Greek mathematician, though describing the sieving by odd numbers instead of by

primes.

A prime number is a natural number that has exactly two distinct natural number divisors: the number 1 and itself. 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*).
2. Initially, let *p* equal 2, the smallest prime number.
3. Enumerate the multiples of *p* by counting in increments of *p* from 2*p* to *n*, *and* mark them in the list (these will be 2*p*, 3*p*, 4*p*, ...; the *p* itself should not be marked).
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.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below *n*.

The key insight is that every number for p must be a prime since if it were composite, it would be labeled as a multiple of a smaller prime. Please take note that some of the numbers may have several markings (e.g., 15 will be marked both for 3 and 5). As an improvement, since all the smaller multiples of p will have already been marked at that time, it is sufficient to start marking the numbers in step 3 at p^2. This indicates that when p^2 is bigger than n, the algorithm may end at step 4.

Another improvement is to designate just odd multiples of p by first listing only odd integers (3, 5,..., n), and then counting in increments of 2p in step 3. The original algorithm includes this. Wheel factorization can be used to generalize this by creating the initial list from only numbers that are coprime with the first few primes and not just from odds (i.e., numbers that are coprime with 2) and counting in the correspondingly adjusted increments so that only such multiples of p are generated in the first place that are coprime with those small primes.

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, the algorithms will receive an array of numbers ranging from n/10 to n with a step of n/10. In this specific case I choose n to be 10000, thus the step will 1000 and the elements will go from 1000 to 2000, 3000 and so forth.

IMPLEMENTATION

Algorithm 1

Algorithm Description:

```
c[1] = False
i = 2
while i <= n:
    if c[i]:
        j = 2 * i
        while j <= n:
            c[j] = False
            j = j + i
    i = i + 1
```

Implementation

```
def alg_1(n):
    c = [True] * (n + 1)
    c[1] = False
    i = 2
    while i <= n:
        if c[i]:
            j = 2 * i
            while j <= n:
                c[j] = False
                j = j + i
        i = i + 1
    return c
```

Figure 1. Algortihm nr.1

Results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 1	0.0002	0.0003	0.0005	0.0007	0.0009	0.001	0.0012	0.0014	0.0016	0.0017

Figure 2. Algortihm nr.1 results

It can be seen that the algorithm is extremely fast, having a time complexity of T(n*log(log(n))). The time complexity is found using the following steps:

1. The array of length n+1 is initialized with True values in O(n) time.
2. The loop from 2 to n iterates n-1 times, and each iteration performs an O(n/i) operation to mark all multiples of i as composite, where i is a prime number.

Therefore, the time complexity of this loop is:

T(n/2 + n/3 + n/5 + n/7 + ...) <= T(n log log n).

The above inequality holds because the sum of the reciprocals of the primes up to n is bounded by

log log n. This is known as the Prime Number Theorem. Finally, the array of boolean values is returned in O(1) time. Thus, the overall time complexity of the algorithm is O(n log log n).

Graphs

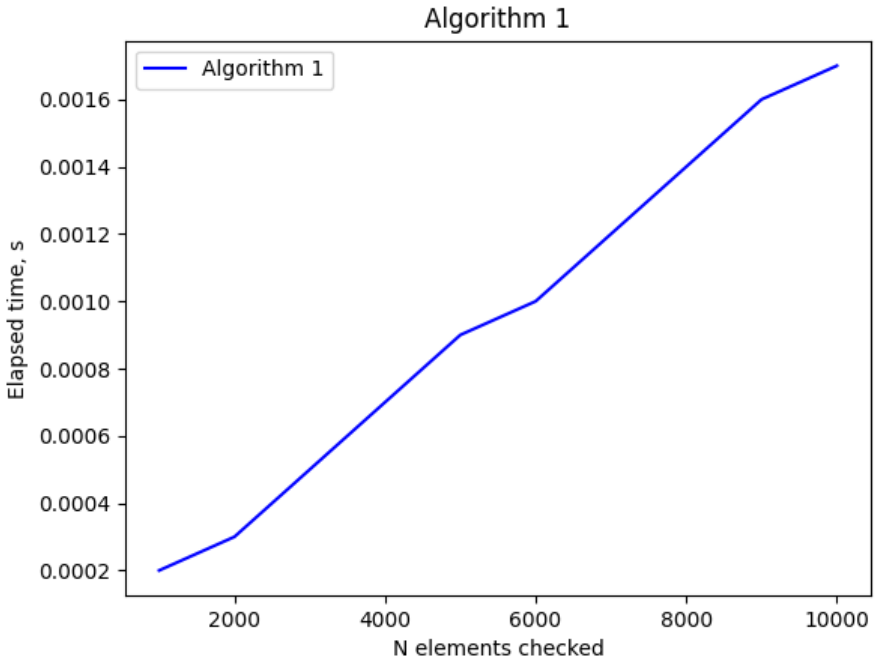

Figure 3. Algortihm nr.1 graph

Algorithm 2

Algorithm Description:

```
c[1] = False
i = 2
while i <= n:
    j = 2 * i
    while j <= n:
        c[j] = False
        j = j + i
    i = i + 1
```

Implementation

```python
def alg_2(n):
    c = [True] * (n + 1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2 * i
        while j <= n:
            c[j] = False
            j = j + i
        i = i + 1
    return c
```

Figure 4. Algortihm nr.2

Results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 2	0.0003	0.0007	0.0012	0.0016	0.0021	0.0027	0.0031	0.0037	0.0043	0.0054

Figure 5. Algortihm nr.2 results

This algorithm again seems to be very efficient, having a time complexity of T(n*log(n)). The time complexity is found through:

The outer loop runs for n - 1 times, so its time complexity is T(n). Its inner loop runs for approximately n/2 + n/3 + n/4 + ... + n/n times, which is equal to n * (1/2 + 1/3 + 1/4 + ... + 1/n). This sum is known as the harmonic series and it is approximately equal to the natural logarithm of n or more formally upper bounded by the natural logarithm. Therefore, the time complexity of the inner loop is O(n log n)

Graphs

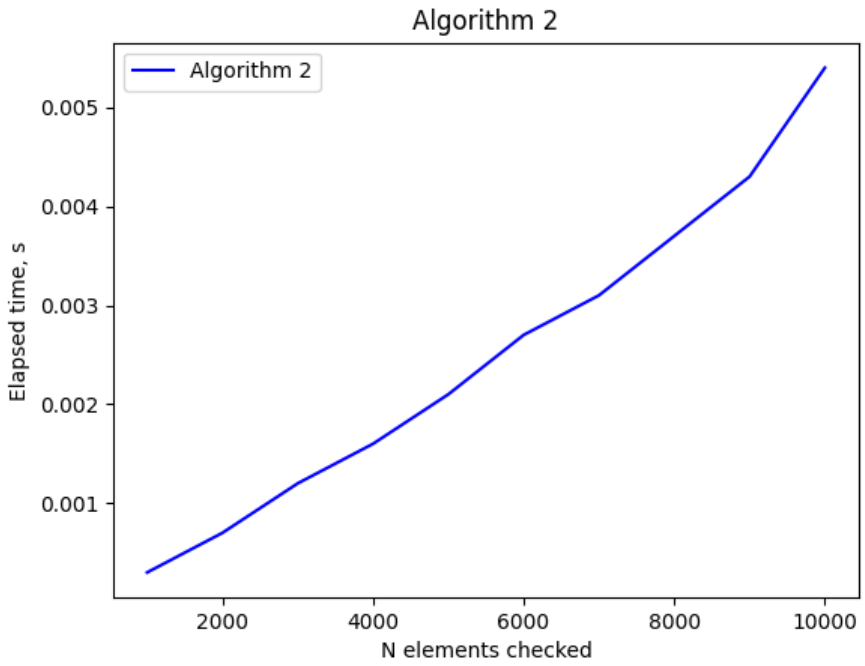

Figure 6. Algortihm nr.2 graph

Algorithm 3

Algorithm Description:

```
c[1] = False
i = 2
while i <= n:
    if c[i]:
        j = i + 1
        while j <= n:
            if j % i == 0:
                c[j] = False
            j = j + 1
    i = i + 1
```

Implementation

```
def alg_3(n):
    c = [True] * (n + 1)
    c[1] = False
    i = 2
    while i <= n:
        if c[i]:
            j = i + 1
            while j <= n:
                if j % i == 0:
                    c[j] = False
                j = j + 1
        i = i + 1
    return c
```

Figure 7. Algortihm nr.3

Results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 3	0.0068	0.0224	0.052	0.1126	0.1343	0.1744	0.2237	0.2978	0.3672	0.422

Figure 8. Algortihm nr.3 results

Algorithm number 3 also seems to be slower by a margin compared to first two, having a time complexity of T(n^2). The time complexity of this algorithm can be calculated as follows:

- The outer while loop runs n-1 times, starting from 2 and incrementing by 1 in each iteration. Hence, its time complexity is O(n).
- For each value of i in the outer loop, the inner while loop runs n/i - 1 times (i.e., the number of multiples of i less than or equal to n), starting from i+1 and incrementing by 1 in each iteration. Hence, the total number of iterations of the inner loop is:
  (n/2 - 1) + (n/3 - 1) + ... + (n/n - 1)

This is an arithmetic series with n-2 terms, where the first term is (n/2 - 1) and the common difference is 1/i. Using the formula for the sum of an arithmetic series, we get:

(n-2) * [(n/2 - 1) + (n/n - 1)] / 2

Simplifying, we get:

(n-2) * [(n-1)/2] / 2

This is equivalent to (n^2 - 3n + 2)/4. Therefore, the time complexity of the inner loop is O(n^2). All other statements in the algorithm (i.e., initializing the array c and setting its elements to False) take constant time, and can be ignored in the analysis. Therefore, the overall time complexity of the algorithm is O(n^2)

Graphs

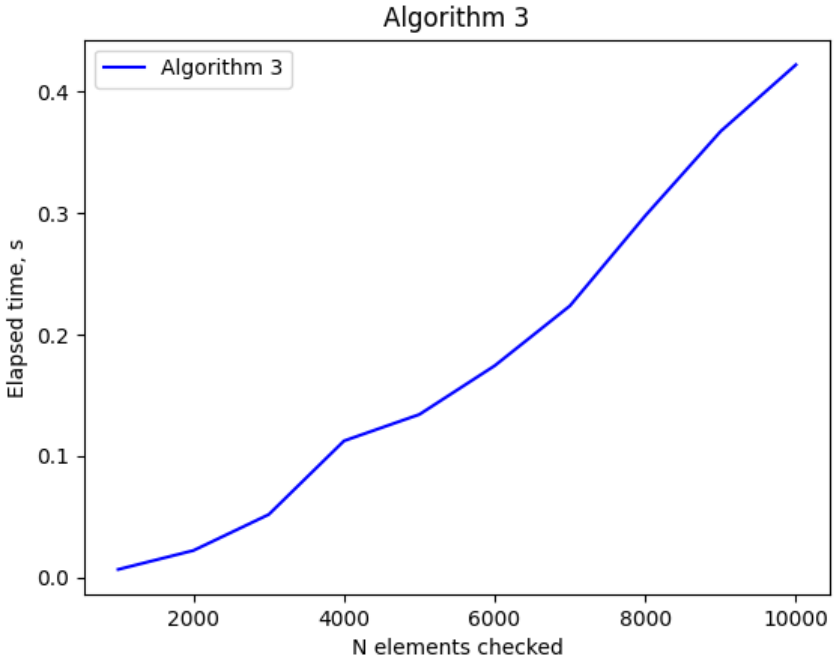

Figure 9. Algortihm nr.3 graph

Algorithm 4

Algorithm Description:

```
c[1] = False
i = 2
while i <= n:
    j = 2
    while j < i:
        if i % j == 0:
            c[i] = False
        j = j + 1
    i = i + 1
```

Implementation

```python
def alg_4(n):
    c = [True] * (n + 1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2
        while j < i:
            if i % j == 0:
                c[i] = False
            j = j + 1
        i = i + 1
    return c
```

Figure 10. Algortihm nr.4

Results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 4	0.0312	0.1878	0.3161	0.5774	0.8712	1.344	1.6489	2.0849	2.6359	3.2608

Figure 11. Algortihm nr.4 results

It can be easily observed that the fourth algorithm is by far the slowest out of the previous ones, having a time complexity of T(n^2). The given code implements a naive algorithm to calculate all prime numbers up to n.

The outer loop runs from i = 2 to n, and for each value of i, the inner loop runs from j = 1 to j = i-1 to check if i is divisible by any number in that range. If i is divisible by any j, then it is not a prime number, and c[i] is set to False. The time complexity of the algorithm depends on the number of times the inner loop is executed. For each i, the inner loop runs i-1 times. Therefore, the total number of iterations of the inner loop is:

1 + 2 + 3 + ... + (n-1) = n*(n-1)/2 = O(n^2)

Thus, the time complexity of the algorithm is O(n^2).

Graphs

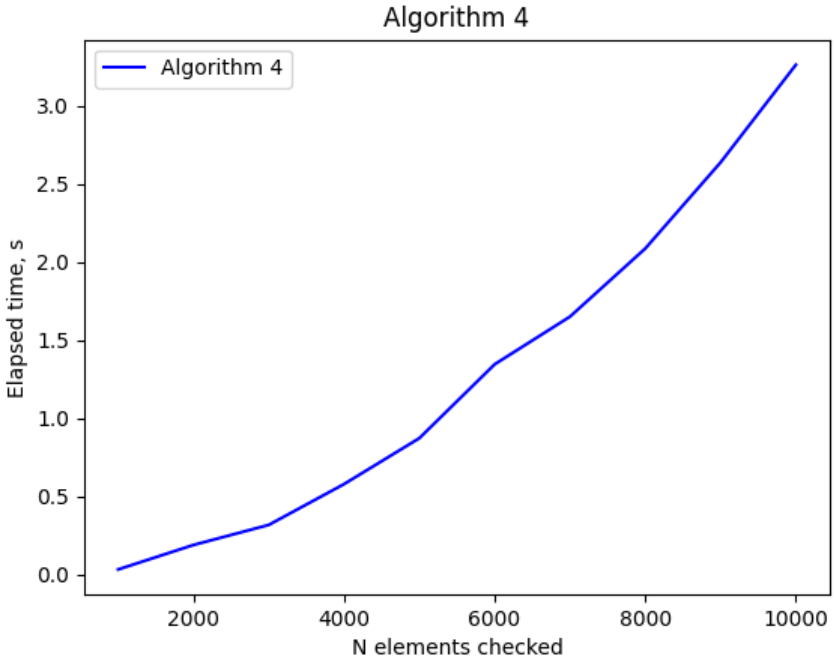

Figure 12. Algortihm nr.4 graph

Algorithm 5

Algorithm Description:

```
c[1] = False
i = 2
while i <= n:
    j = 2
    while j <= math.sqrt(i):
        if i % j == 0:
            c[i] = False
        j = j + 1
    i = i + 1
```

Implementation

```
def alg_5(n):
    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 = j + 1
        i = i + 1
    return c
```

Figure 13. Algortihm nr.5

Results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 5	0.0041	0.0151	0.0223	0.0332	0.0484	0.0635	0.0818	0.0993	0.1122	0.1354

Figure 14. Algortihm nr.5 results

The above algorithm at first glance looks somewhat efficient but we can see that it increases very fast in the time that it takes for computing primes up to the nth number, thus having a time complexity of T(n^3/2). Here is the analysis of the time complexity:

- The outer loop iterates from i=2 to i=n. This loop executes n-1 times.
- The inner loop iterates from j=2 to j=sqrt(i). This loop executes sqrt(i)-1 times for each i. Since i ranges from 2 to n, the total number of iterations of the inner loop is:
- sqrt(2) + sqrt(3) + ... + sqrt(n)
- This sum is less than n times the square root of n, or O(n*sqrt(n)).
- The code inside the inner loop takes constant time to execute.

Therefore, the total time complexity of the algorithm is O(n*sqrt(n)) or O(n^3/2).

Graphs

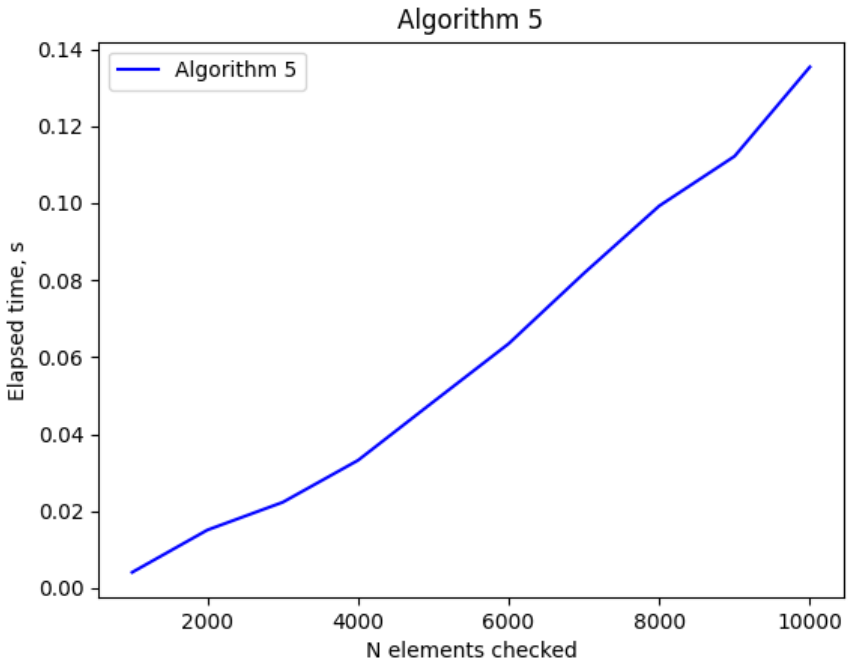

Figure 15. Algortihm nr.5 graph

All Algorithms

All results

Sieve of E. Algs	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Alg 1	0.0002	0.0003	0.0005	0.0007	0.0009	0.001	0.0012	0.0014	0.0016	0.0017
Alg 2	0.0003	0.0007	0.0012	0.0016	0.0021	0.0027	0.0031	0.0037	0.0043	0.0054
Alg 3	0.0068	0.0224	0.052	0.1126	0.1343	0.1744	0.2237	0.2978	0.3672	0.422
Alg 4	0.0312	0.1878	0.3161	0.5774	0.8712	1.344	1.6489	2.0849	2.6359	3.2608
Alg 5	0.0041	0.0151	0.0223	0.0332	0.0484	0.0635	0.0818	0.0993	0.1122	0.1354

Figure 16. All algortihms' results

Algorithm one and two seem to be the fastest by a margin compared to the others and also the fourth one is the slowest out of all of them.

All graphs

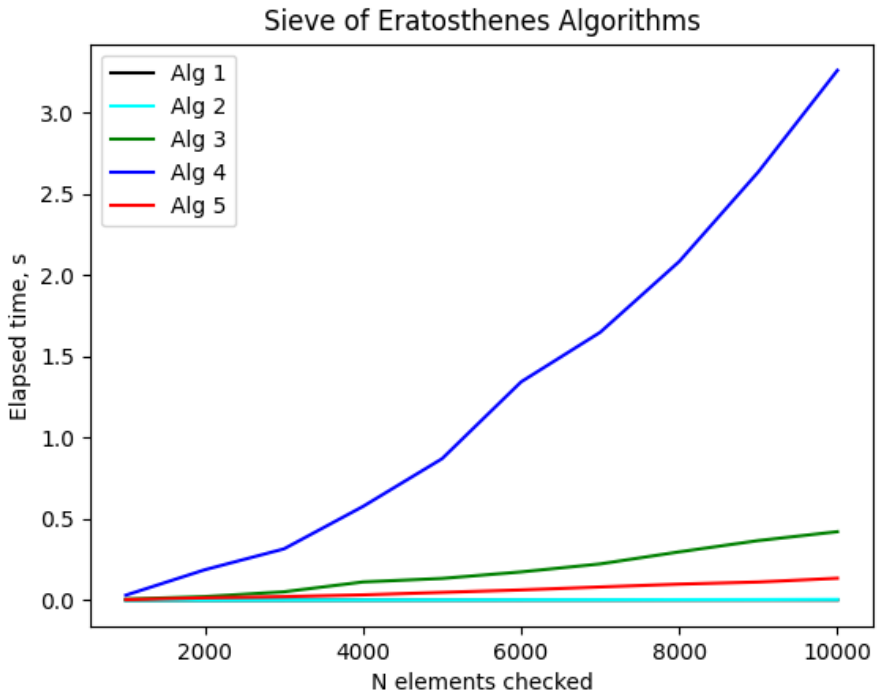

Figure 17. All algortihms' graphs

CONCLUSION

In conclusion, the Sieve of Eratosthenes is a simple and efficient algorithm or family of algorithms for finding all prime numbers up to a given limit. It works by iteratively by marking the multiples of each prime, starting with 2, and moving on to the next unmarked number until all numbers up to the limit have been processed. By testing different algorithms for the Sieve of Eratosthenes several conclusions can be made. The performance of the algorithms is significantly affected by the input format, with some being more effective for smaller inputs and others for larger inputs. The programming language and data structures used can also affect the efficiency of the algorithm. The algorithm's complexity can also impact performance, with more complex algorithms being slower but potentially more accurate. However, the fundamental idea behind the Sieve of Eratosthenes remains the same, and it is possible to efficiently generate a list of prime numbers by iteratively eliminating non-prime numbers. When selecting an algorithm, it is important to consider factors beyond performance, such as ease of implementation and maintainability.

All of the algorithms are being pretty straightforward with regards to their implementations, such that it all comes down to their perfomance when deciding which one to choose:

The first being the fastest, closely followed by the second, the fourth being the slowest, and finally having the fifth and third in the middle, the fifth being a little faster.

Overall, implementing the Sieve of Eratosthenes using different algorithms can provide valuable insights into generating prime numbers and their relation to a more broader pattern of numbers.

BIBLIOGRAPHY

Github: https://github.com/Grena30/APA_Labs/tree/main/Lab3