Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 3

Discipline: Algorithms’ Analysis

Topic: Empirical analysis of algorithms for obtaining Eratosthenes Sieve

Elaborated: Tudor Sclifos,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Transform the pseudocode to compliable code based on the **sieve of Eratosthenes.**

**Tasks:**

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

**Theoretical Notes:**

The sieve of Eratosthenes is an algorithm used to find all prime numbers up to a given limit. It was named after the ancient Greek mathematician Eratosthenes, who used it to find all primes up to 100.

The algorithm works by creating a list of all numbers from 2 to the given limit, and then iteratively marking off all multiples of each prime number starting from 2. At the end of the process, all unmarked numbers remaining in the list are prime.

**Introduction:**

In this laboratory work I am going to implement 5 algorithms to get the sieve of Eratosthenes.

The code is written in pseudocode and I must convert it to a general purpose language. The language I am going to use is Python for its simplicity.

**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, each algorithm will receive (some of the algorithms will get a slice of this) :

[500, 1500, 5000, 15000, 50000, 150000, 500000, 1000000]

**IMPLEMENTATION**

**Algorithm 1**

c[1] = false;

i=2;

while (i<=n){

if (c[i] == true){

j=2\*i;

while (j<=n){

c[j] =false;

j=j+i;

}

}

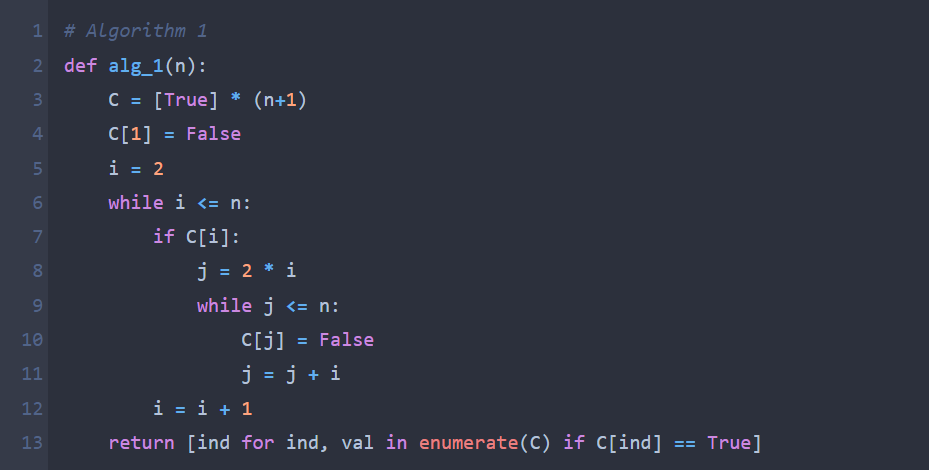
i=i+1;

}

**Implementation:**

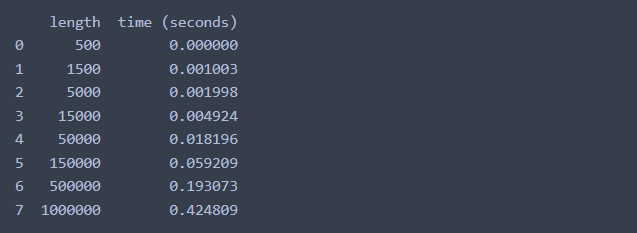
The loop iterates over all integers from 2 to n. If C[i] is True, the inner loop starts at j=2\*i and marks all j values in the list c as False, since they are multiples of i and not prime.

Finally, the code loops over all integers from 2 to n and prints the ones that are still marked as True, which correspond to the prime numbers.



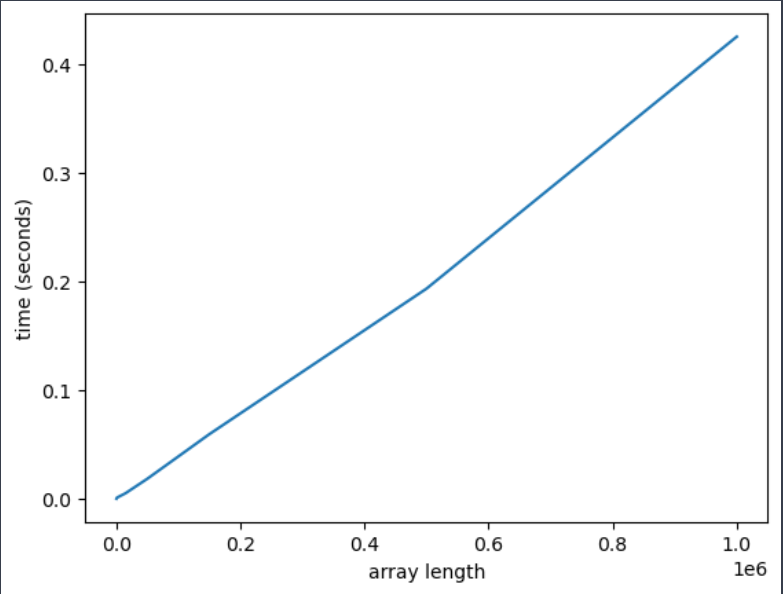
**Figure 1. Implementation Algorithm 1**

**Results:**



**Figure 2. Result Algorithm 1**

**The plot:**



**Figure 3. Plot of Algorithm 1**

**Algorithm 2**

C[1] =false;

i=2;

while (i<=n){

j=2\*i;

while (j<=n){

c[j] =false;

j=j+i;

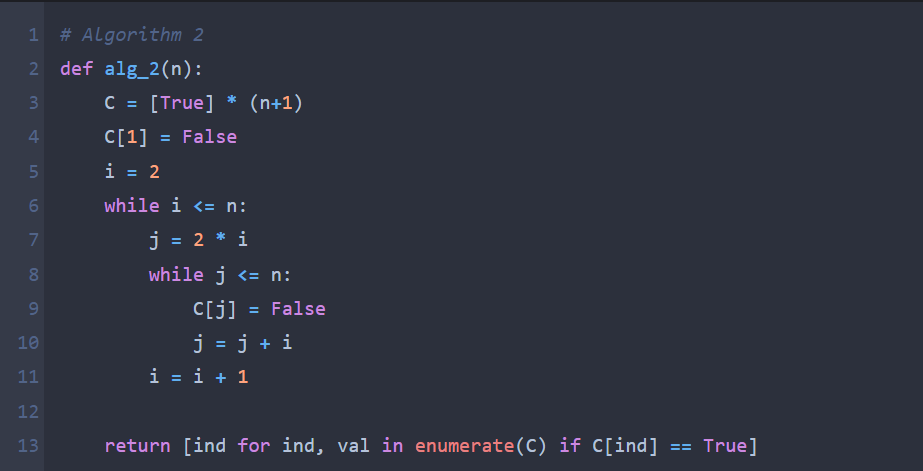
}

i=i+1;

**Implementation:**

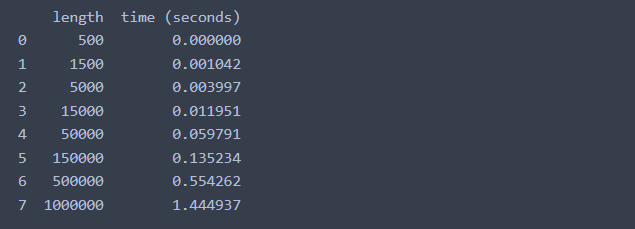
The loop iterates over all integers from 2 to n. The inner loop starts at j=2\*i and marks all j values in the list c as False, since they are multiples of i and not prime.

Finally, the code loops over all integers from 2 to n and prints the ones that are still marked as True, which correspond to the prime numbers.



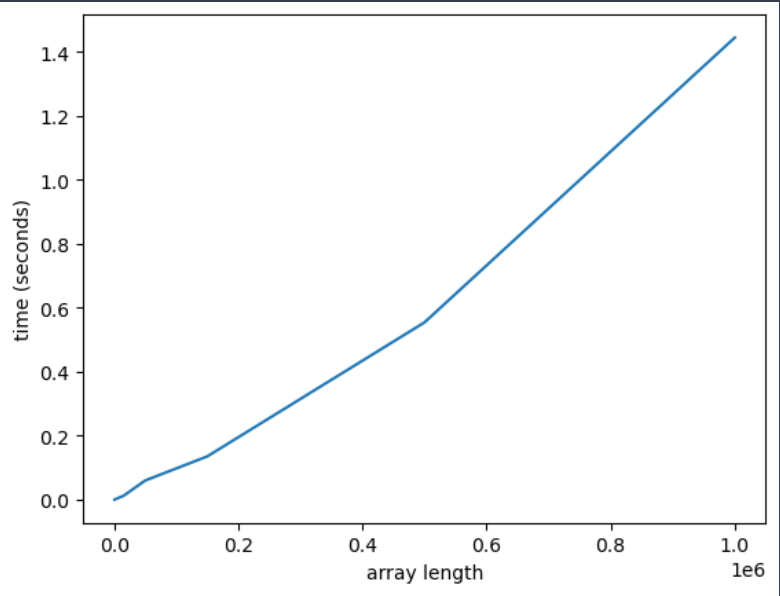
**Figure 4. Implementation Algorithm 2**

**Results:**



**Figure 5. Result Algorithm 2**

**The plot:**



**Figure 6. Plot of Algorithm 2**

**Algorithm 3**

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=j+1;

}

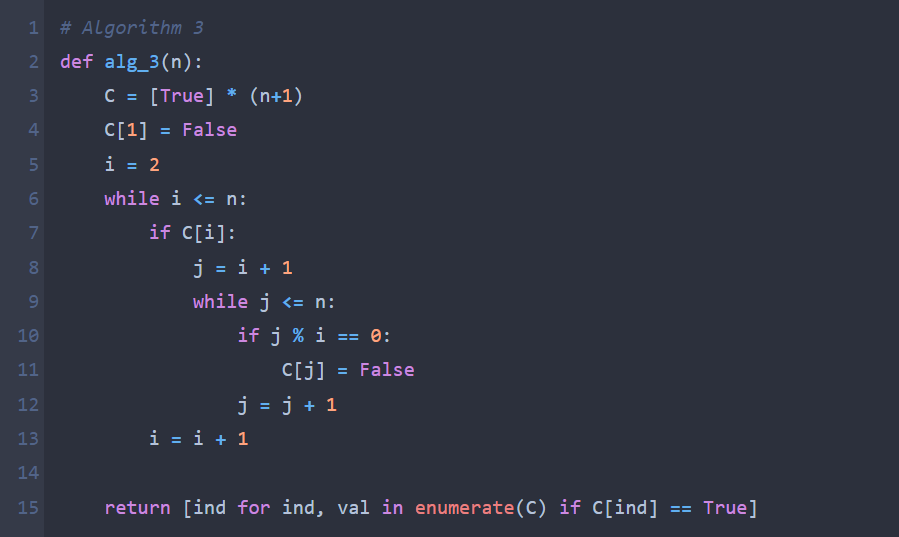
}

i=i+1;

}

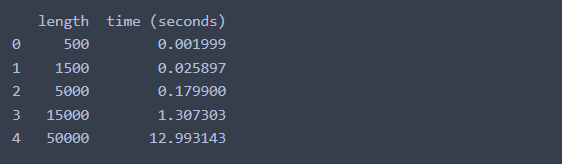
The implementation is similar to the previous functions, but instead of marking all multiples of i as composite, the function only marks the multiples that are greater than i (starting from j=i+1). This is because all the multiples of i that are less than i have already been marked as composite by previous iterations.

**Implementation:**



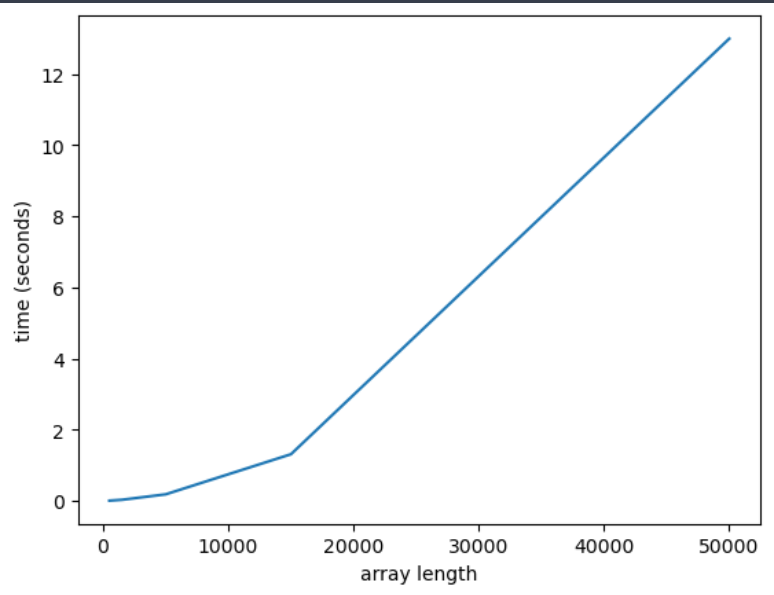
**Figure 7. Implementation Algorithm 3**

**Results:**



**Figure 8. Result Algorithm 3**

**The plot:**



**Figure 9. Plot Algorithm 3**

**Algorithm 4**

C[1] = false;

i = 2;

While (i<=n){

j=1;

while (j<i){

if ( i % j == 0)

{

c[i] = false

}

j=j+1;

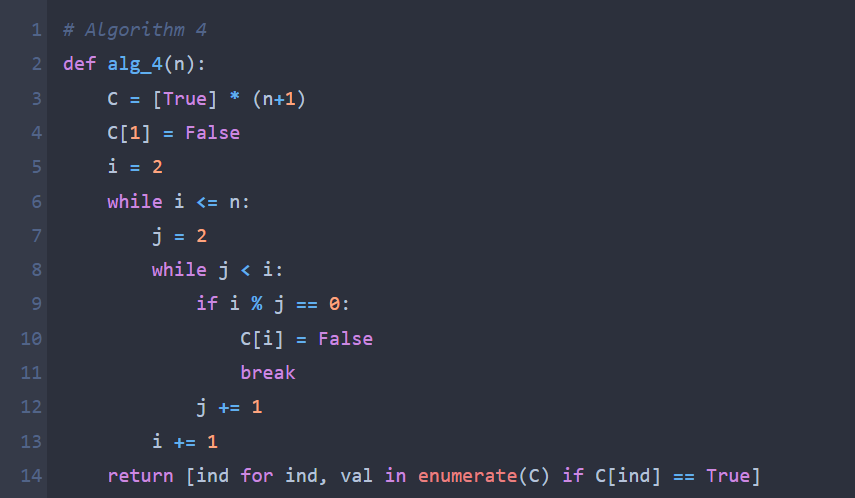
}

i=i+1;

}

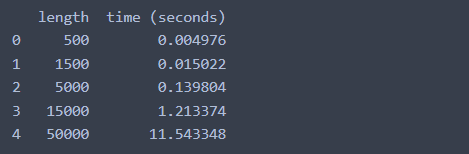
The loop iterates over all integers from 2 to n. The inner loop starts at j=1 and checks all numbers less than i to see if they divide i evenly. If any such number is found, c[i] is marked as False and the loop is exited using break.

**Implementation:**



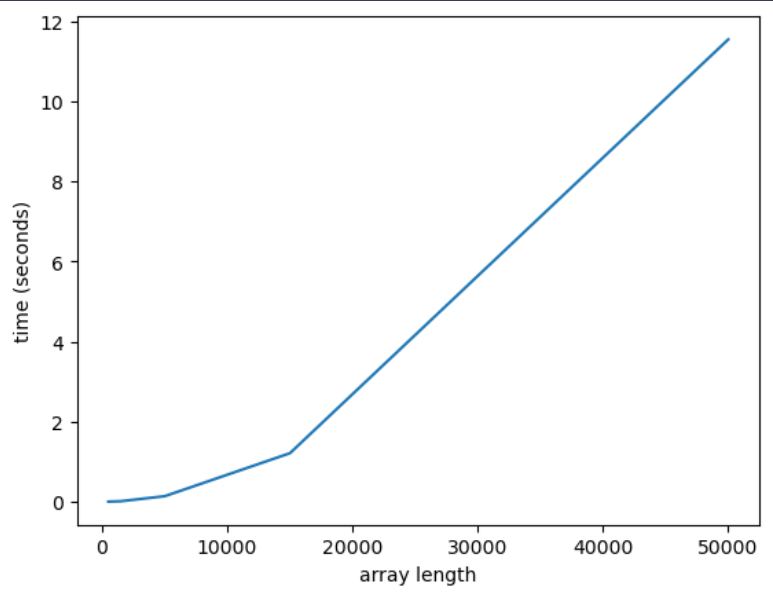
**Figure 10. Implementation Algorithm 4**

**Results:**



**Figure 11. Result Algorithm 4**

**The plot:**



**Figure 12. Plot Algorithm 4**

**Algorithm 5**

C[1] = false;

i=2;

while (i<=n){

j=2;

while (j<=sqrt(i)){

if (i % j == 0) {

c[i] = false;

}

j++;

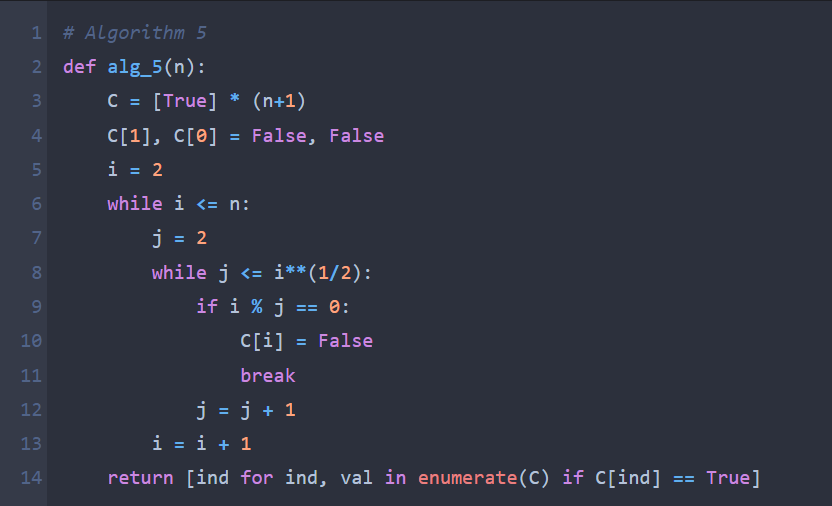
}

i++;

}

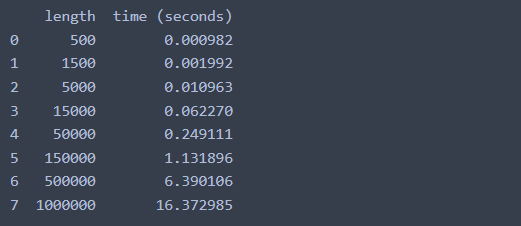
The loop iterates over all integers from 2 to n. The inner loop starts at j=2 and checks all numbers less than or equal to the square root of i to see if they divide i evenly. If any such number is found, c[i] is marked as False and the loop is exited using break.

**Implementation:**



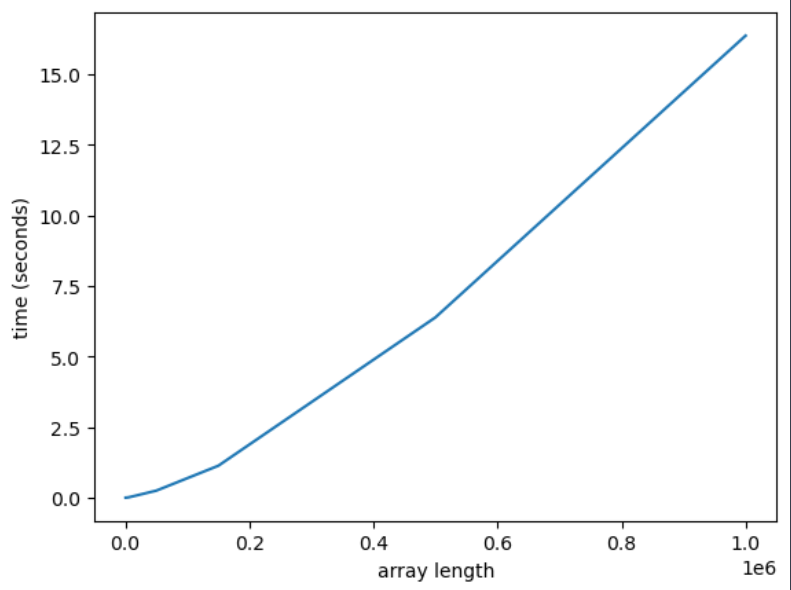
**Figure 13. Implementation Algorithm 5**

**Results:**



**Figure 14. Result Algorithm 5**

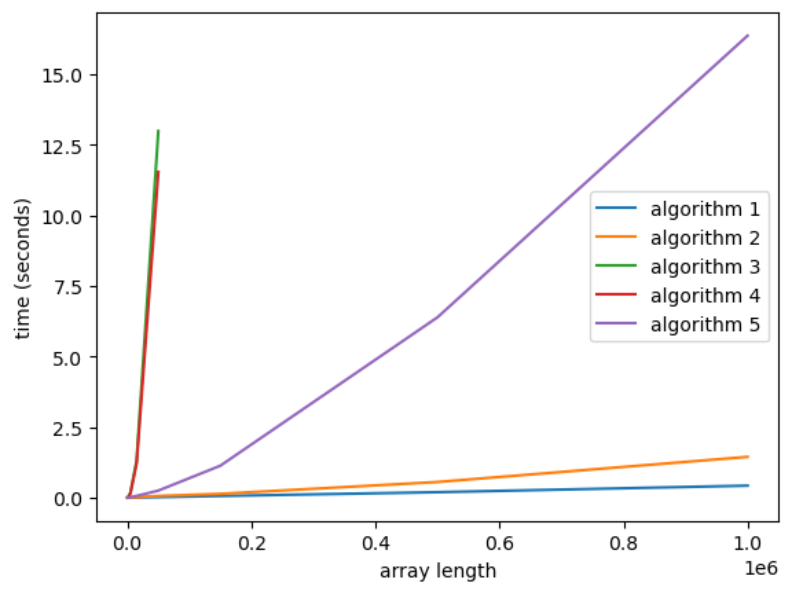
**The plot:**



**Figure 15. Plot Algorithm 5**

**Conclusion:**

The algorithms 3 and 4 performed very bad obtaining a time of 13 and 11 seconds for only 50000 numbers. The 5th algorithm performed much better than the other 2, getting a time of 15 seconds for 1 million numbers. The best results I got are from algorithm 1 and 2, giving the result for 1 million numbers in 0.4 and 1.4 s respectively.



**Figure 16. All algorithms**

**Link to GitHub:** <https://github.com/SexomQ/AlgorithmsAnalysis-labs>