**Ministry of Education and Research of the Republic of Moldova**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**REPORT**

Laboratory work no. 3

*to Eratosthenes Sieve Algorithms*

Did:

St. gr. FAF-211 Andrei Ceban

Checked:

asist. univ. Cristofor Fistic

Chişinău - 2023

**Laboratory work no. 3**

**Objective:** Empirical analysis of algorithms for obtaining Eratosthenes Sieve.

**INTRODUCTION**

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to a given limit. It was developed by the Greek mathematician Eratosthenes around 200 BCE. The algorithm works by repeatedly marking the multiples of each prime number, starting with 2, and then iterating through the remaining unmarked numbers to find the next prime. The process continues until all numbers up to the limit have been marked or identified as prime. The Sieve of Eratosthenes is a simple yet efficient method for finding prime numbers, and it has been used for centuries in various mathematical applications.

The algorithm proceeds as follows:

1. Create a list of consecutive integers from 2 through the given limit.
2. Start with the first prime number, 2, and mark all its multiples greater than 2 in the list. These multiples will not be prime, so they can be crossed out.
3. Move to the next unmarked number in the list, which is 3. This number is a prime, so mark all its multiples greater than 3 in the list.
4. Continue this process, marking multiples of each new prime number found, until all numbers up to the limit have been processed.
5. The unmarked numbers in the list are all prime numbers up to the given limit.

By the end of this process, all composite numbers (numbers that are not prime) will have been crossed out, leaving only the prime numbers in the list. This algorithm is simple and efficient, and can be used to find all prime numbers up to very large limits.

**IMPLEMENTATION**

**Algorithm 1**:

The code starts by initializing an array c of boolean values, where c[i] represents whether the number i is prime or not. The first element, c[1], is initialized as false since 1 is not considered prime.

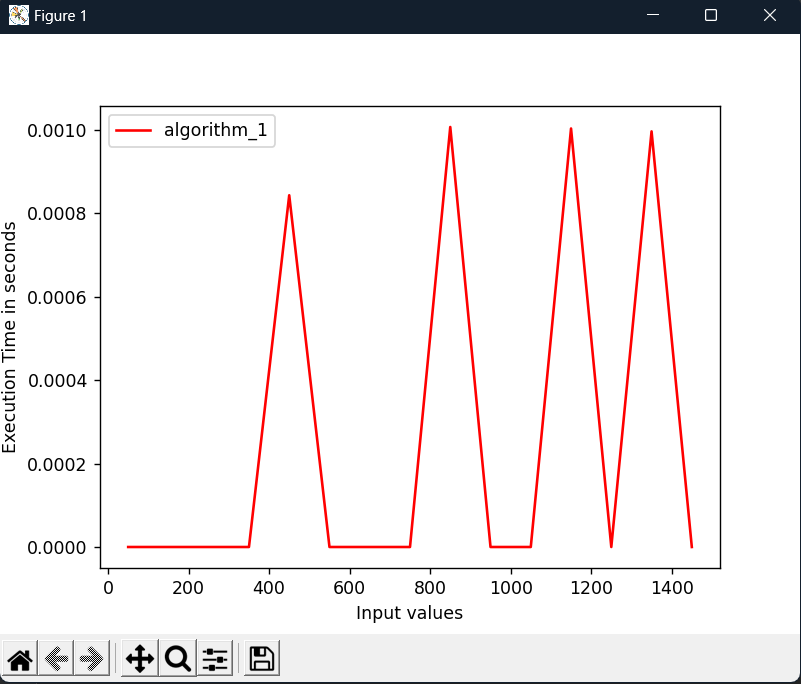
The algorithm then starts with the first prime number, 2, and marks all of its multiples in the array as false, indicating that they are not prime. It then moves to the next unmarked number in the array, which is the next prime number, and repeats the process until all numbers up to n have been processed.

The outer loop iterates through each number in the array, and the inner loop iterates through each multiple of the current prime number found. The inner loop starts with the first multiple of i, which is 2\*i, and marks all of its subsequent multiples as false. This process continues until all multiples of the current prime number have been marked, and the outer loop then moves on to the next unmarked number in the array.

Time Complexity : O(n\*log(log(n)))

**Code :**

# Algorithm 1 -------------------------------------------------------  
def algorithm\_1(n):  
 primes1 = []  
 list1 = [True] \* (n + 1) # puneam toate elementele din lista markate(True)  
 list1[0] = list1[1] = False # primele 2 elemente le punem (False) astfel vom incepe de la indexul 2  
  
 i = 2  
 while (i <= n):  
 if (list1[i] == True):  
 j = 2 \* i  
 while (j <= n):  
 list1[j] = False  
 j = j + i  
 i = i + 1  
  
 for i in range(n + 1):  
 if list1[i]:  
 primes1.append(i)  
 return primes1



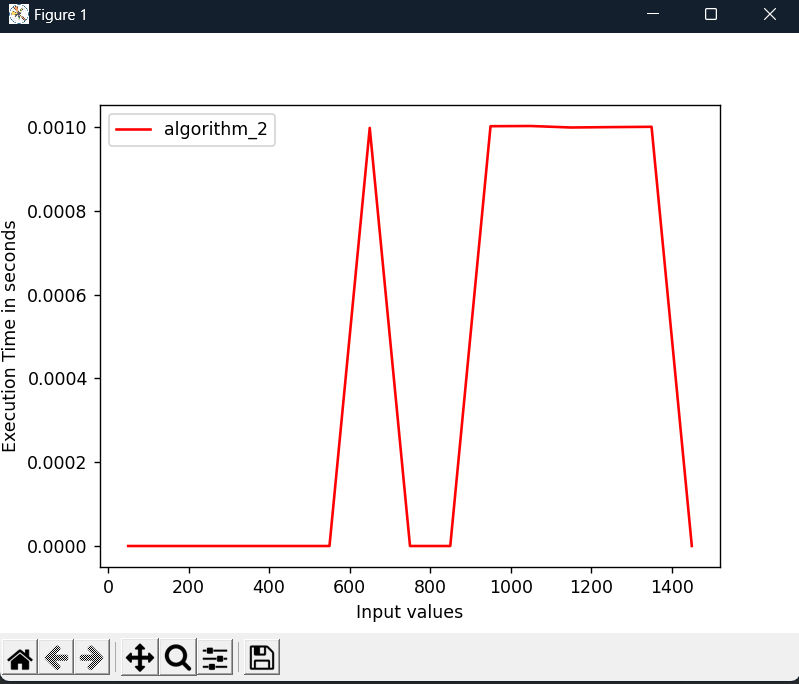
**Algorithm 2**:

This algorithm is similar to Algorithm 1 but it does not have the if statement inside the loop. Instead, it sets all multiples of i to false in the array c, regardless of whether i is a prime number or not. It also starts the loop at j=2\*i instead of j=i because all multiples of i less than i have already been set to false in previous iterations of the outer loop.

Time Complexity : O(n\*log(log(n)))

**Code :**

# Algorithm 2 -------------------------------------------------------  
def algorithm\_2(n):  
 primes2 = []  
 list2 = [True] \* (n + 1) # puneam toate elementele din lista markate(True)  
 list2[0] = list2[1] = False # primele 2 elemente le punem (False) astfel vom incepe de la indexul 2  
  
 i = 2  
 while (i <= n):  
 j = 2 \* i  
 while (j <= n):  
 list2[j] = False  
 j = j + i  
 i = i + 1  
  
 for i in range(n + 1):  
 if list2[i]:  
 primes2.append(i)  
 return primes2



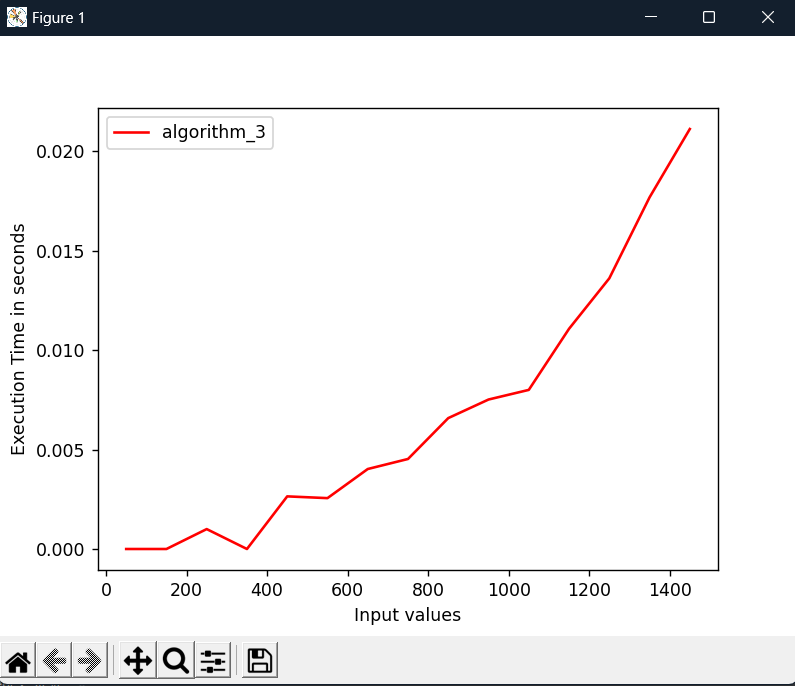
**Algorithm 3**:

This algorithm also finds all prime numbers between 2 and n using a method called the "Sieve of Sundaram." It initializes the array c and sets c[1] to false. It then loops through all numbers between 2 and n and checks if the element c[i] is true. If it is true, it sets all multiples of i+1 (i.e., odd numbers greater than i) in the array c to false. This is because all even numbers greater than 2 have already been set to false by the initialization of c[1] to false and the fact that even numbers are not prime. The algorithm ends once it has looped through all numbers between 2 and n.

Time Complexity : O(n^2)

**Code :**

#Algorithm 3 -------------------------------------------------------  
def algorithm\_3(n):  
 primes3 = []  
 list3 = [True] \* (n + 1) # puneam toate elementele din lista markate(True)  
 list3[0] = list3[1] = False # primele 2 elemente le punem (False) astfel vom incepe de la indexul 2  
  
 i = 2  
 while (i <= n):  
 if (list3[i] == True):  
 j = i + 1  
 while (j <= n):  
 if (j % i == 0):  
 list3[j] = False  
 j = j + 1  
 i = i + 1  
  
 for i in range(n + 1):  
 if list3[i]:  
 primes3.append(i)  
 return primes3



**Algorithm 4**:

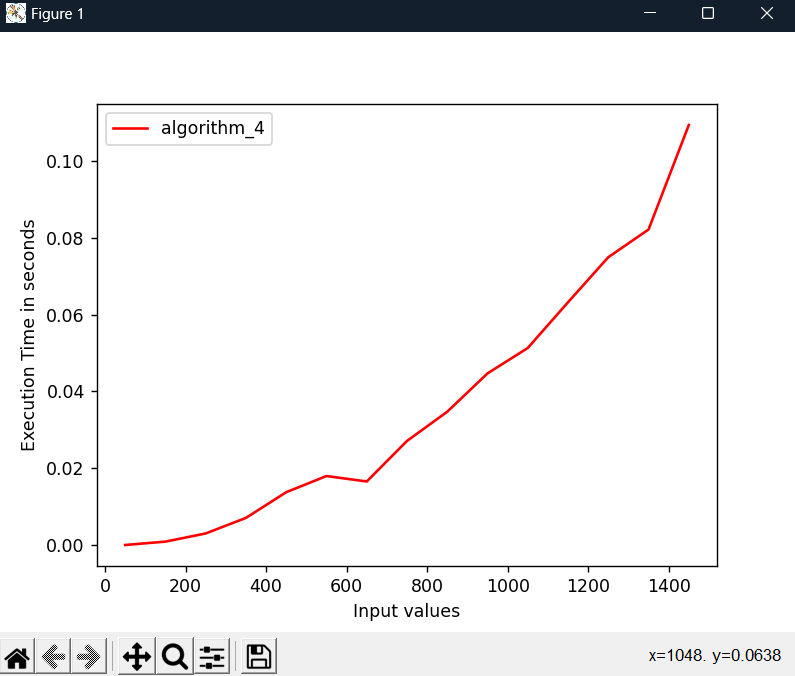
This algorithm finds all prime numbers between 2 and n by checking each number i between 2 and n and testing whether it is divisible by any number j between 1 and i-1. It initializes the array c and sets c[1] to false. It then loops through all numbers between 2 and n and checks whether i is divisible by any j between 1 and i-1. If it is, then c[i] is set to false. The algorithm ends once it has looped through all numbers between 2 and n.

Time Complexity : O(n^2)

I changed the code a little bit, here was j = 1, and i changet to j = 2. Beacuse if it will start from 1 then i divided j will always be 0 when j = 1 for all i .

**Code :**

#Algorithm 4 -------------------------------------------------------  
def algorithm\_4(n):  
 primes4 = []  
 list4 = [True] \* (n + 1) # puneam toate elementele din lista markate(True)  
 list4[0] = list4[1] = False # primele 2 elemente le punem (False) astfel vom incepe de la indexul 2  
  
 i = 2  
 while (i <= n):  
 j = 2  
 while (j < i):  
 if (i % j == 0):  
 list4[i] = False  
 j = j + 1  
 i = i + 1  
  
 for i in range(n + 1):  
 if list4[i]:  
 primes4.append(i)  
 return primes4



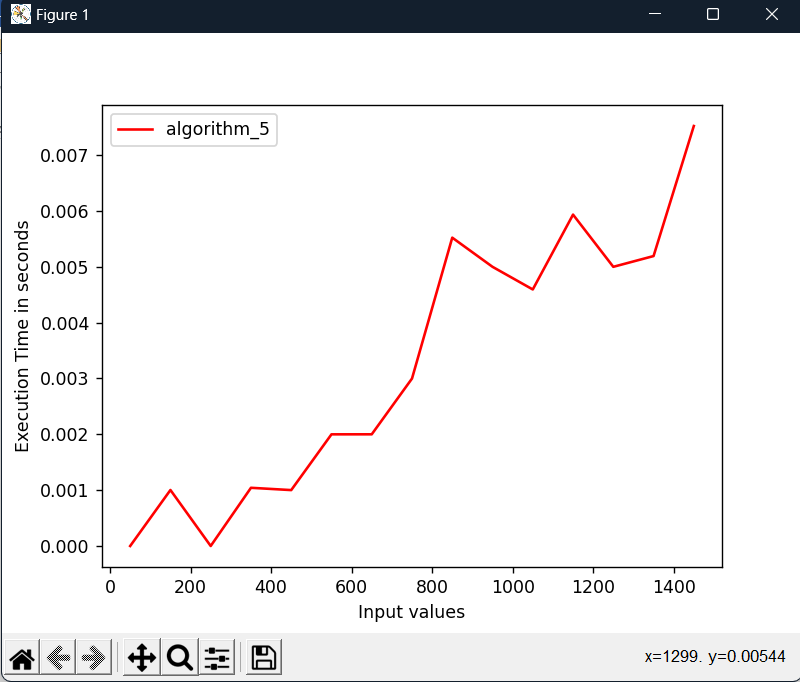
**Algorithm 5**:

This algorithm finds all prime numbers between 2 and n using trial division. It initializes the array c and sets c[1] to false. It then loops through all numbers between 2 and n and checks whether i is divisible by any j between 2 and the square root of i. If it is, then c[i] is set to false. The algorithm ends once it has looped through all numbers between 2 and n.

Time Complexity : O(n sqrt(n))

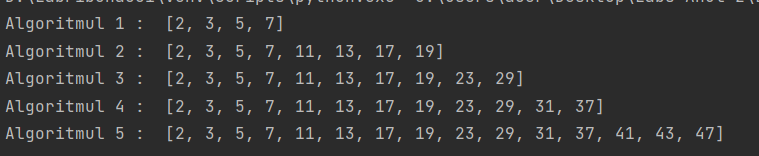
**Code :**

#Algorithm 5 -------------------------------------------------------  
def algorithm\_5(n):  
 primes5 = []  
 list5 = [True] \* (n + 1) # puneam toate elementele din lista markate(True)  
 list5[0] = list5[1] = False # primele 2 elemente le punem (False) astfel vom incepe de la indexul 2  
  
 i = 2  
 while i <= n:  
 j = 2  
 while j <= math.sqrt(i):  
 if i % j == 0:  
 list5[i] = False  
 j = j + 1  
 i = i + 1  
  
 for i in range(n + 1):  
 if list5[i]:  
 primes5.append(i)  
 return primes5



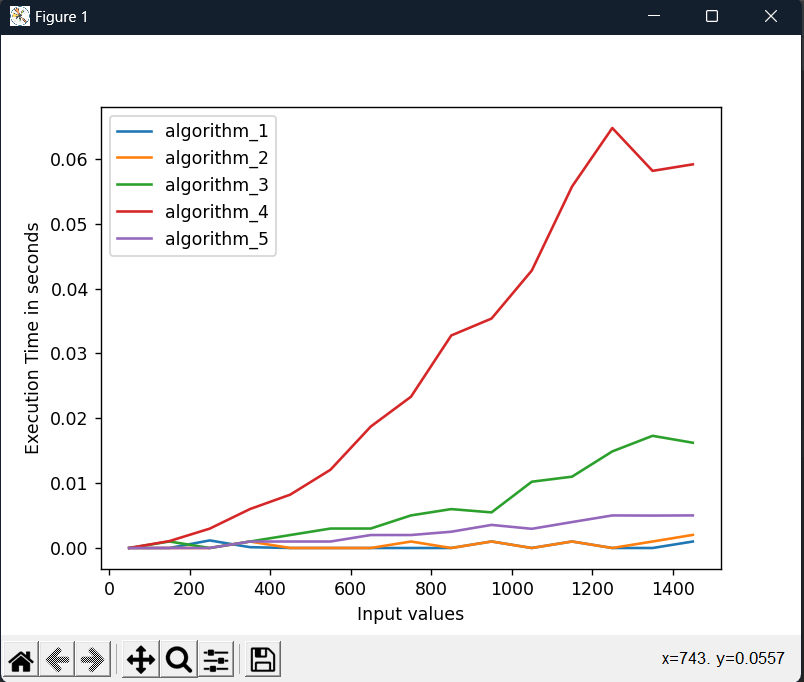
**RESULTS**

Here is represented different results of input for all 5 algorithms.



Here we can see a graph where is represented all 5 algorithms. The graph is based on the time execution of algorithm and the input values.

The lines show us that algorithm 4 is the slowest one. Algorithm 1 and 2 are highly efficient. The algorithm 3 and 5 are between 4 and 1 not the fastest and not the slowest.



**CONCLUSION**

All five algorithms aim to find all prime numbers between 2 and n, but they use different approaches to achieve this goal.

Algorithm 1 and Algorithm 2 use the Sieve of Eratosthenes algorithm, which is a highly efficient algorithm for finding prime numbers. Algorithm 1 has an additional if statement that skips over composite numbers in the inner loop, making it slightly more efficient than Algorithm 2.

Algorithm 3 which is less efficient but still has a relatively low time complexity. It skips over even numbers in the inner loop, which reduces the number of iterations needed to find all primes.

Algorithm 4 and Algorithm 5 use trial division to check each number between 2 and n for primality. Algorithm 4 checks whether a number is divisible by any number less than itself, while Algorithm 5 checks whether it is divisible by any prime number less than or equal to the square root of the number. These algorithms have a higher time complexity than the Sieve algorithms and are less efficient for larger values of n.

In summary, the Sieve of Eratosthenes is the most efficient algorithm for finding prime numbers, while the other algorithms have a higher time complexity and may be less practical for larger values of n.

Git Repo : https://github.com/andeiceban0352/Labs-Anul2/tree/main/Lab%20Analiza%20Algoritmilor/Lab3