**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**REPORT**

Laboratory Work nr.3

*at Algorithms Analysis*

Elaborated by:

st. gr. FAF-211 Hariton Dan

Verified by:

asist. univ. Fiștic Cristofor

Chişinău - 2023

**ALGORITHM ANALYSIS**

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

**Tasks:**

1. Implement the algorithms listed below 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.

**Establish Comaparation:**

We will compare these 5 algorithms using numbers up to:

* 500
* 5000
* 50000

**Comparison Metric:**

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

**IMPLEMENTATION**

**Algorithm 1:**

This is the Sieve of Eratosthenes algorithm. The best case scenario is when all numbers from 1 to n are prime, in which case the algorithm takes O(nloglogn) time. The worst case scenario is when all numbers from 1 to n are composite, in which case the algorithm takes O(nloglogn) time.

def sieve\_of\_eratosthenes(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

## Algorithm 2:

## This is similar to Algorithm 1, but it doesn't check whether a number is prime before marking its multiples as composite. As a result, it is less efficient than Algorithm 1. The best case scenario is the same as Algorithm 1, and the worst case scenario is also O(nloglogn).

def sieve\_of\_eratosthenes(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

## Algorithm 3:

## This algorithm checks whether each number is divisible by any prime number less than itself. The best case scenario is when all numbers from 1 to n are prime, in which case the algorithm takes O(nlogn) time. The worst case scenario is when all numbers from 1 to n are composite, in which case the algorithm takes O(n^2) time.

def sieve\_of\_eratosthenes(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

## Algorithm 4:

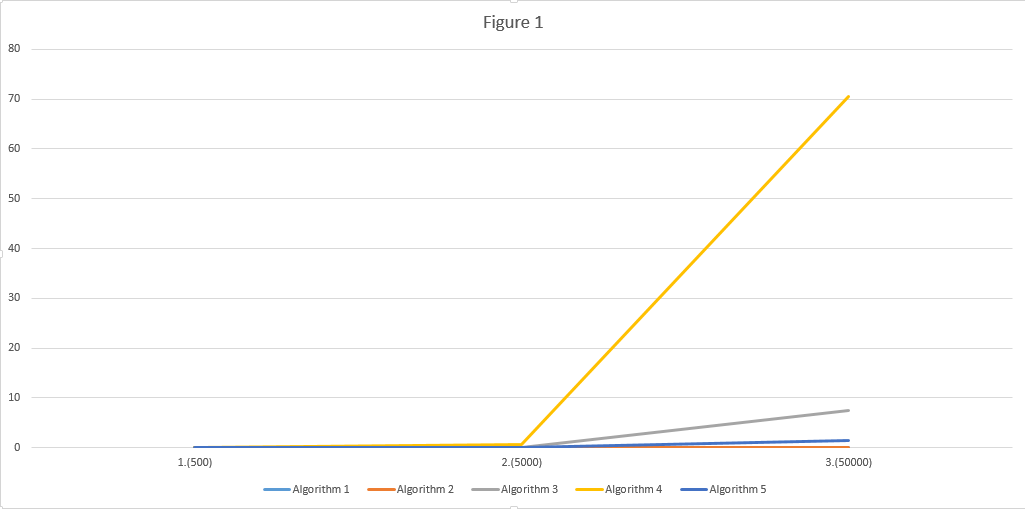
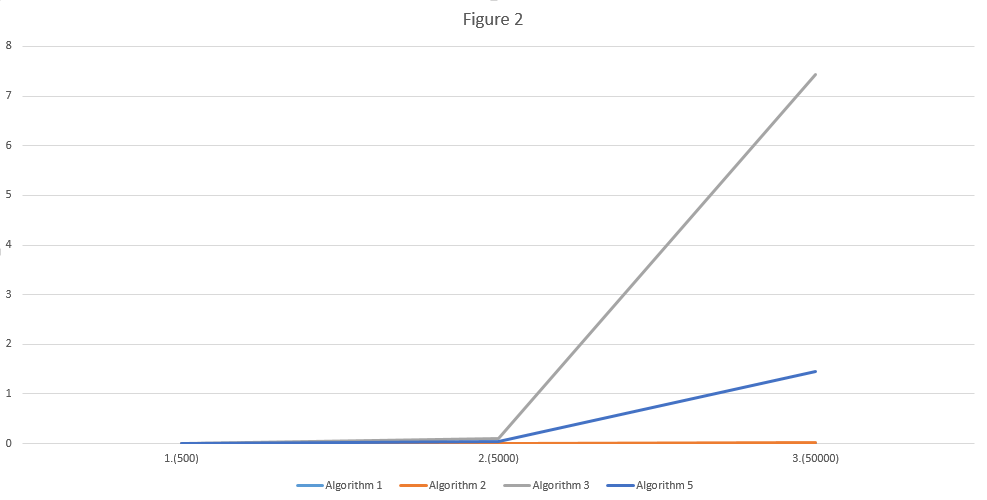
## This algorithm is similar to the trial division algorithm, but it checks all numbers less than the current number to see if they are factors. As a result, it is less efficient than the standard trial division algorithm. The best case scenario is when all numbers from 1 to n are prime, in which case the algorithm takes O(nlogn) time. The worst case scenario is when all numbers from 1 to n are composite, in which case the algorithm takes O(n^2) time.

def sieve\_of\_eratosthenes(n):  
 c = [True] \* (n + 1)  
 c[1] = False  
 i = 2  
 while i <= n:  
 j = 2 # error ?  
 while j < i:  
 if i % j == 0:  
 c[i] = False  
 j = j + 1  
 i = i + 1

## Algorithm 5:

## This algorithm is also a trial division algorithm, but it only checks divisors up to the square root of the current number, since no factor greater than the square root can be multiplied by a factor less than the square root to produce the current number. The best case scenario is when all numbers from 1 to n are prime, in which case the algorithm takes O(nlogn) time. The worst case scenario is when all numbers from 1 to n are composite and have only small prime factors, in which case the algorithm takes O(n^1.5) time.

def sieve\_of\_eratosthenes(n):  
 c = [True] \* (n + 1)  
 c[1] = False  
 i = 2  
 while i <= n:  
 j = 2  
 while j <= int(sqrt(i)):  
 if i % j == 0:  
 c[i] = False  
 j += 1  
 i += 1

**Results:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Algorithm 1 | Algorithm 2 | Algorithm 3 | Algorithm 4 | Algorithm 5 |
| 1.(500) | 0.0 | 0.0 | 0.0009639263153076172 | 0.00551295280456543 | 0.002001523971557617 |
| 2.(5000) | 0.0009996891021728516 | 0.0020003318786621094 | 0.09557247161865234 | 0.657634973526001 | 0.045118093490600586 |
| 3.(50000) | 0.011186361312866211 | 0.0293426513671875 | 7.427042722702026 | 70.60278797149658 | 1.4497005939483643 |

**Conclusion:**

The proposed work's aim, which consists of the time-based and theoretical analysis of Sieve of Eratosthene algorithms, was successfully accomplished. Five different algorithms were used to determine their temporal complexity and to highlight the most efficient algorithm that will display the desired result using the least time that passed by. The comparison of the algorithms was performed using a table where we collected time based data on the iterations received during the work and the graphs constructed in Excel, which show the noticeable difference between the complexities of the algorithms. From this we observe that the fourth is the most inefficient, because it checks all numbers less than the current number to see if they are factors. The rest of them look pretty even, but at a closer inspection we can see that the third algorithm is the second most efficient method. In terms of efficiency, first and second algorithms are generally the most efficient algorithms for finding all primes up to a given number, followed by Algorithm 5 (trial division up to the square root).

**Link to GitHub**: <https://github.com/haritondan/AA-Labs>