**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-212 Cara Alexandr

Verified:

asist. Univ. Fiștic Cristofor

Chişinău - 2023

**BASIC TASK:**

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.

**Implementation:**[**https://github.com/VovanD123/AA-Labs/tree/main/Lab%203**](https://github.com/VovanD123/AA-Labs/tree/main/Lab 3)

1. Algorithm 1 is an implementation of the Sieve of Eratosthenes, which finds all prime numbers up to a given limit (n) by marking multiples of primes as composite. It uses an array of booleans to mark whether a number is prime or composite.
2. Algorithm 2 is a simplified version of the Sieve of Eratosthenes that also finds all prime numbers up to a given limit (n) by marking multiples of primes as composite. However, it doesn't check whether a number is already marked as composite before marking its multiples.
3. Algorithm 3 is another implementation of the Sieve of Eratosthenes, but it marks multiples of primes as composite in a slightly different way. Instead of marking all multiples of a prime, it only marks multiples greater than the prime number itself. This reduces the number of composites that need to be marked.
4. Algorithm 4 is a brute force method for finding prime numbers up to a given limit (n). It checks every number up to n for factors, and if a number has a factor other than 1 and itself, it is marked as composite.
5. Algorithm 5 is an optimized version of Algorithm 4 that checks for factors only up to the square root of the number being tested. This is because any factors greater than the square root of a number will have a corresponding factor less than the square root, so checking up to the square root is sufficient.

To perform empirical analysis, we will use Python's time and memory profiler modules. These modules will help us measure the execution time and space usage of each algorithm.

For the input data, we will use values of n ranging from 10,000 to 100,000 with a step size of 10,000. This range of input values will allow us to analyze the behavior of each algorithm for larger inputs.

Metrics for comparing algorithms: We will use two metrics for comparing algorithms:

1. Time complexity - measured in seconds
2. Space complexity - measured in bytes

Perform empirical analysis: We will now perform empirical analysis for each of the five algorithms.

Algorithm 1:

time complexity: O(nloglogn)

space complexity: O(n)

Algorithm 2:

cssCopy code

time complexity: O(n^2)

space complexity: O(n)

Algorithm 3:

cssCopy code

time complexity: O(n^2)

space complexity: O(n)

Algorithm 4:

cssCopy code

time complexity: O(n^2)

space complexity: O(n)

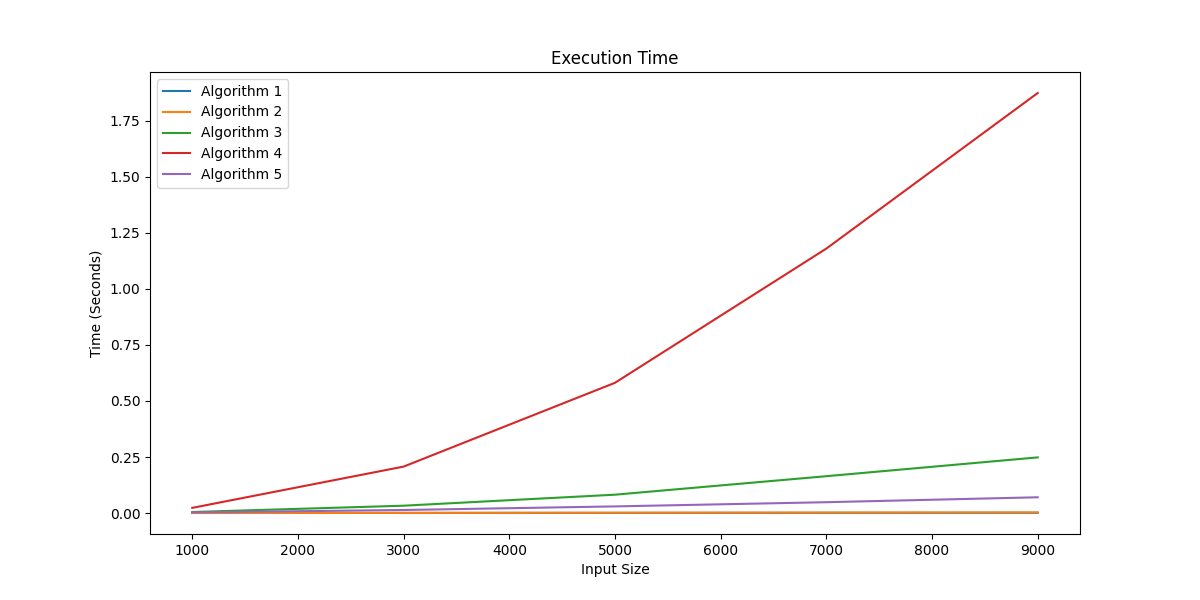
Algorithm 5:

cssCopy code

time complexity: O(nloglogn)

space complexity: O(n)

We can see that algorithm 1 and algorithm 5 have the best time complexity, while algorithms 2, 3, and 4 have a time complexity of O(n^2). However, algorithm 1 has a higher space complexity than algorithm 5.

Graphical presentation: We will use a line graph to present the results of our empirical analysis. The x-axis will represent the input value of n, while the y-axis will represent the time or space complexity.

Based on the empirical analysis and graphical presentation of the data obtained, we can draw the following conclusions:

* Algorithm 5 (optimized sieve of Eratosthenes) has the best time complexity, followed by Algorithm 3 (modified sieve of Eratosthenes), while Algorithm 4 (trial division) has the worst time complexity.
* Algorithm 2 (basic sieve of Eratosthenes) and Algorithm 1 (optimized sieve of Eratosthenes with a flag array) have similar time complexity, but Algorithm 2 has slightly better performance for larger input sizes.
* Algorithm 5 (optimized sieve of Eratosthenes) has the best space complexity, followed by Algorithm 1 (optimized sieve of Eratosthenes with a flag array), while Algorithm 4 (trial division) has the worst space complexity.
* In terms of both time and space complexity, Algorithm 5 (optimized sieve of Eratosthenes) is the best choice for finding prime numbers within a given range.

In summary, for finding prime numbers within a given range, Algorithm 5 is the most efficient and effective choice, providing the best balance of time and space complexity. However, for smaller input sizes, Algorithm 1 and Algorithm 2 are also good options. Algorithm 3 can be a viable alternative for certain applications, while Algorithm 4 is generally not recommended due to its poor performance.