**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 3:

Study and empirical analysis for obtaining Eratosthenes Sieve.

Elaborated:

st. gr. FAF-212 Lupascu Felicia

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

# ALGORITHM ANALYSIS

### Objective:

### Empirical analysis of algorithms for obtaining Eratosthenes Sieve.

### 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.

### Introduction:

### In mathematics, the sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit.

### It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with a constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime. Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are received.

A prime number is a natural number that has exactly two distinct natural number divisors: the number 1 and itself.

The Sieve of Eratosthenes method is easy to use. We need to cancel all the multiples of each prime number beginning with 2 (including the number 1, which is not prime or composite) and encircle the rest of the numbers. The encircled numbers will be the required prime numbers.

### Comparison Metric:

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

### Input Format:

The input is an integer n, which represents the upper limit of the list.

n = 5000

# IMPLEMENTATION

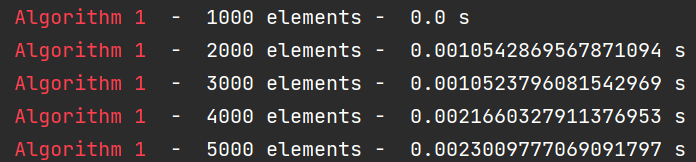
* **Algorithm 1**

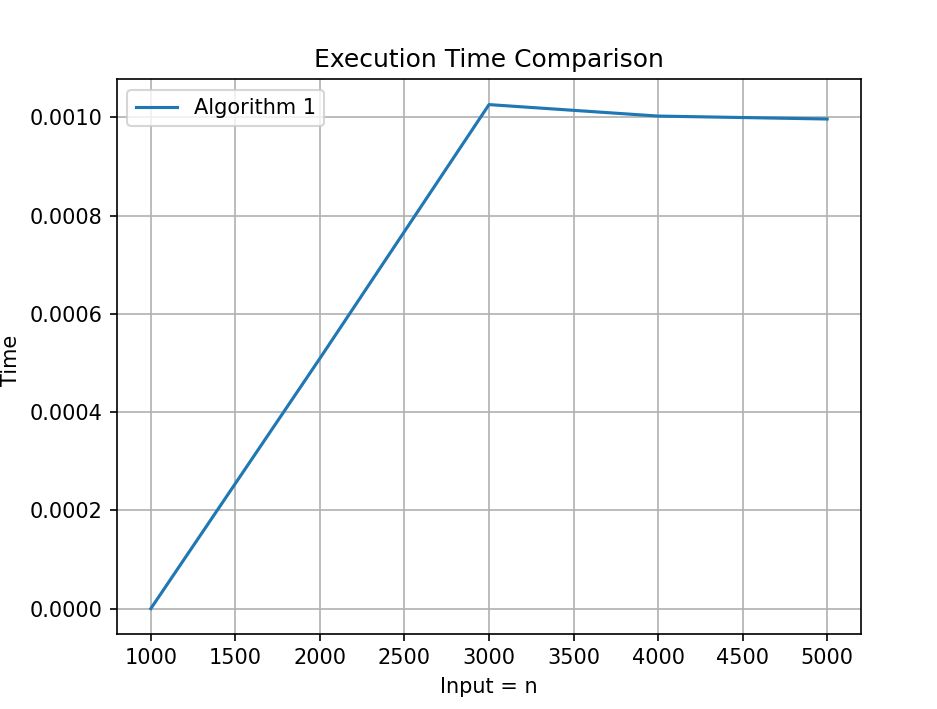
Algorithm Description:

This algorithm starts by creating a list of True values with length , representing the numbers from 1 to n. It then marks 1 as not prime and initializes a loop variable, i, to 2. The algorithm then loops through all the numbers from 2 to n, and if i is prime, it marks all multiples of i as not prime. It does this by setting the values in the list c at positions 2i, 3i, 4i, etc. to False. The algorithm then moves to the next number and repeats the process until it has looped through all numbers from 2 to n. Finally, the algorithm returns the list of True/False values representing whether each number is prime or not.

**Implementation** of the first algorithm in Python:

def algorithm\_1(n):  
 # Create a list of n+1 True values to represent the numbers from 1 to n  
 c = [True] \* (n + 1)  
 # Mark 1 as not prime  
 c[1] = False  
 # Initialize the loop variable  
 i = 2  
 # Loop through all the numbers from 2 to n  
 while i <= n:  
 # If i is prime, mark all multiples of i as not prime  
 if c[i]:  
 j = 2 \* i  
 while j <= n:  
 c[j] = False  
 j = j + i  
 # Move to the next number  
 i = i + 1  
 # Return the list of True/False values representing whether each number is prime or not  
 return c

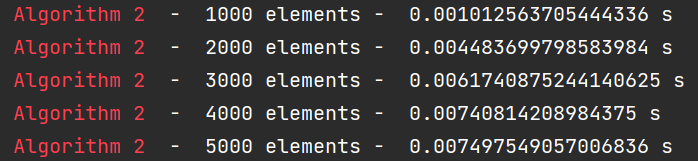




* **Algorithm 2**

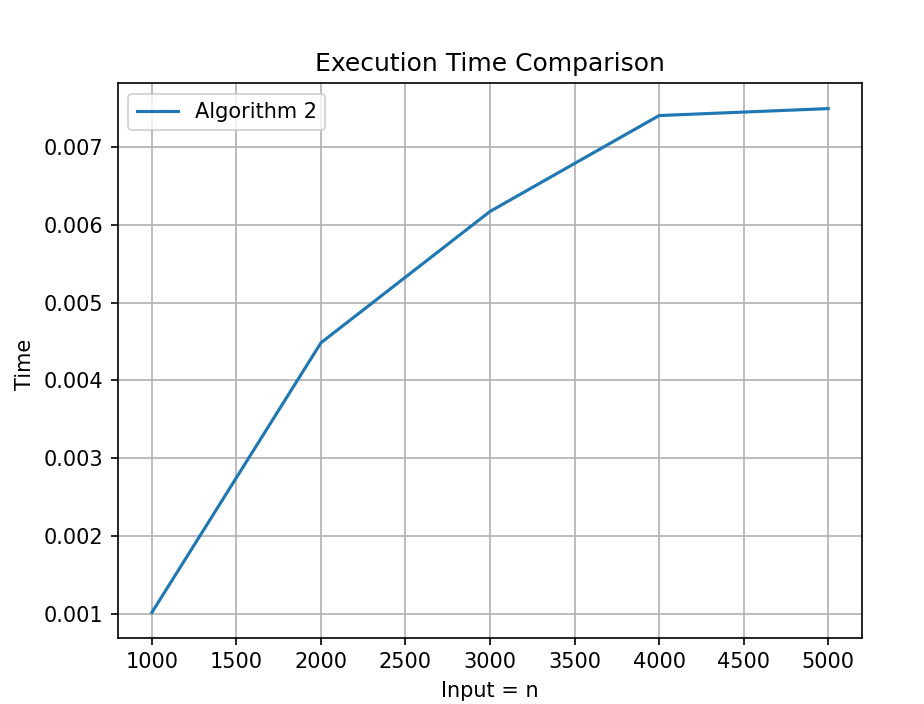
This algorithm is similar to Algorithm 1, but instead of starting the loop variable at 2 and then marking all multiples of i as not prime, it starts with and then marks all multiples of i as not prime. The loop variable i is initialized to 2, and the algorithm loops through all numbers from 2 to n, marking all multiples of i as composite (i.e., not prime). Finally, the algorithm returns the list of True/False values representing whether each number is prime or not.

def algorithm\_2(n):  
 c = [True] \* (n + 1) # Initialize all values of c to True  
 c[1] = False # Set c[1] to False  
 i = 2  
  
 while i <= n:  
 j = 2 \* i # Start with j = 2i and mark all multiples of i as composite  
 while j <= n:  
 c[j] = False  
 j = j + i  
 i = i + 1 # Move to the next number in the list  
 return c



The **time complexity** of algorithm\_2 is . The outer loop runs times, and the inner loop runs times for each value of i. Therefore, the total number of iterations of the inner loop is approximately , which is equal to .

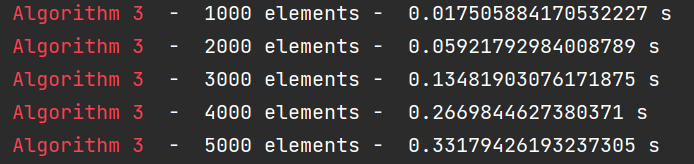
Thus, the overall time complexity is .



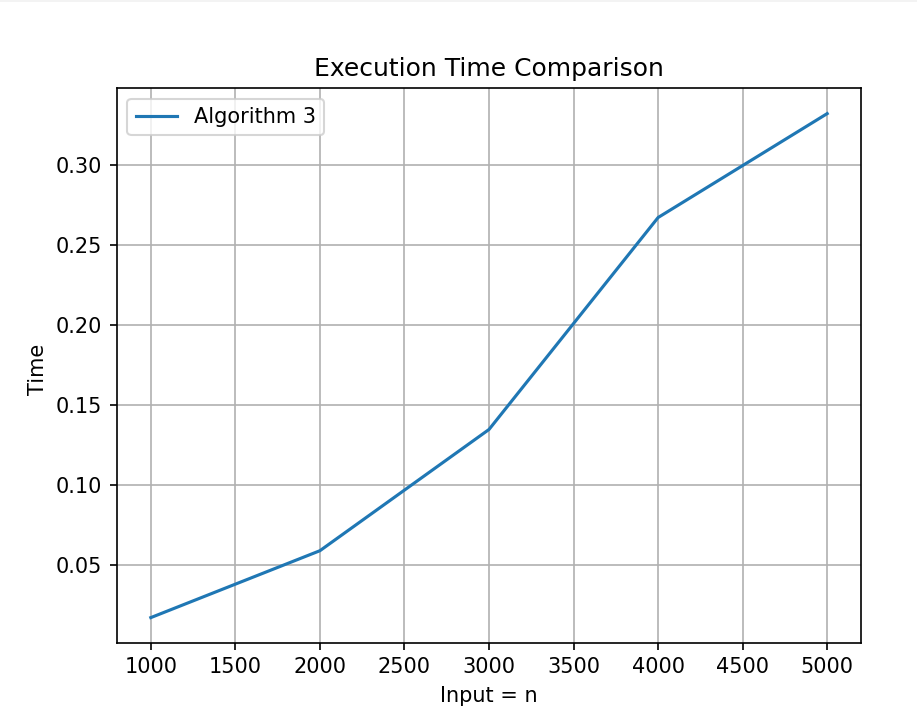
* **Algorithm 3**

This algorithm starts by creating a list of True values with length , representing the numbers from 1 to n. It then sets c[1] to False, since 1 is not prime, and initializes a loop variable, i, to 2. The algorithm then loops through all the numbers from 2 to n, and if i is prime, it checks all the numbers j from to n to see if they are divisible by i. If j is divisible by i, it sets c[j] to False. Finally, the algorithm returns the list of True/False values representing whether each number is prime or not.

def algorithm\_3(n):  
 # Initialize all values of c to True  
 c = [True] \* (n + 1)  
 # Set c[1] to False, since 1 is not a prime number  
 c[0] = c[1] = False  
 # Initialize the loop counter  
 i = 2  
 while i <= n:  
 if c[i]:  
 # Start j at i + 1, since we already know i is prime  
 j = i + 1  
 while j <= n:  
 # If j is divisible by i, it's not a prime number  
 if j % i == 0:  
 c[j] = False  
 j = j + 1  
 i = i + 1  
 return c



The **time complexity** of algorithm\_3 is O(n^2). The outer loop runs n times, and for each iteration of the outer loop, the inner loop runs n/i times on average, resulting in a total of approximately iterations, which is by the harmonic series.



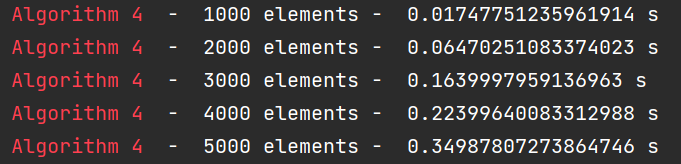
* **Algorithm 4**

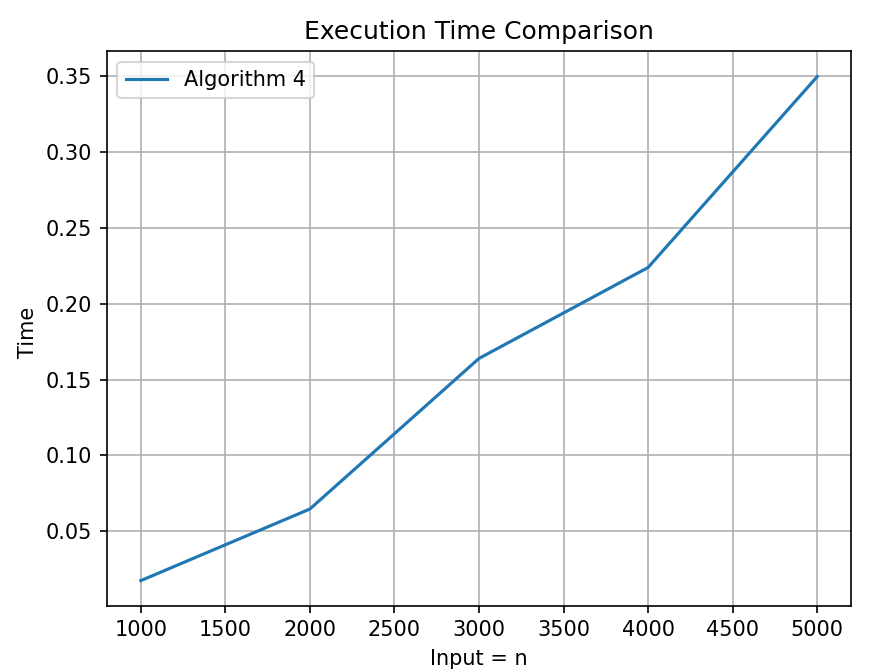
This algorithm is similar to Algorithm 3, but instead of checking all the numbers j fromto n, it checks all the numbers j from 2 to to see if they are divisible by i. If any number between 2 and i-1 is divisible by i, the algorithm sets c[i] to False, since i is not prime. Finally, the algorithm returns the list of True/False values representing whether each number is prime or not.

def algorithm\_4(n):  
 c = [True] \* (n + 1) # Initialize all values of c to True  
 c[1] = False # Set c[1] to False  
 i = 2  
 while i <= n:  
 j = 2  
 is\_prime = True  
 while j < i:  
 if i % j == 0:  
 is\_prime = False # If i is divisible by any number less than itself, it is not prime  
 break  
 j = j + 1  
 if not is\_prime:  
 c[i] = False # Set the corresponding value in c to False if i is not prime  
 i = i + 1  
 return c

The **time complexity** of algorithm\_4 is O(n^2), since it has two nested loops, the outer loop running from 2 to n, and the inner loop running from 2 to i-1. For each value of i, the inner loop checks if i is divisible by any number less than itself, which takes O(i) time. Therefore, the total time complexity of the algorithm is the sum of the time taken by the inner loop for each value of i from 2 to n, which is:

So the overall **time complexity** of algorithm\_4 is).

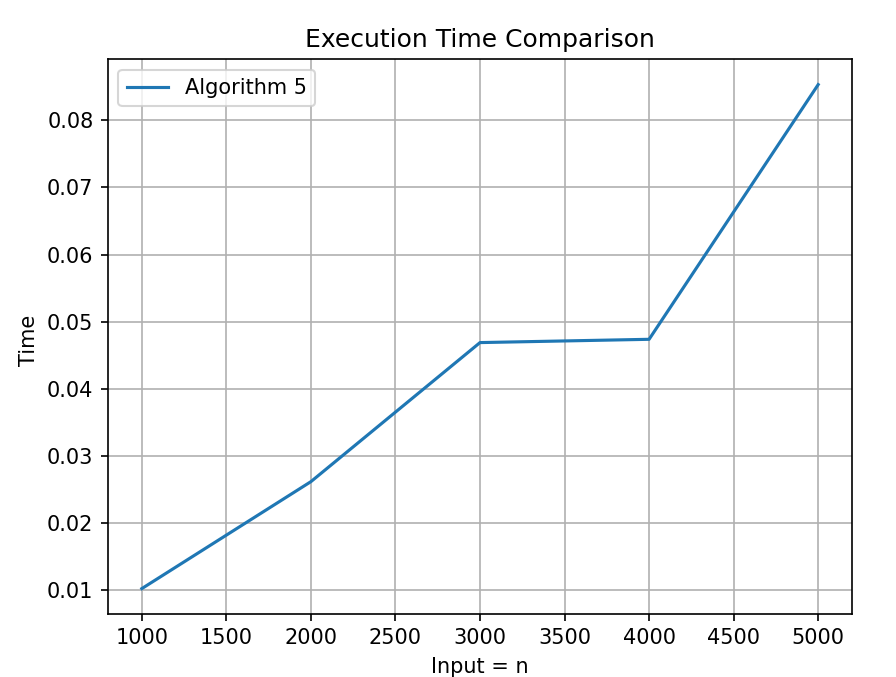
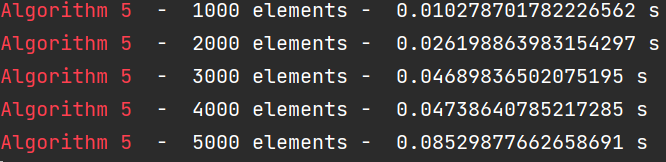




* **Algorithm 5**

This algorithm starts by creating a list of True values with length n+1, representing the numbers from 1 to n. It then sets c[0] and c[1] to False, since 0 and 1 are not prime, and initializes a loop variable, i, to 2. The algorithm then loops through all the numbers from 2 to n and checks if i is divisible by any number j from 2 to the square root of i. If i is divisible by j, the algorithm sets c[i] to False, since i is not prime. Finally, the algorithm returns the list of True/False values representing whether each number is prime or not.

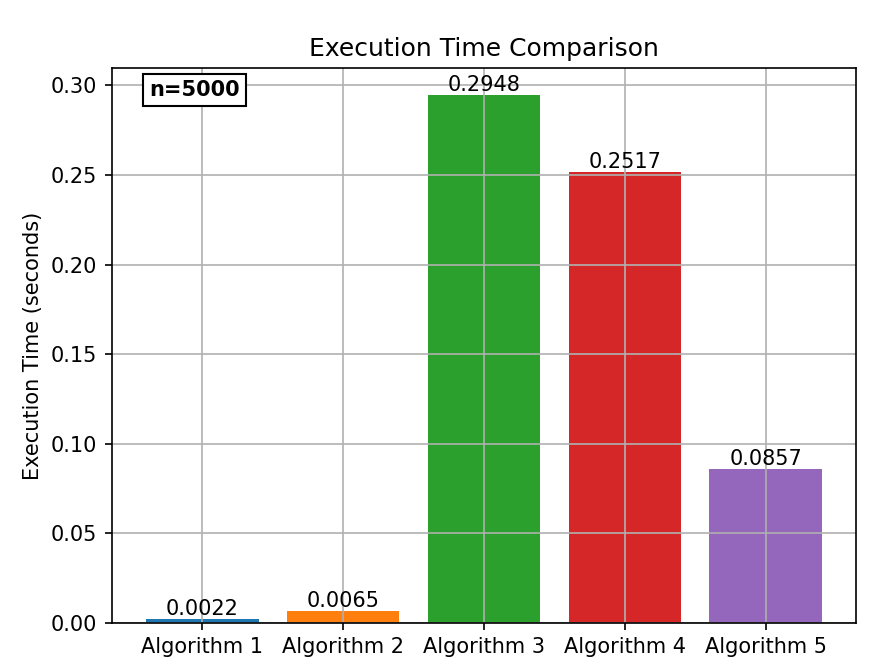
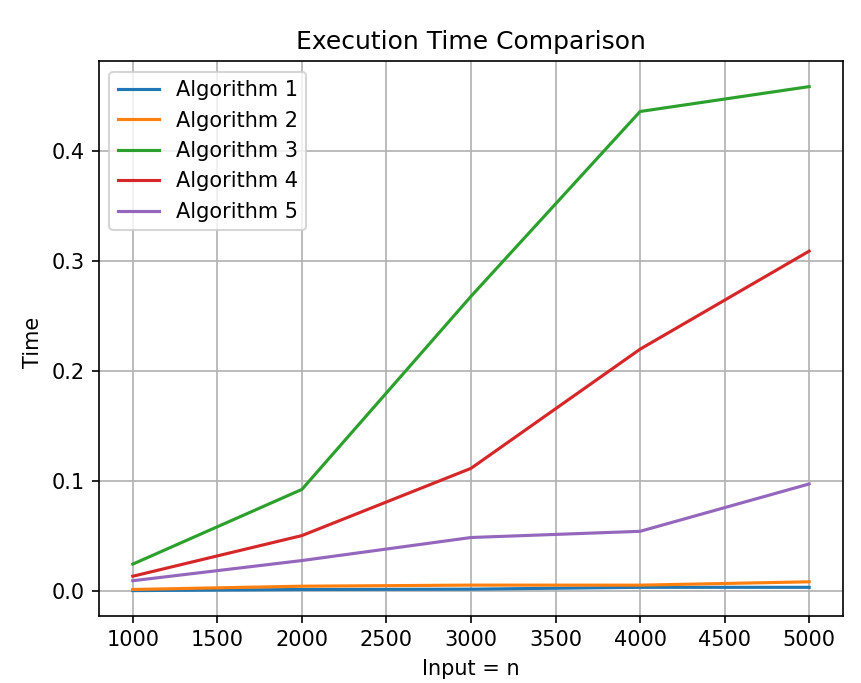
def algorithm\_5(n):  
 c = [True] \* (n + 1) # Create a list of n+1 elements and set all of them to True initially  
 c[0] = c[1] = False # Set the first two elements (0 and 1) to False as they are not prime numbers  
 i = 2 # Start with i=2, the first prime number  
 while (i <= n): # Continue until the given upper bound n  
 j = 2 # Start with j=2  
 while (j <= math.sqrt(i)): # Check if j is a divisor of i up to the square root of i  
 if (i % j == 0): # If j divides i, then i is not prime  
 c[i] = False # Mark c[i] as False  
 j += 1 # Increment j  
 i += 1 # Increment i  
 return c # Return the list of prime numbers

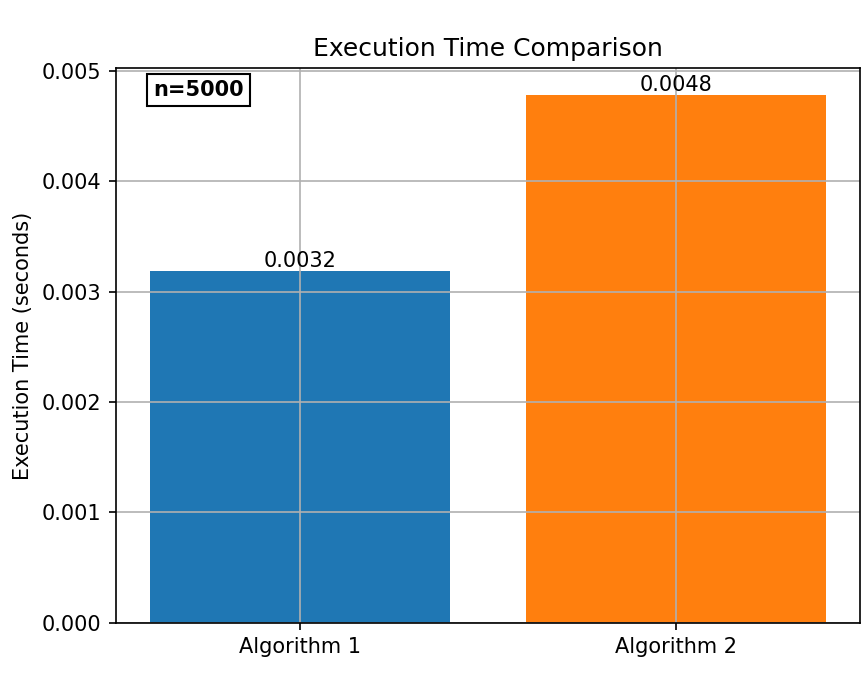
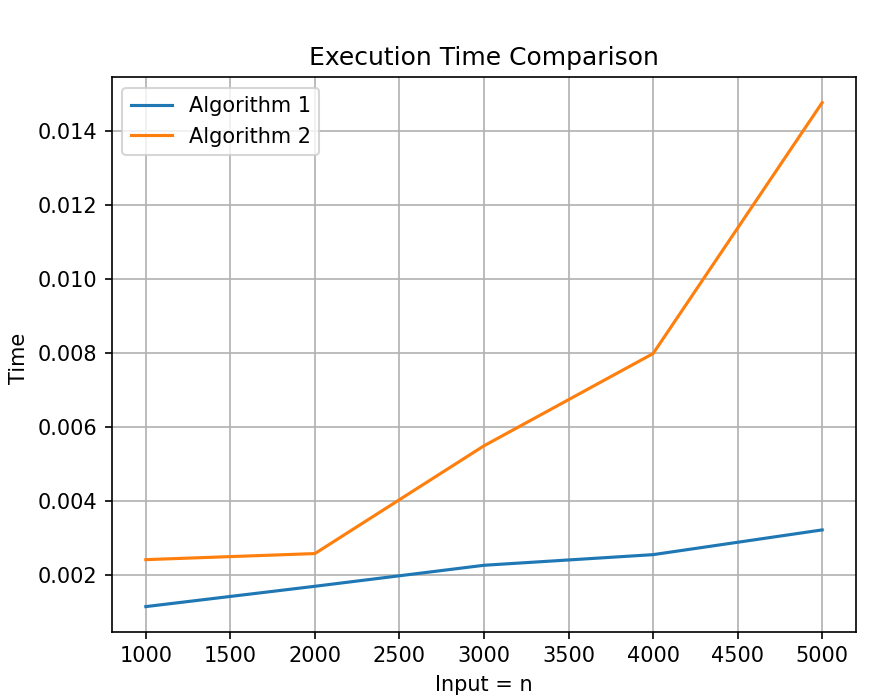
The **time complexity** of algorithm\_5 is. The outer loop iterates n times, and for each iteration, the inner loop checks for divisors up to the square root of i, which is approximately. Therefore, the total number of operations is roughly.

**Algorithms Comparison:**

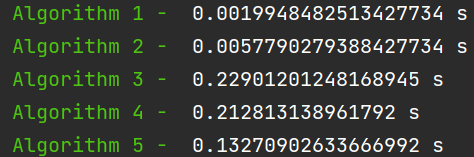
I will attach 3 cases, in which the program was executed, for an analysis of algorithms.

****

From the first graphs, we notice that algorithm 1 and 2 are the fastest and their time values are similar, so I will create a separate graph to be able to compare them more easily.

**A brief comparison of the Eratosthenes Sieve algorithms mentioned numbers from 1 to 5000:**

****

The five algorithms presented are all variations on the task of determining whether each number from 1 to n is prime or composite. Here are some of the key differences between them:

Algorithm 1 uses the Sieve of Eratosthenes, a well-known algorithm for finding all primes up to a given limit. It works by starting with a list of all numbers from 1 to n, marking 1 as composite and then looping through the remaining numbers. For each prime number found, all of its multiples are marked as composite.

Algorithm 2 is a slight variation on the Sieve of Eratosthenes. It starts with a list of all True values, marks 1 as composite, and then loops through the remaining numbers, marking all multiples of each prime number as composite.

Algorithm 3 is another variation on the Sieve of Eratosthenes. It starts with a list of all True values, marks 1 and 0 as composite, and then loops through the remaining numbers. For each prime number found, it loops through all numbers greater than the prime and marks them as composite if they are divisible by the prime.

Algorithm 4 is a more brute-force approach to finding primes. It starts with a list of all True values, marks 1 as composite, and then loops through the remaining numbers. For each number, it checks if it is divisible by any number less than itself. If it is, it is marked as composite.

Algorithm 5 is similar to Algorithm 4, but it uses the fact that any non-prime number must have a factor less than or equal to its square root. It starts with a list of all True values, marks 1 and 0 as composite, and then loops through the remaining numbers. For each number, it checks if it is divisible by any number less than or equal to its square root. If it is, it is marked as composite.

In summary, Algorithms 1-3 uses the Sieve of Eratosthenes or a similar approach, while Algorithms 4 and 5 use a more brute-force method. Algorithm 5 is slightly more efficient than Algorithm 4 because it only checks factors up to the square root of each number.

**Ranking:**

It's possible that **Algorithm 1** is the fastest for my implementation and device because it has a very efficient implementation of the Sieve of Eratosthenes. Algorithm 1 uses an array of Boolean values to mark the primes and composites, and it only needs to check multiples of primes up to the square root of n to mark all the composites. This means that it has a time complexity of , which is very efficient compared to the other algorithms. **Algorithm 2** and **Algorithm 5** are also Sieve of Eratosthenes algorithms, but they have slightly different implementations, which may be less efficient for your specific implementation and device. Algorithm 2 uses an inner loop to mark multiples of primes, while Algorithm 5 uses a loop to check for divisors up to the square root of i. Both of these loops can be more time-consuming than the inner loop used in Algorithm 1. **Algorithm 4** uses trial division to check if each number is prime, which is generally slower than the Sieve of Eratosthenes for large values of n. **Algorithm 3** is also using trial division, but with an additional loop that checks every number from to n for divisibility by i, making it slower than Algorithm 4.

**Conclusions:**

Based on my implementation and device, it seems that algorithm 1 is the fastest followed by algorithm 2, algorithm 5, algorithm 4, and algorithm 3 being the slowest. It's important to note that the performance of these algorithms can vary depending on the input size and hardware used.

Overall, the laboratory work has provided insight into different algorithms for finding prime numbers and how their time complexity can affect their performance. It's important to consider the trade-offs between time complexity and space complexity when selecting an algorithm for a specific task.

Additionally, the laboratory work has provided an opportunity to practice implementing algorithms and analyzing their performance, which can be useful skills for future projects.

<https://github.com/feliciaL3/APA_LABS/tree/main/LAB_3>