MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 3 : Empirical analysis of algorithms for obtaining Eratosthenes Sieve.

Elaborated:
st.gr. FAF-211 Gazea Sandu
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to a given limit. It was invented by the Greek mathematician Eratosthenes in the 3rd century BC and is considered one of the earliest known algorithms.

The algorithm works by creating a list of all numbers from 2 to the given limit. It then starts with the first prime number, which is 2, and marks all of its multiples as composite numbers. It then moves on to the next unmarked number, which is a prime, and repeats the process until all numbers up to the limit have been checked.

The algorithm is based on the fact that every composite number can be factored into prime factors. Therefore, if a number has already been marked as composite, then it must have a prime factor that is less than or equal to the square root of the number. This means that we only need to consider the prime numbers up to the square root of the given limit.

The Sieve of Eratosthenes is an efficient algorithm for finding prime numbers because it only needs to check each number once. The time complexity of the algorithm is O(n log log n), where n is the given limit. This means that the algorithm scales very well for large values of n.

The space complexity of the algorithm is also relatively low, as it only requires an array of size n to store the state of each number. The array is initially set to true for all numbers, indicating that they are potentially prime. As each multiple of a prime number is marked as composite, its corresponding entry in the array is set to false.

The Sieve of Eratosthenes has many applications in mathematics and computer science. For example, it can be used to find all prime numbers up to a certain limit, to factorize large numbers into their prime factors, and to generate prime numbers for use in cryptographic algorithms.

In conclusion, the Sieve of Eratosthenes is a fascinating algorithm that has stood the test of time and continues to be relevant today. Its efficiency, simplicity, and low memory requirements make it a popular choice for finding prime numbers in a variety of applications.

Objectives

1. To gain a practical understanding of how to implement and analyze algorithms using a programming language.
2. To develop skills in establishing the properties of input data, such as the size and distribution of prime numbers, and how these properties impact algorithm performance.
3. To explore various metrics and techniques for comparing algorithm performance, and gain insights into the strengths and weaknesses of each approach.
4. To gain experience in running experiments to empirically measure the performance of different algorithms and analyze the results.
5. To develop proficiency in presenting data using graphical visualization techniques, and gain insights into how to effectively communicate complex data to others.

These objectives reflect the broader goals of the laboratory work, which include building practical skills in algorithm analysis, developing critical thinking and problem-solving abilities, and fostering an appreciation for the nuances and complexities of algorithmic performance. Through this work, students can gain valuable experience in software development, data analysis, and scientific communication, which can help prepare them for a range of careers in computer science and related fields.

Algorithm 1

This is an implementation of the Sieve of Eratosthenes algorithm. It works by creating a boolean array of size n and initializing all elements to true, indicating that they are potentially prime. It then iterates through the array, starting with 2, and marks all multiples of each prime number as composite by setting their corresponding array element to false. The algorithm has a time complexity of O(n log log n) and a space complexity of O(n). Code:

```
            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;
}
```

Algorithm 2

This is also an implementation of the Sieve of Eratosthenes algorithm, but with a slightly simpler implementation. It works by creating a boolean array of size n and initializing all elements to true. It then iterates through the array, starting with 2, and marks all multiples of each prime number as composite. The algorithm has the same time and space complexities as Algorithm 1.

Code:

```
            C[1] =false;
i=2;
while (i<=n){
  j=2*i;
  while (j<=n){
    c[j] =false;
    j=j+i;
  }
  i=i+1;
}
```

Algorithm 3

This algorithm is similar to the Sieve of Eratosthenes, but with a different approach to marking composite numbers. It starts with a boolean array of size n and initializes all elements to true. It then iterates through the array, starting with 2, and for each prime number i, marks all multiples of i as composite by setting their corresponding array element to false. The algorithm has a time complexity of $O(n^2)$ $and a space complexity of O(n)$.

Code:

```
            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;
}
```

Algorithm 4

This algorithm uses a brute-force approach to determining prime numbers. It starts with a boolean array of size n and initializes all elements to true. It then iterates through the array, starting with 2, and for each number i, checks if any number less than i divides evenly into i. If so, i is marked as composite by setting its corresponding array element to false. The algorithm has a time complexity of $O(n^2)$ *andaspacecomplexityof* $O(n)$.

Code:

```
            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;
}
```

However, there is a mistake in the provided code since If j starts from 1, then i divided j will always be zero when j=1 for all i, resulting in all values in c to be marked as composite numbers, including the prime numbers.

Here is the corrected code:

```
            C[1] = false;
i = 2;
While (i<=n){
  j=2;
  while (j<i){
    if ( i % j == 0)
    {
      c[i] = false
    }
    j=j+1;
  }
  i=i+1;
}
```

Algorithm 5

This algorithm also uses a brute-force approach to determining prime numbers, but with a slight optimization. It starts with a boolean array of size n and initializes all elements to true. It then iterates through the array, starting with 2, and for each number i, checks if any number less than or equal to the square root of i divides evenly into i. If so, i is marked as composite by setting its corresponding array element to false. The algorithm has a time complexity of O(n sqrt(n)) and a space complexity of O(n).

Code:

```
  C[1] = faux;
i=2;
while (i<=n){
  j=2;
  while (j<=sqrt(i)){
    if (i % j == 0) {
      c[i] = false;
    }
    j++;
  }
  i++;
}
```

Implementation

```
import math
import time
import matplotlib.pyplot as plt
def sieve_of_eratosthenes_1(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        if c[i]:
            j = 2 * i
            while j <= n:
                c[j] = False
                j = j + i
        i = i + 1
    return [i for i in range(1, n+1) if c[i]]

def sieve_of_eratosthenes_2(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2 * i
        while j <= n:
            c[j] = False
            j = j + i
        i = i + 1
    return [i for i in range(1, n+1) if c[i]]

def sieve_of_eratosthenes_3(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
    return [i for i in range(1, n+1) if c[i]]

def sieve_of_eratosthenes_4(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2
        while j < i:
            if i % j == 0:
                c[i] = False
            j = j + 1
        i = i + 1
    return [i for i in range(1, n+1) if c[i]]



def sieve_of_eratosthenes_5(n):
    c = [True] * (n+1)
    c[1] = False
    i = 2
    while i <= n:
        j = 2
        while j <= math.sqrt(i):
            if i % j == 0:
```

```
                c[i] = False
            j = j + 1
        i = i + 1
    return [i for i in range(1, n+1) if c[i]]

print(sieve_of_eratosthenes_1(10000))
print(sieve_of_eratosthenes_2(10000))
print(sieve_of_eratosthenes_3(10000))
print(sieve_of_eratosthenes_4(10000))
print(sieve_of_eratosthenes_5(10000))



def time_it(func, *args):
    start_time = time.time()
    func(*args)
    end_time = time.time()
    return end_time - start_time

def plot_time_complexity_sep(algorithms, sizes):
    for i, algorithm in enumerate(algorithms):
        times = []
        for size in sizes:
            times.append(time_it(algorithm, size))
        plt.plot(sizes, times, label=f'Algorithm {i+1}')
        plt.legend()
        plt.xlabel('Input Size')
        plt.ylabel('Execution Time (s)')
        plt.show()
        plt.clf()

def plot_time_complexity(algorithms, sizes):
    for i, algorithm in enumerate(algorithms):
        times = []
        for size in sizes:
```

```
            times.append(time_it(algorithm, size))
        plt.plot(sizes, times, label=f'Algorithm {i+1}')
    plt.legend()
    plt.xlabel('Input Size')
    plt.ylabel('Execution Time (s)')
    plt.show()

sizes = range(100, 1100, 100)
algorithms = [
    sieve_of_eratosthenes_1,
    sieve_of_eratosthenes_2,
    sieve_of_eratosthenes_3,
    sieve_of_eratosthenes_4,
    sieve_of_eratosthenes_5
]

algorithm_1 = [sieve_of_eratosthenes_1]
algorithm_2 = [sieve_of_eratosthenes_2]
algorithm_3 = [sieve_of_eratosthenes_3]
algorithm_4 = [sieve_of_eratosthenes_4]
algorithm_5 = [sieve_of_eratosthenes_5]
plot_time_complexity(algorithms, sizes)
plot_time_complexity_sep(algorithms, sizes)
```

Screenshot:

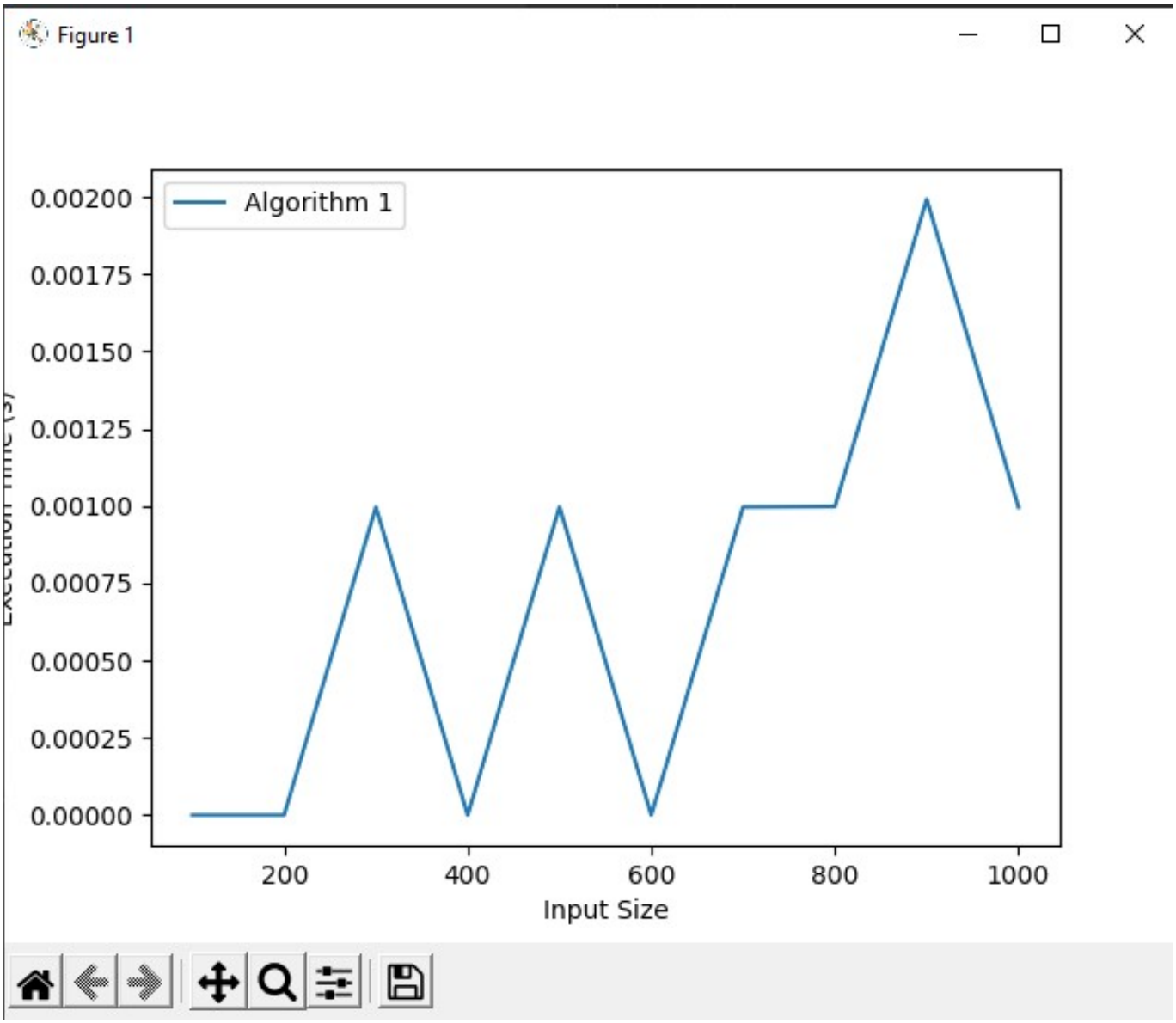
Figure 1
Algorithm 1
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
200
400
600
800
1000
Input Size

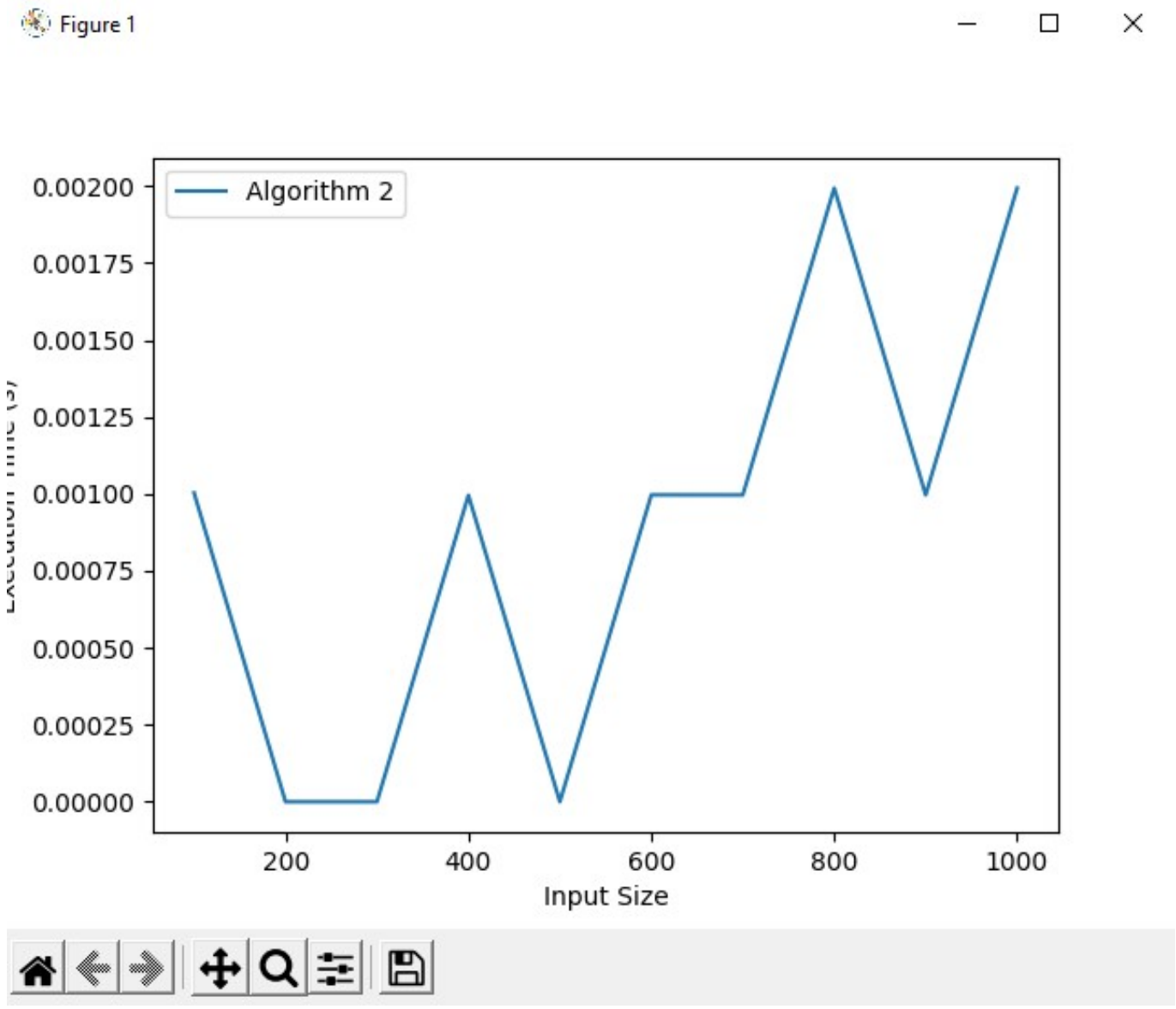
Figure 1
Algorithm 2
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
200
400
600
800
1000
Input Size

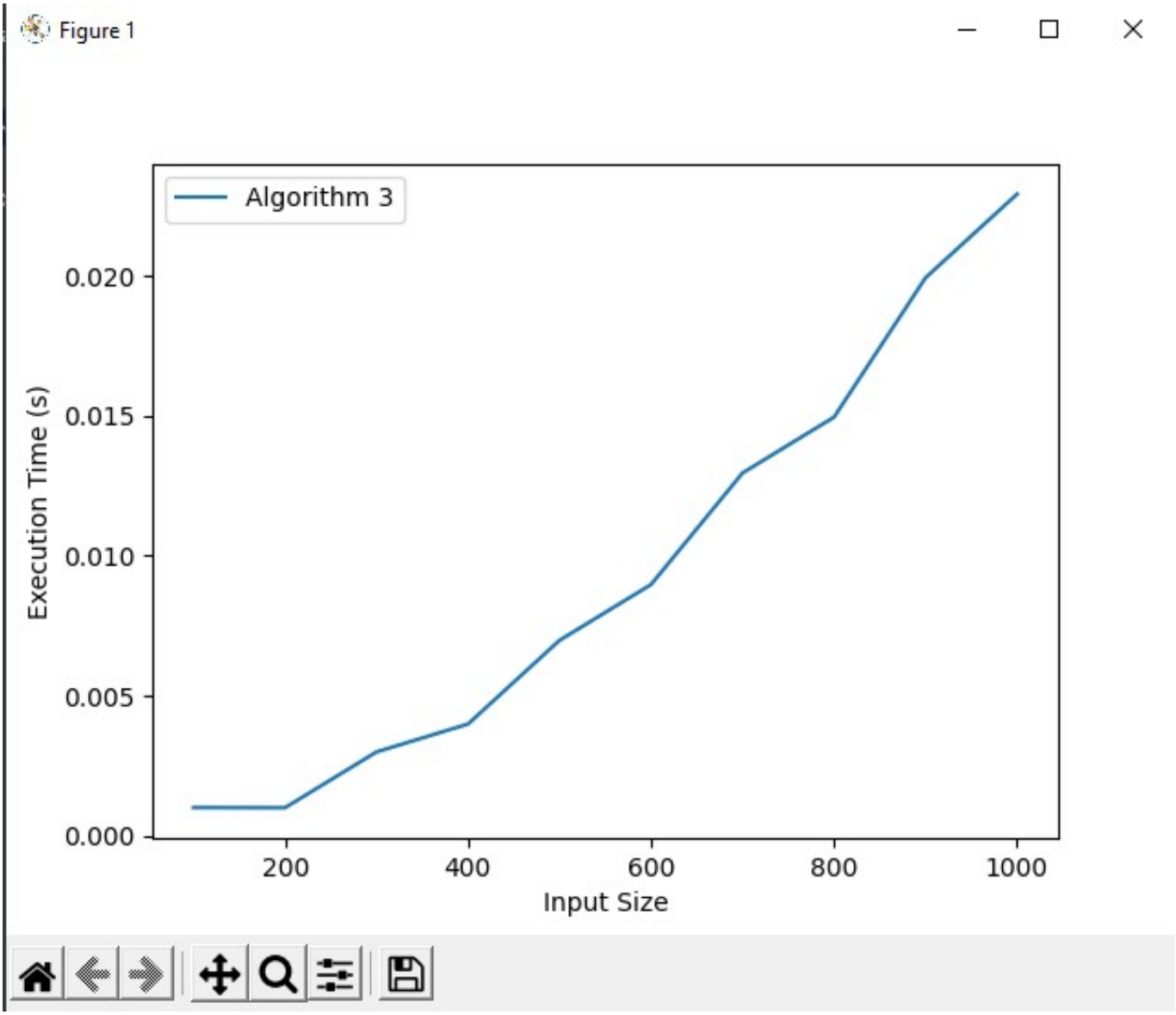
Figure 1
Algorithm 3
0.020
0.015
0.010
0.005
0.000
Execution Time (s)
200
400
600
800
1000
Input Size

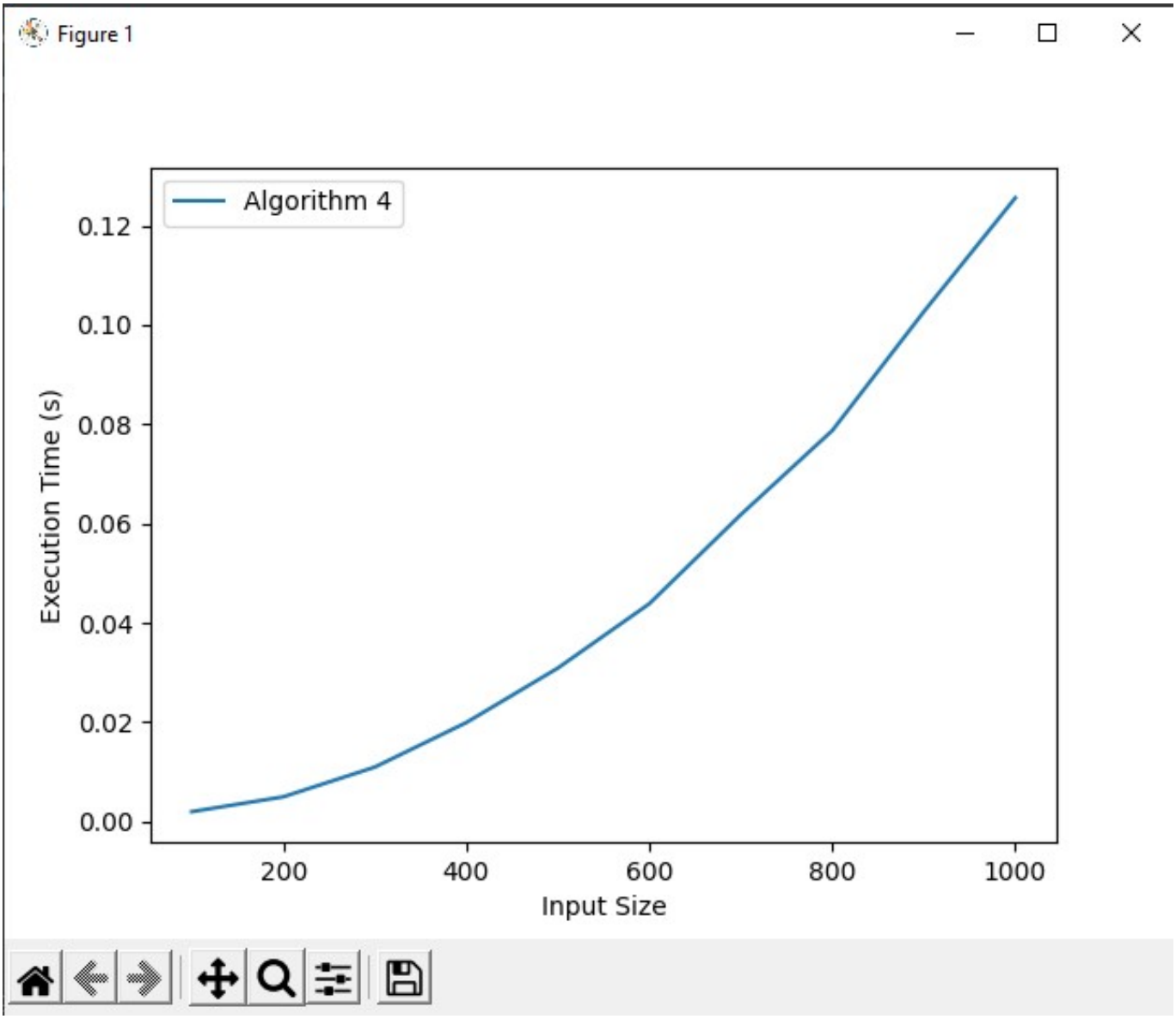
Figure 1
Algorithm 4
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Execution Time (s)
200
400
600
800
1000
Input Size

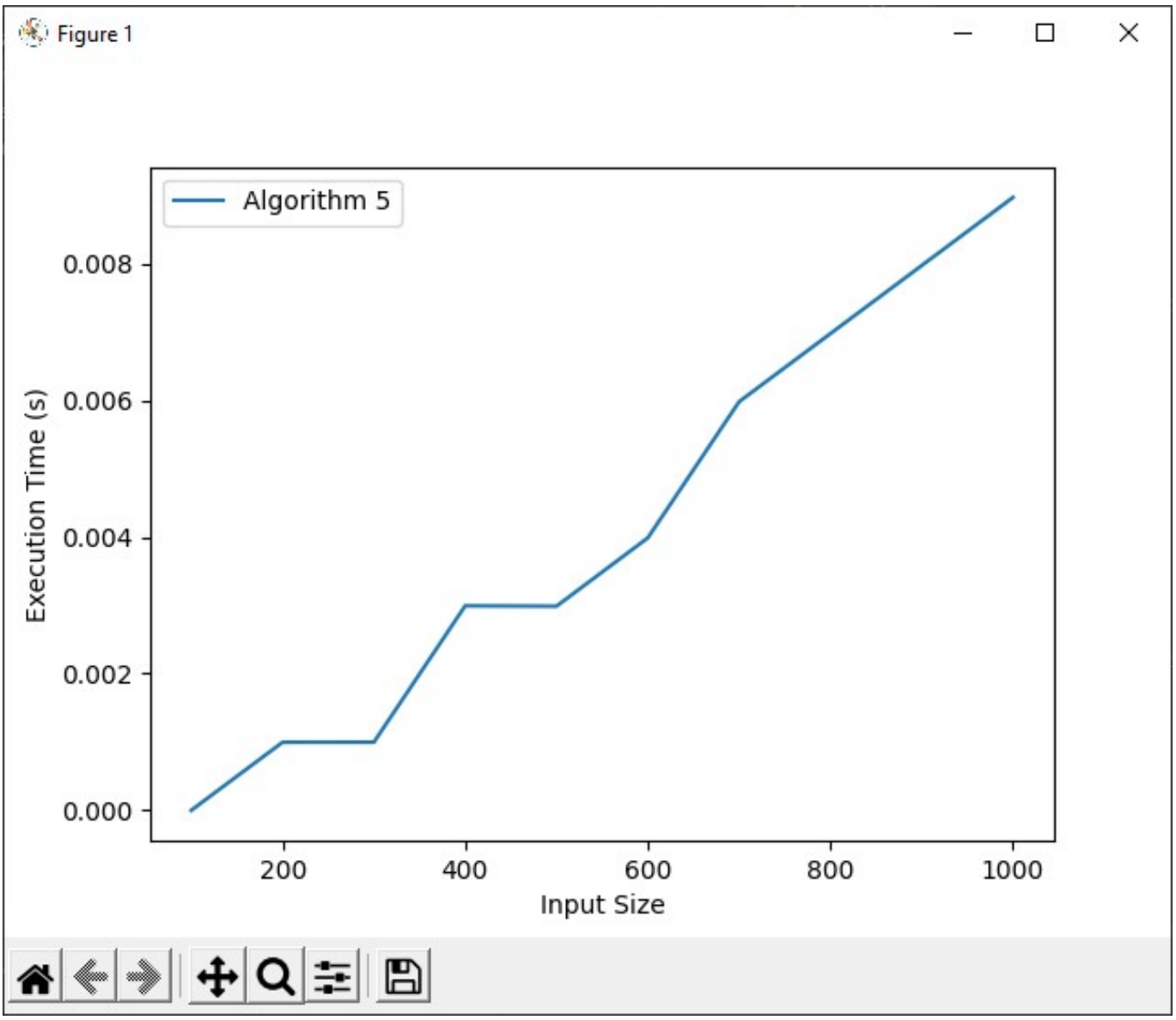
Figure 1
Algorithm 5
0.008
0.006
0.004
0.002
0.000
Execution Time (s)
200
400
600
800
1000
Input Size

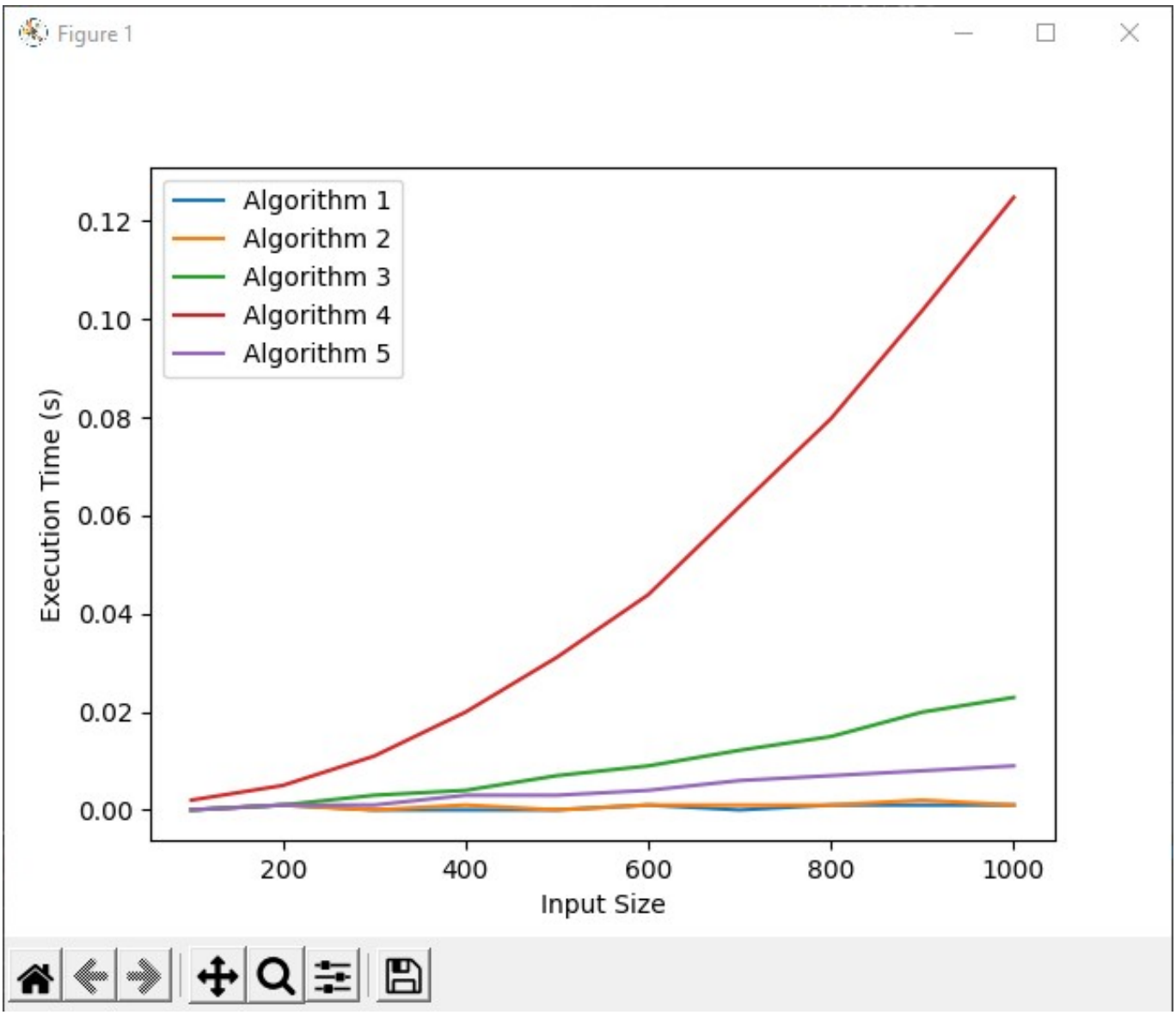
Figure 1
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 5
Execution Time (s)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
200
400
600
800
1000
Input Size

Conclusion

The presented algorithms offer various techniques to discover prime numbers up to a certain limit. Each algorithm has its own strengths and weaknesses, and the algorithm chosen will depend on the particular requirements of the problem.

Algorithms 1 and 2 are implementations of the Sieve of Eratosthenes algorithm, which is an extremely effective way to find prime numbers up to a limit. Both algorithms initialize a boolean array to track the prime numbers and then iterate through the array to mark the multiples of each prime number as composite. However, Algorithm 1 optimizes the process by marking only multiples of prime numbers as composite, making it more efficient.

Algorithm 3 takes a different approach by iterating through all integers from 2 to n, checking for divisibility by any prime number before marking it as composite. This algorithm is less efficient than Algorithms 1 and 2 but may be practical for smaller values of n.

Algorithm 4 is a brute-force method that iterates through all integers from 2 to n, checking for divisibility by any integer between 1 and itself. This approach is the least efficient of the presented algorithms.

Algorithm 5 is another brute-force method but with an optimization that checks only integers up to the square root of the integer being tested for divisibility. This optimization reduces the number of iterations required and makes the algorithm more efficient.

In conclusion, the choice of algorithm depends on the specific needs of the problem. The Sieve of Eratosthenes algorithm is the most efficient for finding prime numbers up to a certain limit, but for larger values of n, other algorithms may also be practical options. For smaller values of n, Algorithm 3, 4, and 5 may also be useful depending on the specific requirements of the problem.

Github repository: https://github.com/GSandu1/Lab3AA.git