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 Corețchi Mihai

Verified:

asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Scope: Empirically analyze different algorithms for obtaining Eratosthenes Sieve.

Objectives:

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.

The Sieve of Eratosthenes is a historic algorithm dating back to the 3rd century BC that allows us to identify all prime numbers up to a specified limit. The algorithm, which was invented by the Greek mathematician Eratosthenes, works by creating a list of all numbers from 2 to the given limit and then marking all multiples of the first prime number, which is 2, as composite numbers. It then proceeds 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, resulting in a more efficient algorithm for identifying prime numbers.

The Sieve of Eratosthenes has a time complexity of O(n log log n), where n is the given limit, which means that the algorithm scales very well for large values of n. Additionally, the space complexity of the algorithm is 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 practical applications in mathematics and computer science. For example, it can be used to find all prime numbers up to a certain limit, which is important in number theory and cryptography. It can also be used to factorize large numbers into their prime factors, which is essential in modern cryptography systems.

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 $O(nloglogn)$ 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 $O(n^2)$ and a space complexity of 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.

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 $O(nsqrt(n))$ 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

Code:

```
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]]
```

Screenshot:

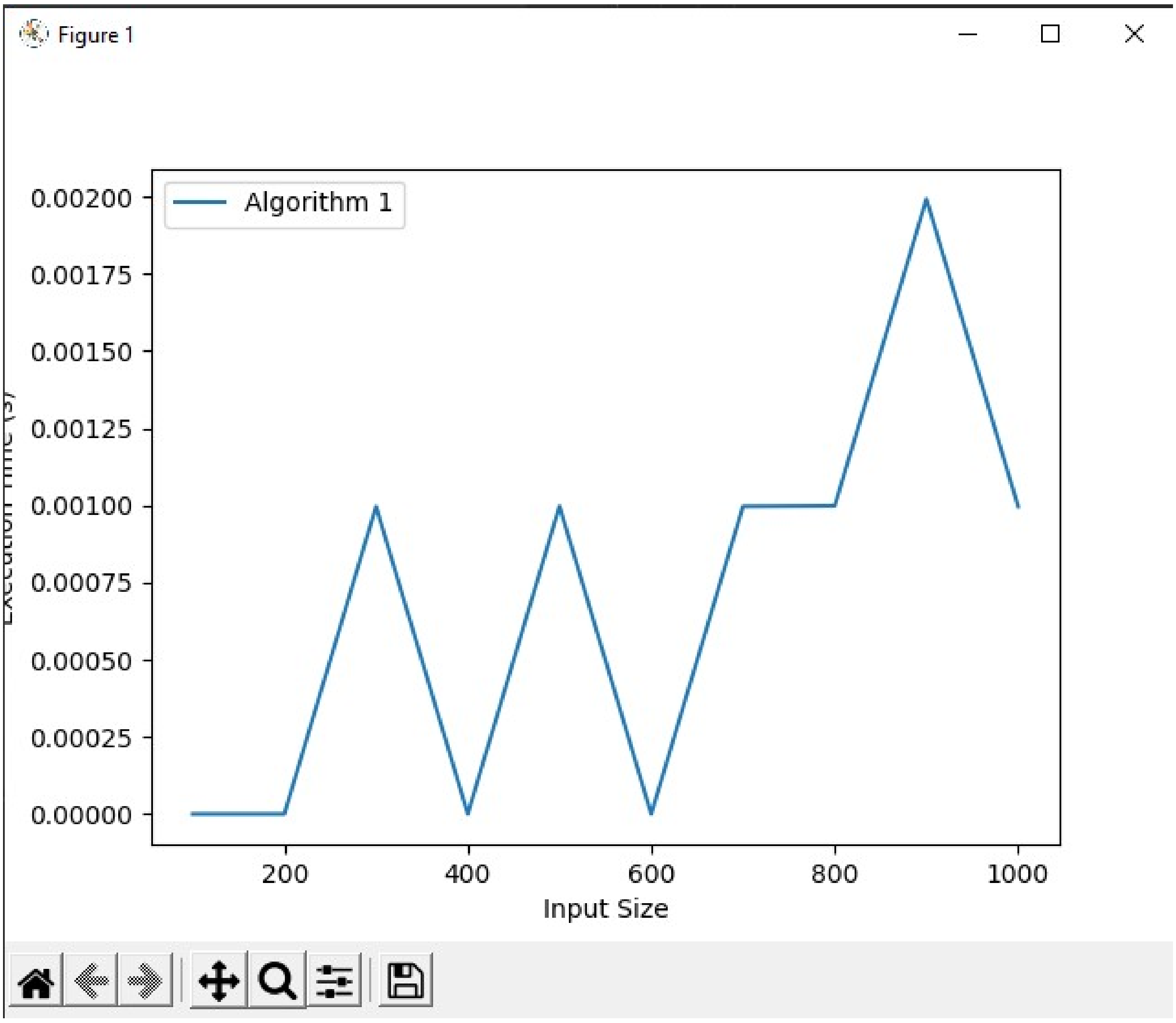

Figure 1
Algorithm 1
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
200
400
600
800
1000
Input Size

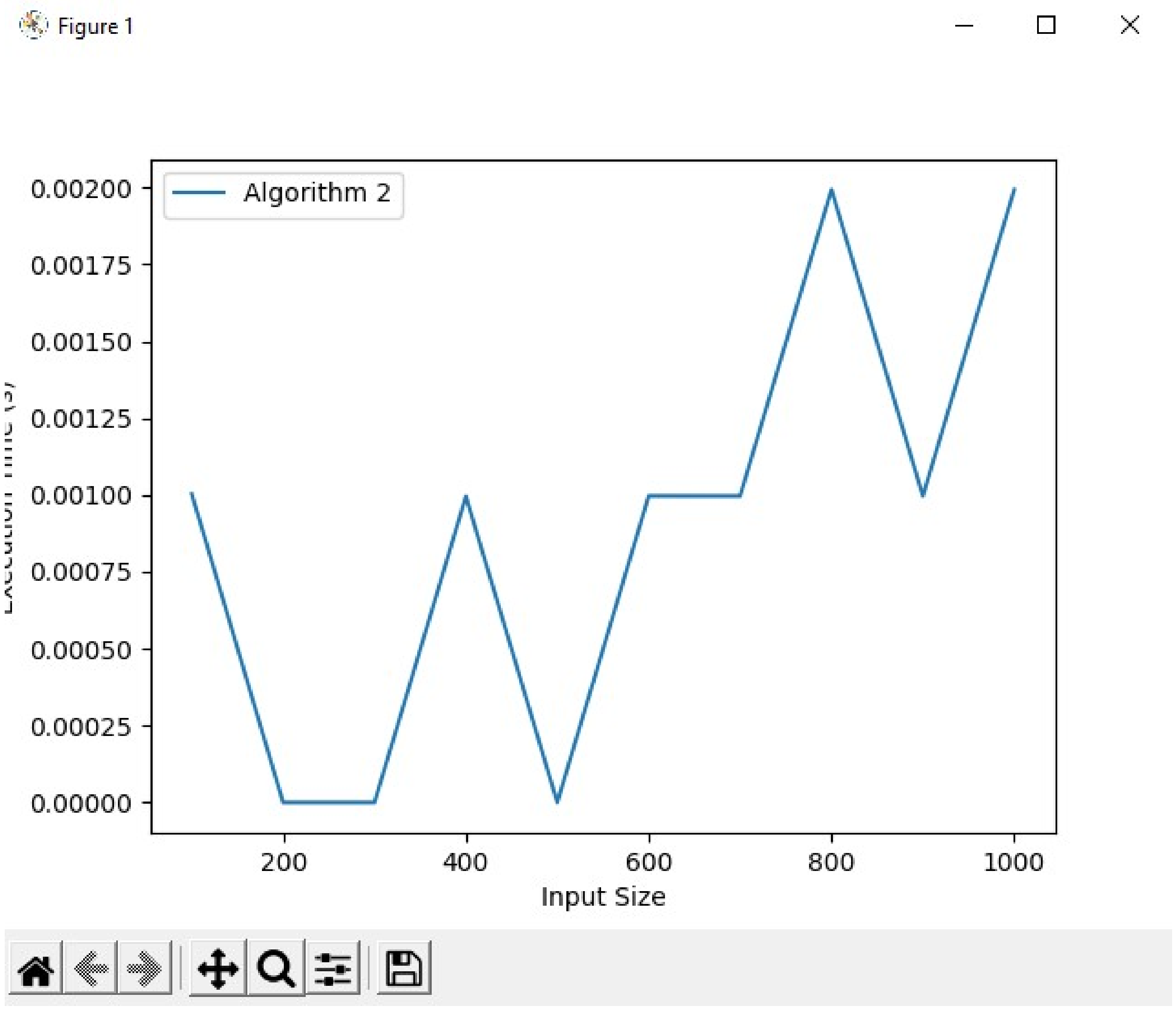
Figure 1
Algorithm 2
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
200
400
600
800
1000
Input Size

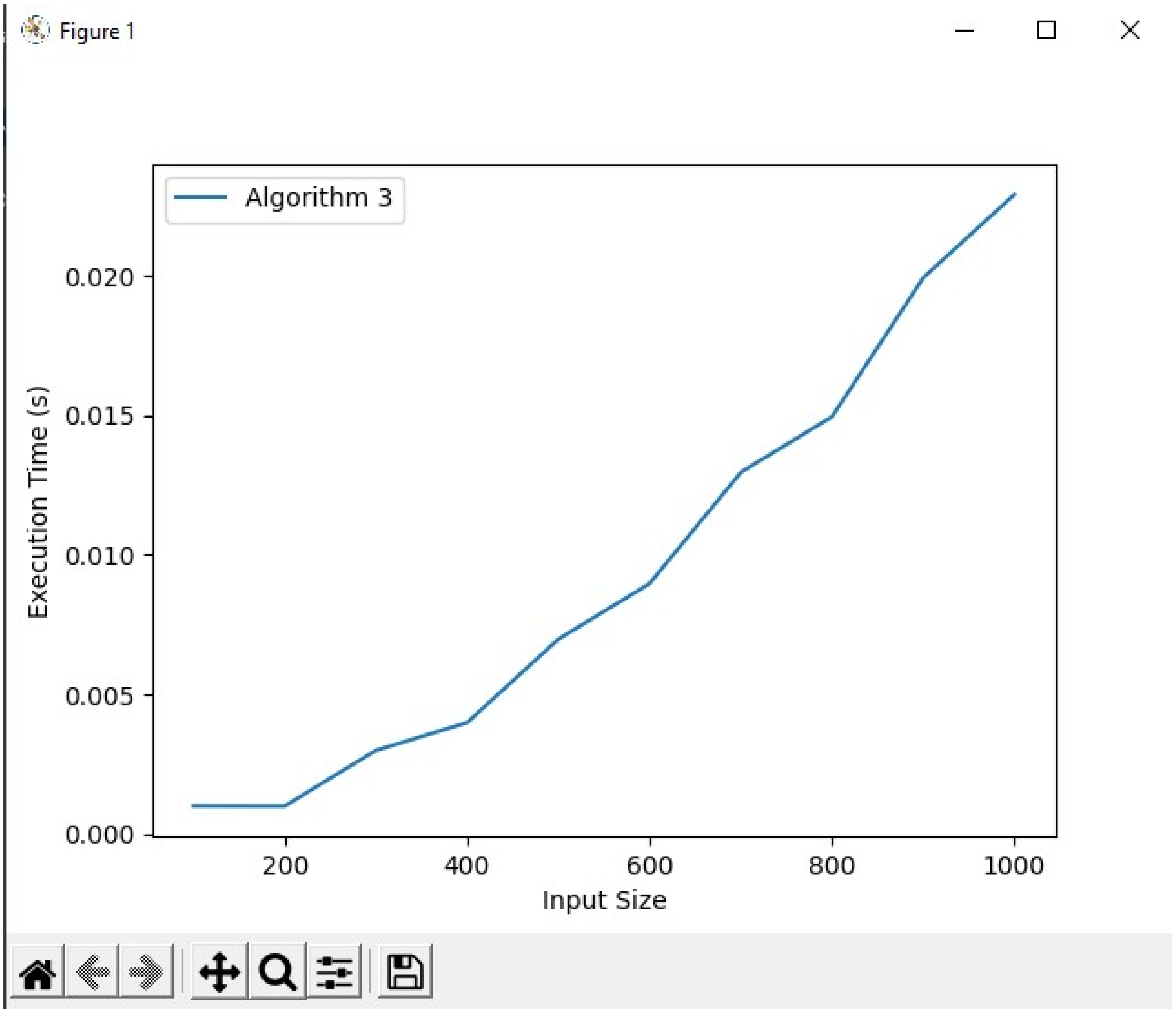
Figure 1
Algorithm 3
0.000
0.005
0.010
0.015
0.020
Execution Time (s)
200
400
600
800
1000
Input Size

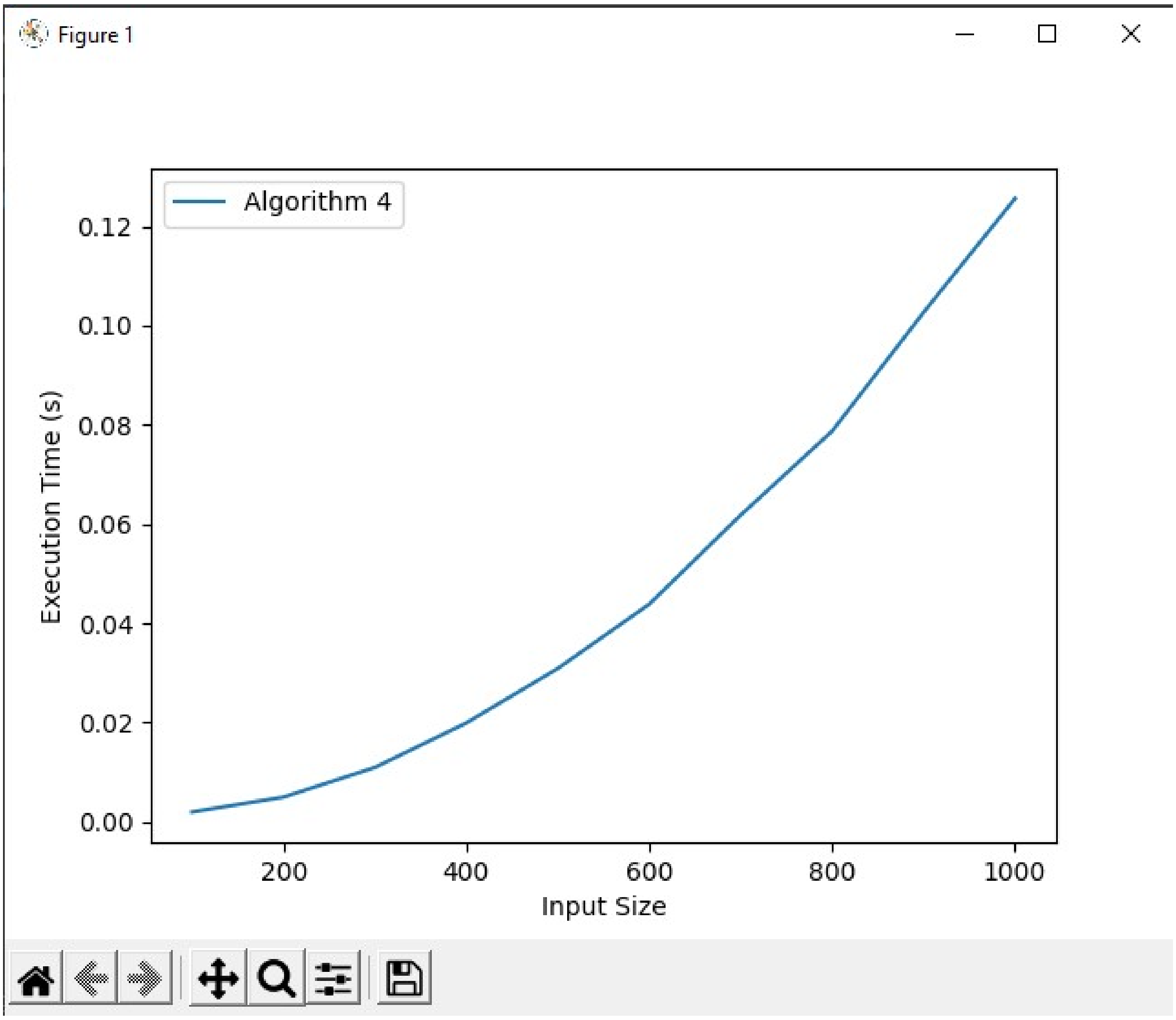
Figure 1
Algorithm 4
Execution Time (s)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
200
400
600
800
1000
Input Size

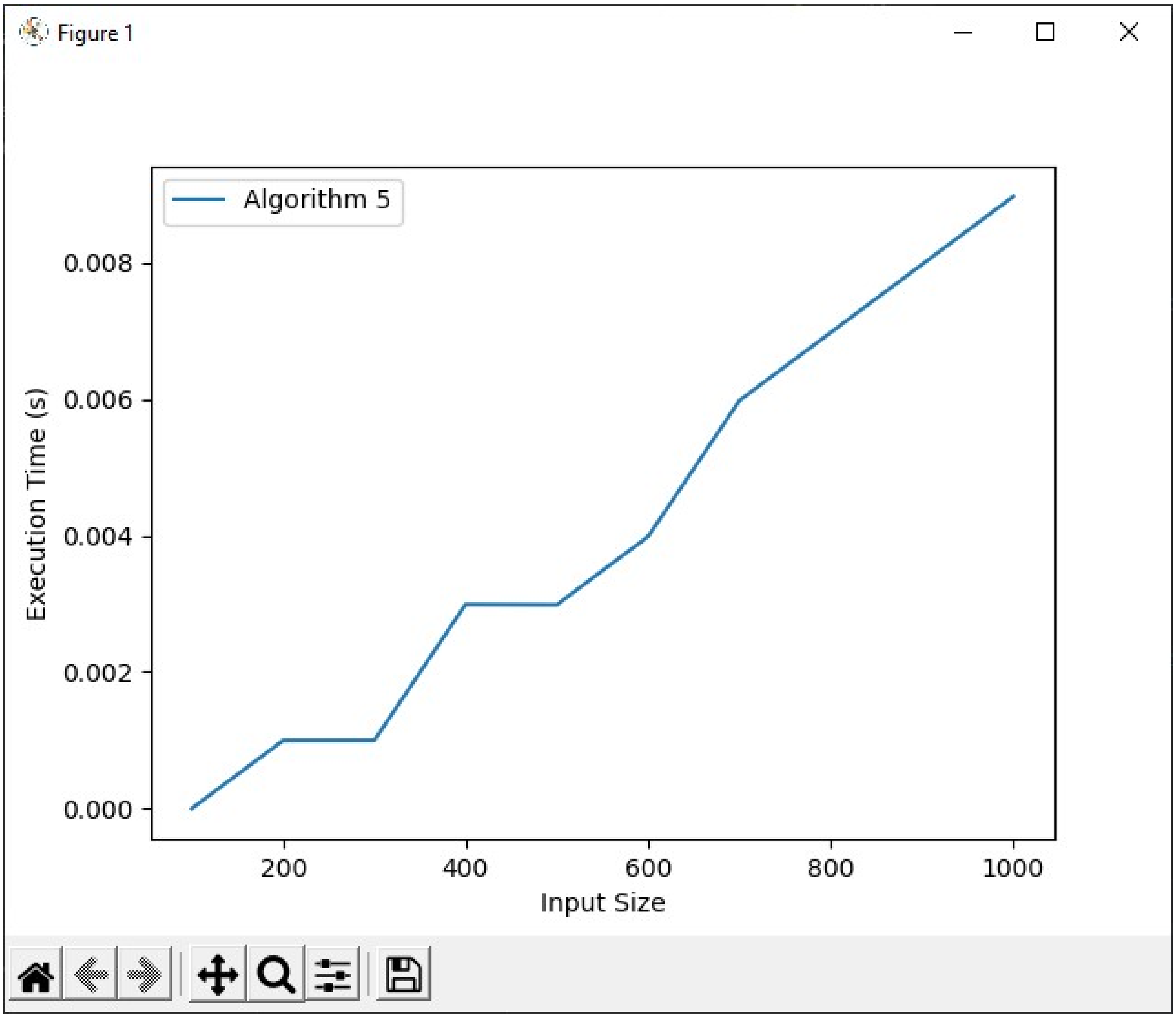
Figure 1
Algorithm 5
Execution Time (s)
0.000
0.002
0.004
0.006
0.008
200
400
600
800
1000
Input Size

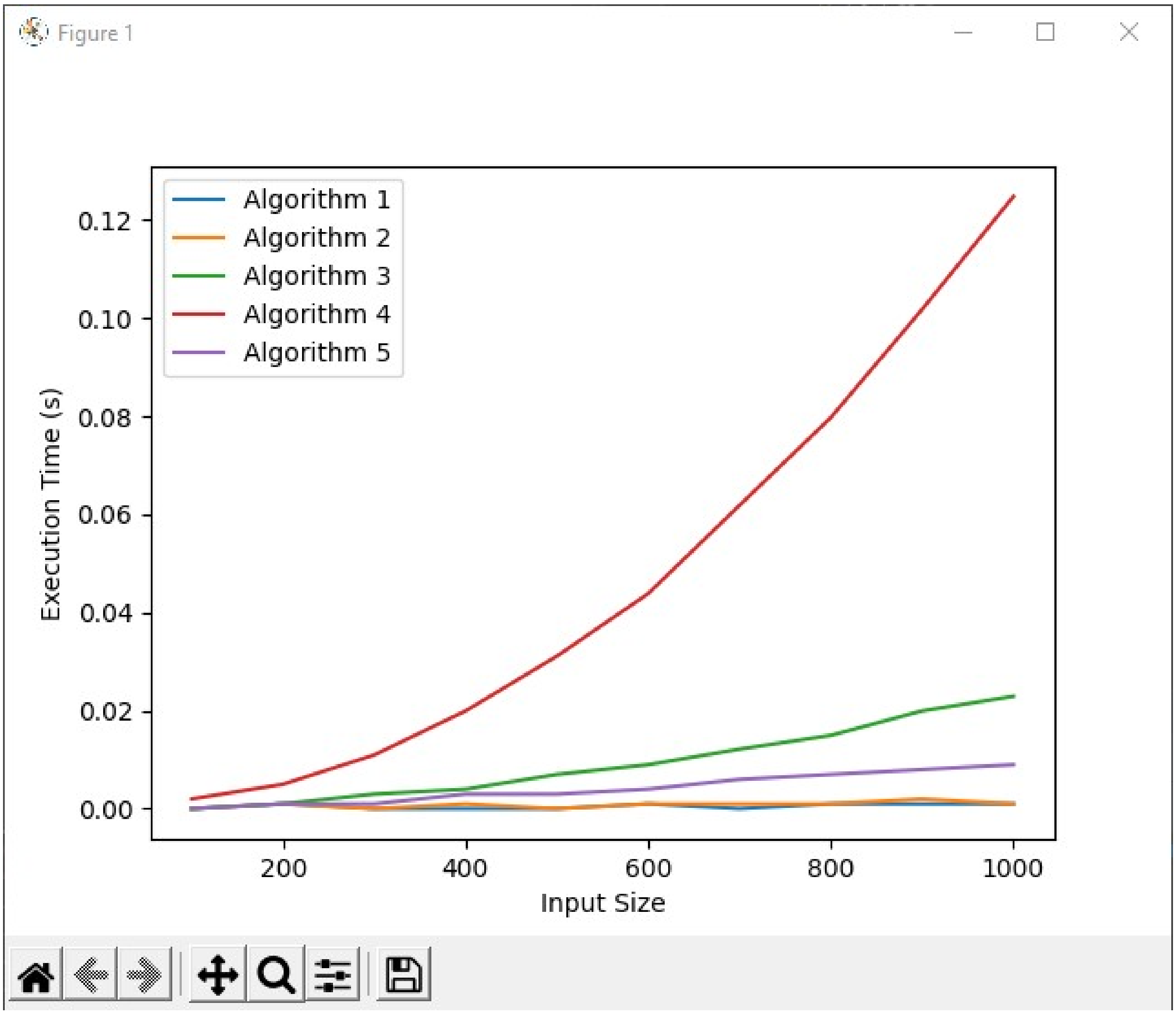
Figure 1
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 5
Execution Time (s)
0.12
0.10
0.08
0.06
0.04
0.02
0.00
200
400
600
800
1000
Input Size

Conclusion

Different algorithms are presented to find prime numbers up to a specific limit, each having its own advantages and limitations. The selection of an algorithm depends on the problem's specific requirements.

The first two algorithms implement the Sieve of Eratosthenes algorithm, which is highly efficient for finding prime numbers up to a certain limit. Both algorithms initialize a boolean array c of length n, where c[i] represents whether or not i is prime. The algorithm then marks all multiples of each prime number as composite. However, Algorithm 1 only marks multiples of prime numbers as composite, which makes it more efficient than Algorithm 2.

Algorithm 3 takes a different approach and iterates through all integers from 2 to n, checking if each integer is divisible by any prime number before marking it as composite. It doesn't use a boolean array to keep track of the prime numbers.

Algorithm 4 is a brute-force approach and checks if each integer between 2 and n is divisible by any integer between 1 and itself. If an integer is divisible by any integer other than 1 and itself, it is marked as composite.

Algorithm 5 is another brute-force approach, but with an optimization. It only checks 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 requirements of the problem. The Sieve of Eratosthenes is highly efficient for finding prime numbers up to a certain limit, but for larger values of n, other algorithms may also be practical options. Algorithm 3, 4, and 5 are less efficient than Algorithm 1 and 2, but may be practical options for smaller values of n. Ultimately, the choice of algorithm will depend on the specific needs of the problem at hand.

Github repository: https://github.com/eamtcPROG/AA/tree/main/Lab3