Ministerul Educației 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-213 Bardier Andrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyze different sorting algorithms, including merge sort, quick sort, heap sort and one for my choice.

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.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis, which can provide useful information about the efficiency of algorithms. The stages of empirical analysis typically include:

1. Establishing the purpose of the analysis.
2. Choosing an efficiency metric, such as the number of operations executed or the execution time of the algorithm or a portion of it.
3. Defining the properties of the input data that the analysis will be based on, such as size or specific properties.
4. Implementing the algorithm in a programming language.
5. Generating multiple sets of input data.
6. Running the program for each set of input data.
7. Analyzing the obtained data.

The choice of the efficiency measure depends on the purpose of the analysis. For example, if the goal is to obtain information about the complexity class or verify a theoretical estimate, the number of operations performed is appropriate, while the execution time is more relevant if the aim is to evaluate the implementation of the algorithm.

It is important to use multiple sets of input data to reflect the different characteristics of the algorithm, covering a range of sizes and different values or configurations of input data. To obtain meaningful results, it is crucial to use input data that reflects the range of situations the algorithm will encounter in practice.

To perform empirical analysis of an algorithm's implementation in a programming language, monitoring sequences are introduced to measure the efficiency metric. If the efficiency metric is the number of operations executed, a counter is used to increment after each execution. If the metric is execution time, the time of entry and exit must be recorded. Most programming languages have functions to measure the time elapsed between two moments, and it is important to only count the time affected by the execution of the analyzed program, especially when measuring time.

After the program has been run for the test data, the results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated or the results are represented graphically as pairs of points in the form of (problem size, efficiency measure).

Introduction:

The Sieve of Eratosthenes is an ancient algorithm named after the Greek mathematician Eratosthenes. It is an efficient and simple way to find all prime numbers up to a given limit. The algorithm starts by creating a list of all numbers up to the limit, then iteratively crossing out all multiples of each prime number found until only primes are left.

The Sieve of Eratosthenes has numerous applications in computer science and algorithms. One of its most common uses is in cryptography, where prime numbers play a critical role in generating secure encryption keys. The algorithm can quickly generate a list of all prime numbers up to a given limit, making it easier to find large prime numbers for encryption purposes.

The Sieve of Eratosthenes is also useful in number theory, where prime numbers are central to many important theorems and conjectures. The algorithm can generate a list of all primes up to a given limit, which can be used to study the distribution and properties of prime numbers.

In computer science, the Sieve of Eratosthenes is often used as a benchmark for testing the efficiency of other algorithms. Because the algorithm is relatively simple and efficient, it provides a good baseline for comparing the performance of more complex algorithms for finding prime numbers.

Despite its simplicity, the Sieve of Eratosthenes remains a widely used algorithm today and has inspired many other algorithms for finding prime numbers. The Sieve of Sundaram and the Sieve of Atkin are both based on similar principles as the Sieve of Eratosthenes but use different strategies for crossing out multiples of prime numbers.

In conclusion, the Sieve of Eratosthenes is a powerful algorithm with a variety of applications in cryptography, number theory, and computer science. Its simplicity and efficiency make it an important tool for finding prime numbers and a benchmark for testing other algorithms. Despite being over 2,000 years old, the Sieve of Eratosthenes remains a fundamental algorithm in modern computer science, a testament to the lasting legacy of ancient Greek mathematics.

Comparison Metric:

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

Input Format:

As input, each algorithm will receive a series of numbers from 5’000 to 70’000, with step 2’500.

In other words, our input will be an array, containing these elements:

[5000, 7500, 10000, 12500, 15000, 17500, 20000, 22500, 25000, 27500, 30000, 32500, 35000, 37500, 40000, 42500, 45000, 47500, 50000, 52500, 55000, 57500, 60000, 62500, 65000, 67500, 70000]

IMPLEMENTATION

The five algorithms will be implemented without optimization in Python and evaluated based on their completion time. The results obtained may resemble those of other similar experiments, but the specific performance in relation to the input size may vary depending on the memory specifications of the device being used.

First algorithm

The algorithm presented is a simple and efficient way to generate all prime numbers up to a given positive integer n. It is called the Sieve of Eratosthenes, and it works by iteratively marking the multiples of each prime number starting from 2.

The algorithm initializes an array c of size n+1 with all elements set to True. It then sets c[1] to False, as 1 is not a prime number. The algorithm starts with i = 2, which is the first prime number, and iterates through the array. For each prime number i, it iterates through its multiples j = 2i, 3i, 4*i, ..., up to n, and marks them as not prime by setting their corresponding element in the array to False.

At the end of the algorithm, the array c contains True values only for prime numbers up to n. This array can be used to efficiently test whether a given number is prime or not, by simply checking whether its corresponding element in the array is True or False.

The time complexity of the Sieve of Eratosthenes algorithm is O(n log log n), which is significantly faster than other prime number generation algorithms, such as trial division or Fermat's little theorem. Therefore, it is a popular choice for generating prime numbers in computer programs.

```python
def algorithm_1(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:
        if c[i]:
            j = 2 * i
            while j <= n:
                c[j] = False
                j = j + i
        i = i + 1
    return c
```

Second algorithm

The algorithm presented is similar to the Sieve of Eratosthenes, but with a small modification. Instead of checking whether each number is prime, it simply iterates through all numbers from 2 to n and marks their multiples as not prime.

The algorithm initializes an array c of size n+1 with all elements set to True. It then sets c[1] to False, as 1 is not a prime number. The algorithm starts with i = 2, which is the first prime number, and iterates through all numbers from 2 to n. For each number i, it iterates through its multiples j = 2i, 3i, 4*i, ..., up to n, and marks them as not prime by setting their corresponding element in the array to False.

At the end of the algorithm, the array c contains True values only for prime numbers up to n. However, unlike the Sieve of Eratosthenes, the algorithm also marks some composite numbers as prime. For example, when i = 4, it marks 8 as not prime, but when i = 8, it marks 8 as prime. This means that the algorithm is less efficient than the Sieve of Eratosthenes, as it performs more unnecessary iterations.

The time complexity of this algorithm is also O(n log n), which is slower than the Sieve of Eratosthenes. Therefore, it is not a popular choice for generating prime numbers in computer programs. However, it can be useful in situations where a simple prime number generator is needed, and the number n is relatively small.

```python
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
        while j <= n:
            c[j] = False
            j = j + i
        i = i + 1
    return c
```

Third algorithm

The algorithm presented is a naive approach to generate all prime numbers up to a given positive integer n. It works by iteratively checking each number from 2 to n, and marking all its multiples as not prime.

The algorithm initializes an array c of size n+1 with all elements set to True. It then sets c[1] to False, as 1 is not a prime number. The algorithm starts with i = 2, which is the first prime number, and iterates through all numbers from 2 to n. For each prime number i, it iterates through all numbers from i+1 to n, and marks any multiple of i as not prime by setting their corresponding element in the array to False.

At the end of the algorithm, the array c contains True values only for prime numbers up to n. However, the algorithm is much less efficient than the previous two algorithms presented. It performs unnecessary iterations by checking non-multiples of i, and by checking multiples of i that have already been marked as not prime.

The time complexity of this algorithm is $O(n^2)$, which makes it impractical for generating prime numbers for large values of n. It can only be used for small values of n, or as a simple demonstration of the concept of prime number generation.

In summary, while algorithm 3 is a simple approach to generating prime numbers, it is significantly less efficient than other prime number generation algorithms, such as the Sieve of Eratosthenes or Algorithm 2 presented earlier.

```python
def algorithm_3(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:
        if c[i]:
            j = i + 1
            while j <= n:
                if j % i == 0:
                    c[j] = False
                j = j + 1
        i = i + 1

    return c
```

Fourth algorithm

The algorithm presented is a naive approach to generating all prime numbers up to a given positive integer n. It works by checking each number from 2 to n, and determining if it is a prime number by iterating through all integers from 2 to i-1 and checking if any of them divide i evenly.

The algorithm initializes an array c of size n+1 with all elements set to True. It then sets c[1] to False, as 1 is not a prime number. The algorithm starts with i = 2, which is the first prime number, and iterates through all numbers from 2 to n. For each number i, it iterates through all numbers from 2 to i-1 and checks if i is divisible by any of them. If i is divisible by any number between 2 and i-1, it is marked as not prime by setting the corresponding element in the array c to False.

At the end of the algorithm, the array c contains True values only for prime numbers up to n. However, the algorithm is significantly less efficient than the previous two algorithms presented, as it performs many unnecessary iterations. For example, if i is a composite number, the algorithm will still check all numbers from 2 to i-1, even though it has already found a factor of i.

The time complexity of this algorithm is O(n^2), which makes it impractical for generating prime numbers for large values of n. It can only be used for small values of n, or as a simple demonstration of the concept of prime number generation.

In summary, while algorithm 4 is a straightforward approach to generating prime numbers, it is significantly less efficient than other prime number generation algorithms, such as the Sieve of Eratosthenes or Algorithm 2 presented earlier.

```python
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
                break
            j = j + 1
        if not is_prime:
            c[i] = False
        i = i + 1

    return c
```

Fifth algorithm

The algorithm presented is an improvement on Algorithm 4 for generating all prime numbers up to a given positive integer n. Similar to Algorithm 4, it iterates through all numbers from 2 to n and checks if they are prime by iterating through all numbers from 2 to the square root of i and checking if any of them divide i evenly.

The algorithm initializes an array c of size n+1 with all elements set to True. It then sets c[1] to False, as 1 is not a prime number. The algorithm starts with i = 2, which is the first prime number, and iterates through all numbers from 2 to n. For each number i, it iterates through all numbers from 2 to the square root of i and checks if i is divisible by any of them. If i is divisible by any number between 2 and the square root of i, it is marked as not prime by setting the corresponding element in the array c to False.

The key improvement of this algorithm over Algorithm 4 is that it only checks divisors up to the square root of i, rather than iterating through all numbers up to i-1. This is because any factor of i that is greater than the square root of i must have a corresponding factor that is less than the square root of i. Therefore, if i is not divisible by any numbers up to the square root of i, it is guaranteed to be a prime number.

The time complexity of this algorithm is O($n^{1.5}$), which is significantly better than Algorithm 4, but still not as efficient as the Sieve of Eratosthenes or Algorithm 2 presented earlier. It can be used for generating prime numbers for medium-sized values of n, but is not suitable for large values of n.

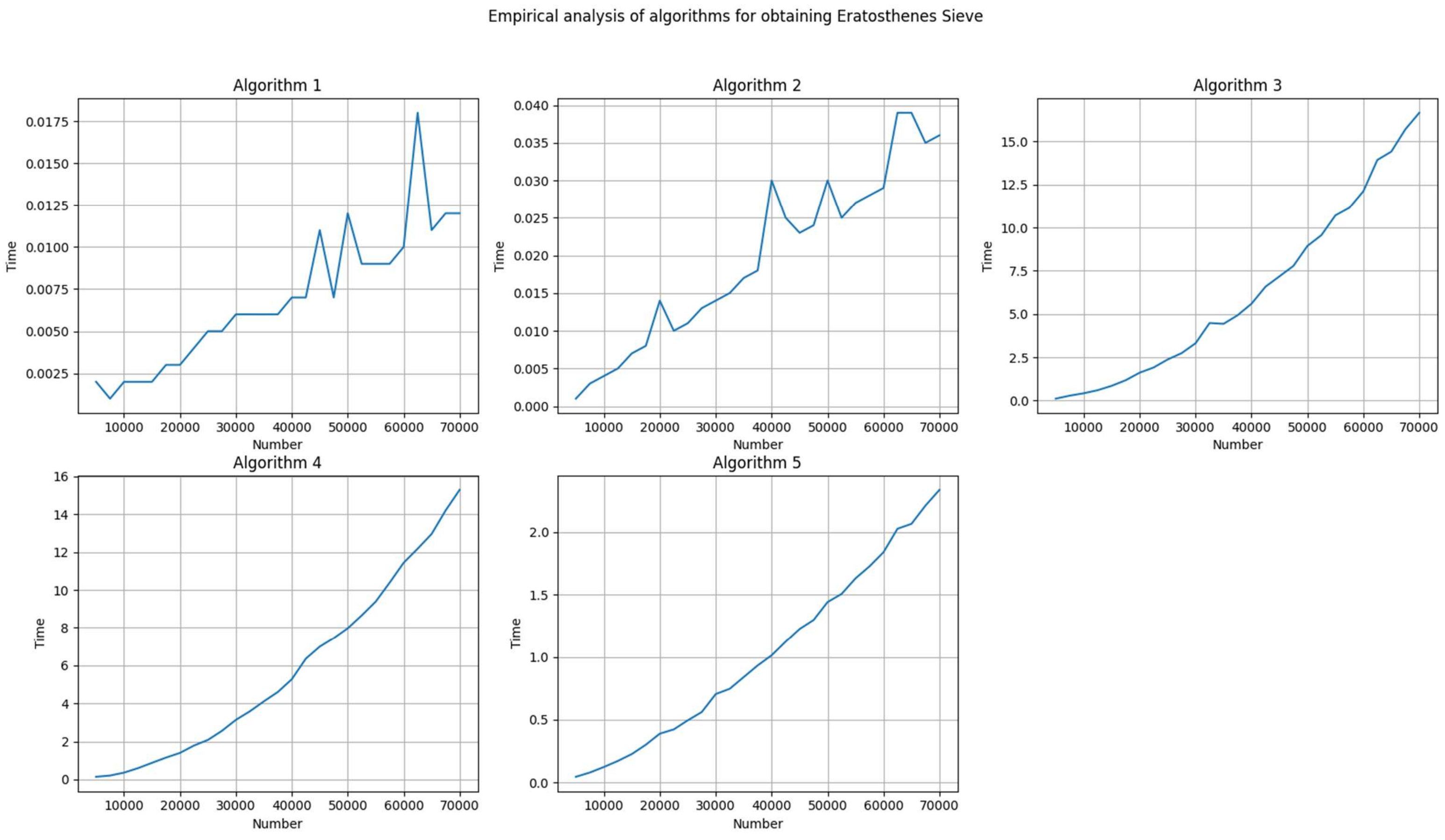
In summary, Algorithm 5 is an improvement on Algorithm 4 by checking divisors up to the square root of i, rather than iterating through all numbers up to i-1. However, it is still less efficient than other prime number generation algorithms, and its use is limited to medium-sized values of n.

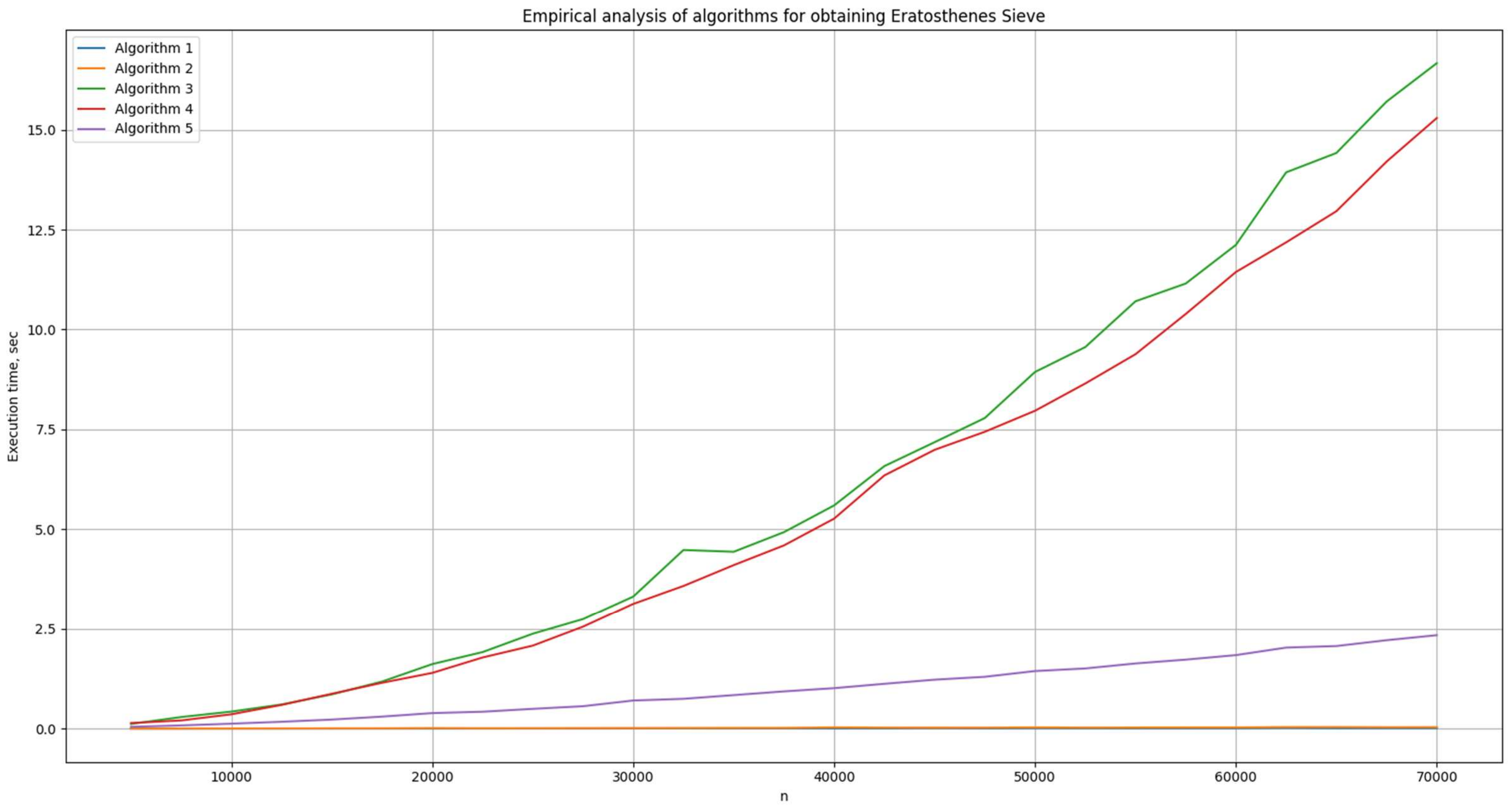
```python
def algorithm_5(n):
    c = [True] * (n + 1)  # Initialize all values of c to True
    c[1] = False  # Set c[1] to False
    i = 2

    while i <= n:
        j = 2
        while j <= i**0.5:
            if i % j == 0:
                c[i] = False
            j = j + 1
        i = i + 1

    return c
```

Results


Empirical analysis of algorithms for obtaining Eratosthenes Sieve
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 5
Time
Number


Empirical analysis of algorithms for obtaining Eratosthenes Sieve
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 5
Execution time, sec
n

CONCLUSION

The Sieve of Eratosthenes is a highly efficient algorithm for generating all prime numbers up to a given integer n. It works by iteratively marking as composite the multiples of each prime, starting with the multiples of 2. The algorithm has a time complexity of O(n log log n) and is suitable for generating prime numbers for large values of n.

Algorithm 1 is a simple algorithm that iteratively checks each number from 2 to n and marks all of its multiples as composite. It has a time complexity of $O(n^2)$ and is less efficient than the Sieve of Eratosthenes.

Algorithm 2 is a modified version of Algorithm 1 that only checks multiples of primes. It has a time complexity of O(n log log n) and is more efficient than Algorithm 1.

Algorithm 3 is another modification of Algorithm 1 that checks all divisors of each number from 2 to n. It has a time complexity of $O(n^2)$ and is less efficient than Algorithm 1 and Algorithm 2.

Algorithm 4 is another modification of Algorithm 1 that checks if each number from 2 to n is prime by iterating through all numbers from 2 to i-1 and checking if any of them divide i evenly. It has a time complexity of $O(n^2)$ and is less efficient than Algorithm 1 and Algorithm 2.

Algorithm 5 is a modification of Algorithm 4 that only checks divisors up to the square root of i. It has a time complexity of $O(n^{1.5})$ and is more efficient than Algorithm 4, but still less efficient than the Sieve of Eratosthenes and Algorithm 2.

In conclusion, the Sieve of Eratosthenes and Algorithm 2 are the most efficient algorithms for generating prime numbers up to a given integer n. Algorithm 1, Algorithm 4, and Algorithm 5 are less efficient and should only be used for small to medium-sized values of n. Algorithm 3 is the least efficient algorithm presented and should be avoided for all but the smallest values of n. The choice of algorithm will depend on the size of n and the desired level of efficiency.

APPENDIX

Git-hub link:
https://github.com/Ricigeroi/AA-Lab-3