Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

REPORT

Laboratory work no. 3
Eratosthenes Sieve

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

Study and analyze different algorithms for finding prime numbers until a limit.

Tasks

1. Implement 5 algorithms for determining prime numbers;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical notes

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction

The sieve of Eratosthenes is an ancient algorithm used in mathematics to determine all prime numbers up to a given limit. This method involves marking the multiples of each prime number as composite or not prime, starting with the number 2, which is the first prime. The multiples of a prime number are calculated as a sequence of numbers beginning from the prime number, with a constant difference equal to that prime. This approach differs from trial division, which sequentially tests each candidate number for divisibility by each prime. Once all multiples of each identified prime have been marked, the remaining unmarked numbers are considered prime.

The earliest reference to this method is found in Nicomachus of Gerasa's Introduction to Arithmetic, which attributes it to the Greek mathematician Eratosthenes of Cyrene from the 3rd century BCE. This sieve is one of the most efficient ways to find smaller primes and can also be used to determine primes in arithmetic progressions.

A prime number is a natural number that divides evenly into exactly two different natural numbers: 1 and itself. The Eratosthenes' approach can be used to find all the prime numbers that are less than or equal to a given integer n.

1. Make a list of 2 through n consecutive integers: (2, 3, 4, ..., n).
2. Let p begin with the value 2, the smallest prime number.
3. Create a list of all the multiples of p and mark each one in it by counting in p increments from 2p to n. (these will be 2p, 3p, 4p, ...; the p itself should not be marked).
4. Find the smallest unmarked number greater than p in the list. Stop if there wasn't such a number. If not, set p to this new number (which is the following prime), then proceed to step 3 as normal.
5. All the prime numbers below n that are still unmarked in the list when the algorithm is finished.

As every value supplied to p must be a prime because if it were composite, it would be labeled as a multiple of a smaller prime, the fundamental principle is that every value given to p must be a prime. Please take note that some of the numerals may have several markings (e.g., 15 will be marked both for 3 and 5).

As an improvement, since all of the smaller multiples of p will have already been marked at that time, it is sufficient to start marking the numbers in step 3 at p2. This indicates that when p2 is bigger than n, the algorithm may end at step 4. Another improvement is to designate just odd multiples of p by first listing only odd numbers (3, 5,..., n), and then counting in increments of 2p in step 3. Actually, the original algorithm included this. Wheel factorization can be used to generalize this, producing the first list exclusively from numbers coprime with the first few primes rather than only from odds (i.e., numbers coprime with 2), and counting in the appropriately modified increments to ensure that only those multiples of p are initially generated that are coprime with those small primes.

5 variations of Eratosthenes Sieve algorithm are:

1. **Algorithm 1**
2. **Algorithm 2**
3. **Algorithm 3**
4. **Algorithm 4**
5. **Algorithm 5**

Comparison Metric

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

Input Format

As input, each variation of the algorithm will receive multiple lists with different number of elements, their length being:

- 1000
- 2000
- 3000
- 4000
- 5000
- 6000
- 7000
- 8000
- 9000
- 10000

IMPLEMENTATION

All variations of the algorithm will be put into practice in Python and objectively evaluated depending on how long it takes to complete each one. The particular efficiency in report with input will vary based on the memory of the device utilized, even though the overall trend of the results may be similar to other experimental data. As determined by experimental measurement, the error margin will be a few seconds.

Algorithm 1

Algorithm Description:

This algorithm generates a list of boolean values, where the i-th element in the list represents whether the integer i is a prime number or not. It does so using the Sieve of Eratosthenes algorithm, which is a well-known method for finding all prime numbers up to a given limit.

Implementation

```
def algorithm_1(n):
    c = [True] * (n+1)
    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
    return c
```

Results

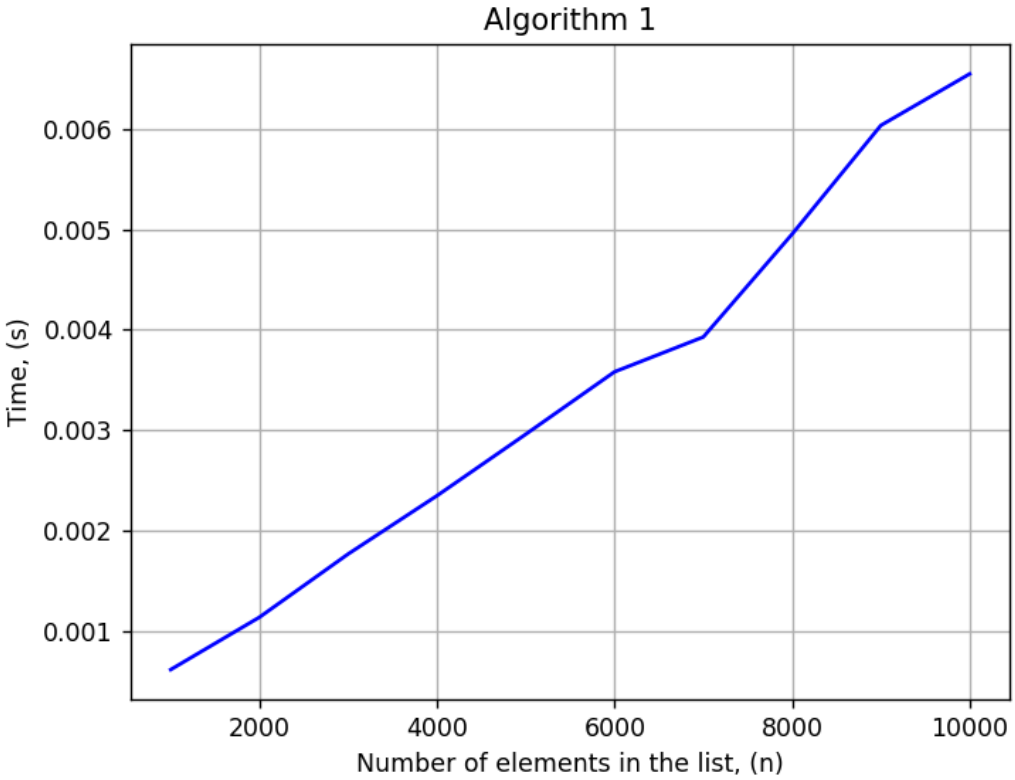
Algorithm / n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Algorithm 1	0.00060	0.00116	0.00183	0.00259	0.00286	0.00396	0.00438	0.00474	0.00524	0.00649
Algorithm 2	0.00119	0.00249	0.00406	0.00534	0.00681	0.00779	0.01063	0.01075	0.01344	0.01547
Algorithm 3	0.01800	0.06674	0.14119	0.24547	0.37797	0.52623	0.71232	0.90617	1.14919	1.54422
Algorithm 4	0.09356	0.40091	0.93287	1.68939	2.65205	3.85972	5.33491	7.18957	9.01292	11.27274
Algorithm 5	0.01005	0.02911	0.05189	0.08560	0.11650	0.15491	0.18794	0.22929	0.26897	0.32413

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm is T(n log(log n)). The outer loop iterates n times, and for each prime number i, the inner loop marks all multiples of i up to n. This inner loop iterates at most n/i times, so the total number of iterations of the inner loop is:

$2 + 3 + 4 + ... + n = n(n+1)/2 - 1 = T(n^2)$

However, each composite number less than n is marked only once, so the total number of operations is $T(n \cdot \log(\log n))$, which is the complexity of the Sieve of Eratosthenes algorithm.

Graph



Algorithm 2

Algorithm Description:

This algorithm initializes an array **c** of Boolean values, where each value initially represents a potential prime number. It then iterates over each integer **i** from 2 to **n**, and for each **i**, it marks all of its multiples as non-prime by setting the corresponding values in the array **c** to **False**.

The inner loop starts with **j = 2*i** and increments **j** by **i** until **j** reaches **n**. Since the loop starts at **2*i** instead of **i**, it skips marking **i** itself as non-prime. However, this means that all composite numbers will be marked multiple times, resulting in a less efficient algorithm.

Implementation

```
def algorithm_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 c
```

Results

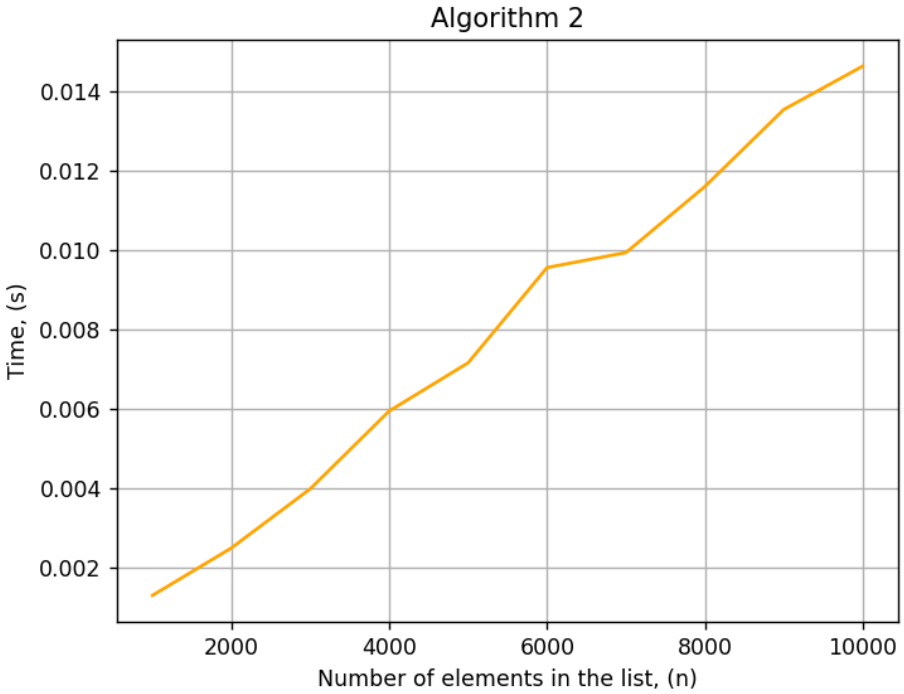
Algorithm / n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Algorithm 1	0.00060	0.00116	0.00183	0.00259	0.00286	0.00396	0.00438	0.00474	0.00524	0.00649
Algorithm 2	0.00119	0.00249	0.00406	0.00534	0.00681	0.00779	0.01063	0.01075	0.01344	0.01547
Algorithm 3	0.01800	0.06674	0.14119	0.24547	0.37797	0.52623	0.71232	0.90617	1.14919	1.54422
Algorithm 4	0.09356	0.40091	0.93287	1.68939	2.65205	3.85972	5.33491	7.18957	9.01292	11.27274
Algorithm 5	0.01005	0.02911	0.05189	0.08560	0.11650	0.15491	0.18794	0.22929	0.26897	0.32413

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm can be analyzed as follows:

- The outer while loop runs n-1 times (from i = 2 to i = n), so it has a time complexity of T(n).
- The inner while loop runs n/i times for each value of i, where i ranges from 2 to n. Therefore, the total number of iterations of the inner loop is roughly:
  n/2 + n/3 + n/4 + ... + n/n
  This is known as the harmonic series and it's bounded by log(n), so we can say that the inner loop runs T (n log n) times in total.
- The other operations in the algorithm (array initialization and updating) are constant time operations that don't depend on n.

Combining these results, we get a time complexity of T (n log n) for the Sieve of Eratosthenes algorithm.

Graph



Algorithm 3

Algorithm Description:

First, it initializes a list **c** of boolean values of length **n+1**, where **c[i]** represents whether **i** is a prime number or not. All values in the list are set to **True**, except for **c[1]** which is set to **False**.

Then, the algorithm iterates through the list starting from 2, the first prime number. For each prime number **i**, it sets all multiples of **i** to **False** in the **c** list. This is done by iterating through the list again, starting from **i+1**, and checking whether each number is a multiple of **i**. If it is, the corresponding value in **c**

is set to **False**.

Finally, the algorithm returns the **c** list, which contains **True** values for all prime numbers up to **n**, and **False** values for all composite numbers.

Implementation

```python
def algorithm_3(n):
    c = [True] * (n+1)
    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
    return c
```

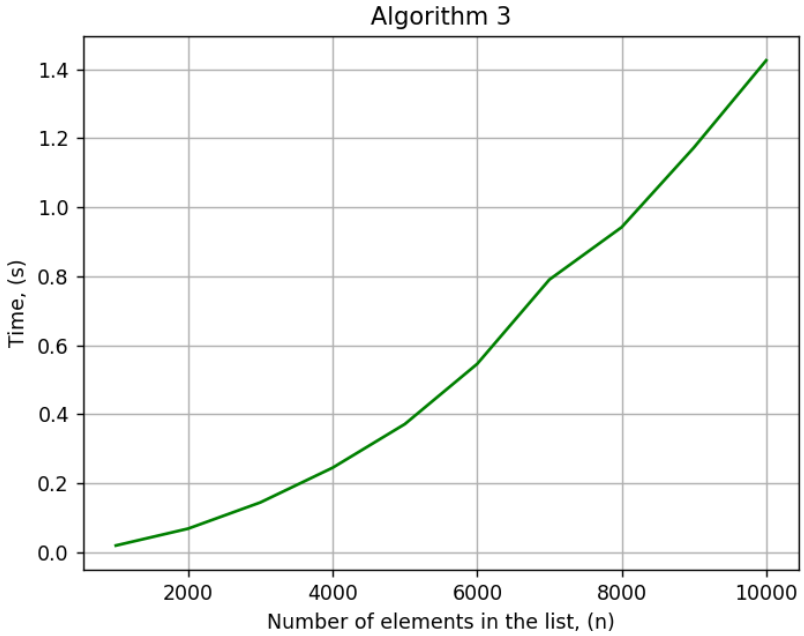
Results

Algorithm / n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Algorithm 1	0.00060	0.00116	0.00183	0.00259	0.00286	0.00396	0.00438	0.00474	0.00524	0.00649
Algorithm 2	0.00119	0.00249	0.00406	0.00534	0.00681	0.00779	0.01063	0.01075	0.01344	0.01547
Algorithm 3	0.01800	0.06674	0.14119	0.24547	0.37797	0.52623	0.71232	0.90617	1.14919	1.54422
Algorithm 4	0.09356	0.40091	0.93287	1.68939	2.65205	3.85972	5.33491	7.18957	9.01292	11.27274
Algorithm 5	0.01005	0.02911	0.05189	0.08560	0.11650	0.15491	0.18794	0.22929	0.26897	0.32413

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm can be analyzed as follows:

- The outer while loop iterates **n-1** times (from 2 to n) which takes T(n) time.
- The inner while loop iterates **(n-1) + (n-2) + ... + 2 + 1** times in total, which is equal to **(n-1)*n/2 - 1** times, so it takes $T(n^2)$ time.
- All other operations in the algorithm take constant time.
- Therefore, the overall time complexity of the algorithm is $T(n^2)$.

Graph



Algorithm 4

Algorithm Description:

The algorithm takes an integer n as input and initializes a list c with n+1 elements, all set to True except for the first element (c[1]=False). It then loops through integers from 2 to n, and for each integer i, loops through integers from 1 to i-1, checking if i is divisible by j. If i is divisible by j, then c[i] is set to False. The algorithm returns the list c, which contains True or False values indicating whether each number from 1 to n is prime or composite.

Implementation

```
def algorithm_4(n):
    c = [True] * (n+1)
    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
    return c
```

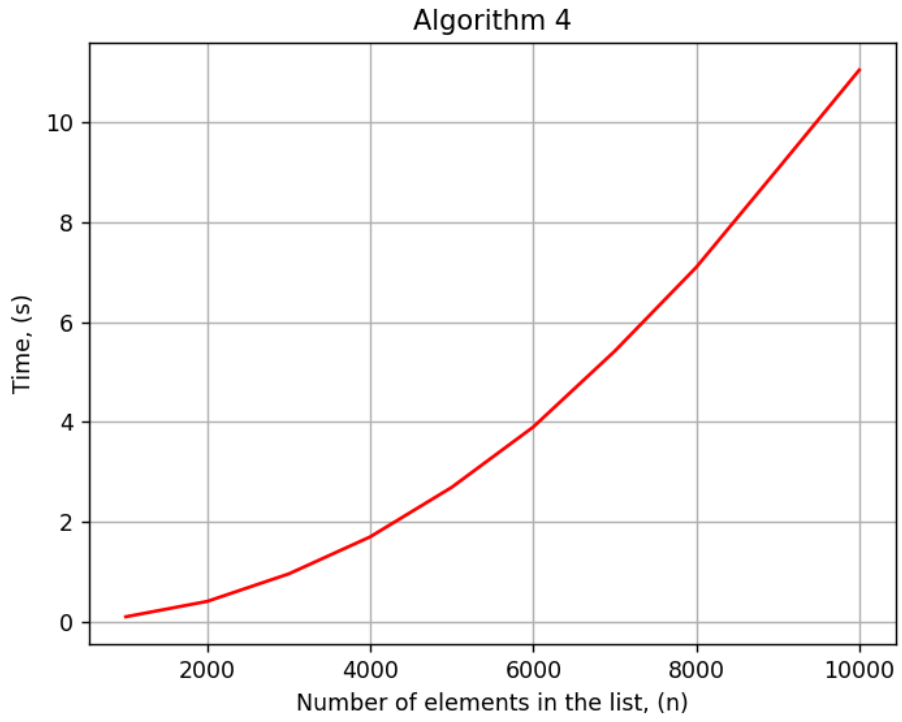
Results

Algorithm / n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Algorithm 1	0.00060	0.00116	0.00183	0.00259	0.00286	0.00396	0.00438	0.00474	0.00524	0.00649
Algorithm 2	0.00119	0.00249	0.00406	0.00534	0.00681	0.00779	0.01063	0.01075	0.01344	0.01547
Algorithm 3	0.01800	0.06674	0.14119	0.24547	0.37797	0.52623	0.71232	0.90617	1.14919	1.54422
Algorithm 4	0.09356	0.40091	0.93287	1.68939	2.65205	3.85972	5.33491	7.18957	9.01292	11.27274
Algorithm 5	0.01005	0.02911	0.05189	0.08560	0.11650	0.15491	0.18794	0.22929	0.26897	0.32413

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm can be analyzed as follows:

- The outer while loop runs n-1 times.
- The inner while loop for each i runs i-1 times.
- The time taken for the inner while loop is T(i).
- Therefore, the total time taken by the algorithm is: $T(2 + 3 + 4 + ... + n) + T(1) = T(n^2)$
- This is because the sum of 2 to n is (n+2)(n-1)/2, which is $T(n^2)$.

Graph



Algorithm 5

Algorithm Description:

The algorithm takes an integer n as input and initializes a list c with n+1 elements, all set to True except for the first element (c[1]=False). It then loops through integers from 2 to n, and for each integer i, loops through integers from 2 to the square root of i, checking if i is divisible by any of those integers. If i is divisible by any of those integers, then c[i] is set to False. The algorithm returns the list c, which contains True or False values indicating whether each number from 1 to n is prime or composite.

Implementation

```
def algorithm_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 c
```

Results

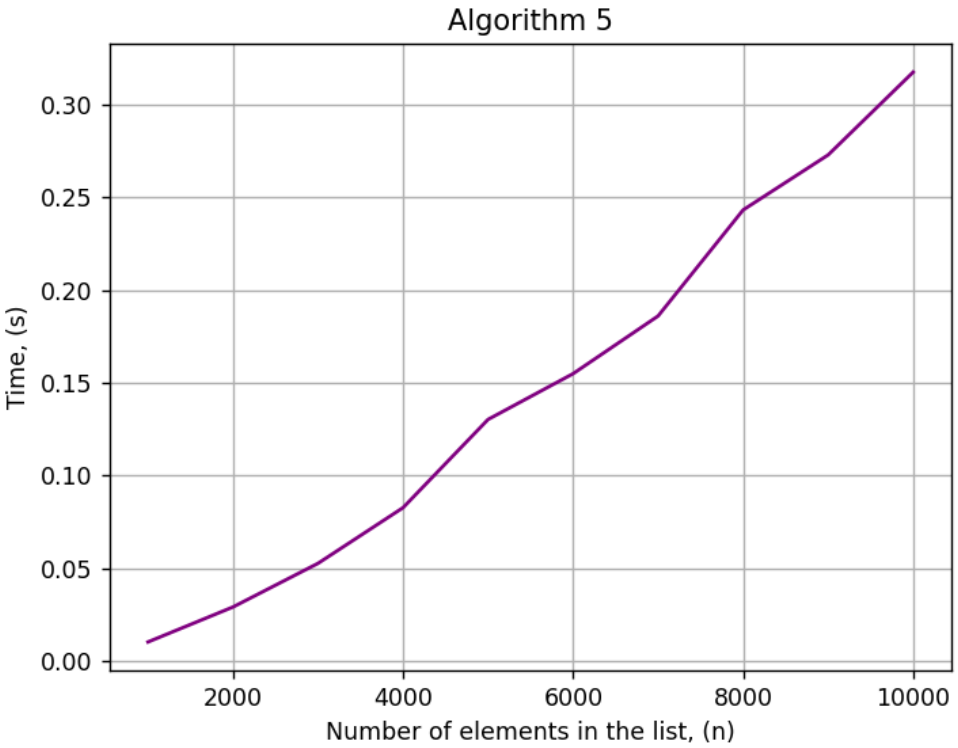
Algorithm / n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Algorithm 1	0.00060	0.00116	0.00183	0.00259	0.00286	0.00396	0.00438	0.00474	0.00524	0.00649
Algorithm 2	0.00119	0.00249	0.00406	0.00534	0.00681	0.00779	0.01063	0.01075	0.01344	0.01547
Algorithm 3	0.01800	0.06674	0.14119	0.24547	0.37797	0.52623	0.71232	0.90617	1.14919	1.54422
Algorithm 4	0.09356	0.40091	0.93287	1.68939	2.65205	3.85972	5.33491	7.18957	9.01292	11.27274
Algorithm 5	0.01005	0.02911	0.05189	0.08560	0.11650	0.15491	0.18794	0.22929	0.26897	0.32413

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm can be analyzed as follows:

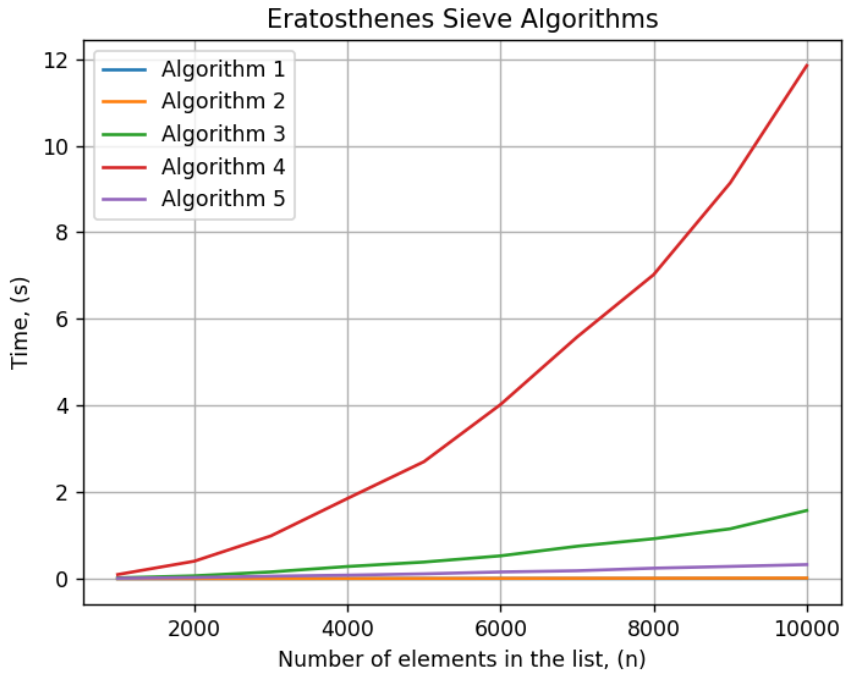
- The outer while loop runs n-1 times.
- The inner while loop for each i runs $\sqrt{i}$-1 times.
- The time taken for the inner while loop is T ($\sqrt{i}$).
- Therefore, the total time taken by the algorithm is: T ($\sqrt{2}$+ $\sqrt{3}$+ $\sqrt{4}$ + ... + $\sqrt{n}$) + T (1) = T(n · $\sqrt{n}$)
- This is because the sum of the square roots of 2 to n is bounded by the integral of $\sqrt{x}$ with respect to x from 2 to n, which is approximately 2/3 · n · $\sqrt{n}$.

Therefore, the time complexity of the algorithm is O(n · $\sqrt{n}$). This is a faster algorithm for prime number identification than Algorithm 4, but still slower than some other algorithms like Algorithm 1 and 2.

Graph



Comparison



Rankings in terms of performance of Sieve of Eratosthenes algorithms based on the graph are as follows:

1. Algorithm 1
2. Algorithm 2
3. Algorithm 5
4. Algorithm 3
5. Algorithm 4

CONCLUSION

From the conducted empirical analysis, it can be concluded that Algorithm 1 (Sieve of Eratosthenes) is the fastest algorithm for finding prime numbers among the given five algorithms. This is followed closely by Algorithm 2, which is a simplified version of Algorithm 1 that eliminates redundant checks for already marked composite numbers.

Algorithm 5, which checks each number from 2 to the square root of n, performed relatively well and came in third place in terms of execution time. On the other hand, Algorithm 3, which eliminates composite numbers by marking their multiples, and Algorithm 4, which checks each number from 2 to n-1 to see if it is a factor of n, were the slowest algorithms.

The reason for Algorithm 1's superior performance can be attributed to its efficient marking of composite numbers using a Boolean array. Algorithm 2, which is a simplified version of Algorithm 1, also benefits from this approach.

Overall, the choice of algorithm for finding prime numbers would depend on the specific requirements of the application, such as the size of the input and the available memory for processing. If memory is not an issue, Algorithm 1 or Algorithm 2 would be the best choices for finding prime numbers efficiently. However, if memory is limited, Algorithm 5 would be a better choice, even though it may be slightly slower than Algorithm 1 and Algorithm 2.

GITHUB

Sufferal/Algorithms: Analysis of algorithms (github.com)