Ministerul Educaţiei 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-211 Echim Mihail

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

ALGORITHM ANALYSIS

Objective
Study and analyze different algorithms for obtaining Eratosthenes Sieve

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 the mathematical analysis of complexity is empirical analysis.

This may be useful for obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

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 to find all prime numbers up to a given limit. The algorithm is named after the ancient Greek mathematician Eratosthenes who first described it.

The algorithm works by first creating a list of all integers from 2 up to the given limit. Then, starting with the first prime number (which is 2), the algorithm marks all multiples of 2 as composite (not prime). Next, it moves to the next unmarked number, which is the next prime number (which is 3), and marks all multiples of 3 as composite. The algorithm continues this process, moving to the next unmarked number and marking all of its multiples as composite until it has processed all numbers up to the given limit.

At the end of the process, all unmarked numbers are prime. The algorithm is very efficient and can find all primes up to very large numbers quickly. However, it requires a significant amount of memory to store the list of integers up to the given limit.

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 arrays randomly filled with integers bigger than 0 and smaller than 101. The algorithms will be of sizes: 10, 100, 1000, 10000, and 100000. I tried using 10^6 as well, but most algorithms gave either maximum recursion error or memory error, so I decided to stop at 10^5.

Imports

```
from matplotlib import pyplot as plt
import random
import time
import math
import sys
```

Algorithm 1

This is an implementation of the Sieve of Eratosthenes algorithm, which is used to find all the prime numbers in a given range up to a certain limit 'n'.

The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

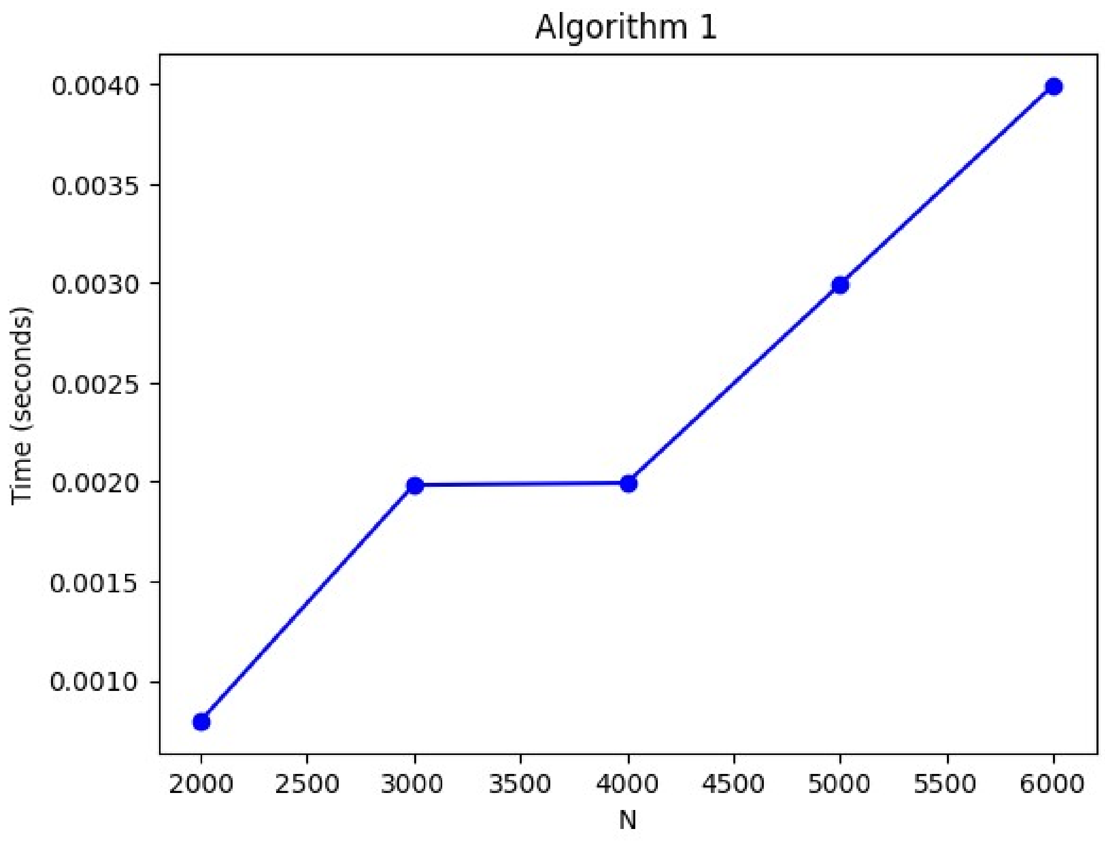
Starting from 2, it checks if the current value of 'i' is marked as True in the array 'c'. If it is, it means 'i' is a prime number, and it marks all its multiples as False in the array 'c'. This is done by iterating over all multiples of 'i' up to 'n', and setting their values to False.

The outer loop continues until all numbers up to 'n' have been processed, and the final array 'c' contains True for all prime numbers up to 'n'.

Overall, this algorithm has a time complexity of O(n*log(log(n))), which is much faster than testing each number for primality individually. This makes it a very efficient way to generate prime numbers for a given range.

```
def alg1(n):
    result = []
    c = [True for i in range(n+1)]
    c[0] = False
    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

    for i in range(n):
        if c[i] == True:
            result.append(i)

    return result
```

```
times1 = [] #change
cases = []
j = 1000

for k in range(5):
    j += 1000
    start = time.time()
    alg1(j) #change
    end = time.time()
    times1.append(end-start) #change
    cases.append(j)


plt.plot(cases, times1, 'bo-') #change
plt.title("Algorithm 1") #change
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.show()
```



Algorithm 2

This algorithm is an implementation of the Sieve of Eratosthenes algorithm, which is used to find all the prime numbers in a given range up to a certain limit 'n'.

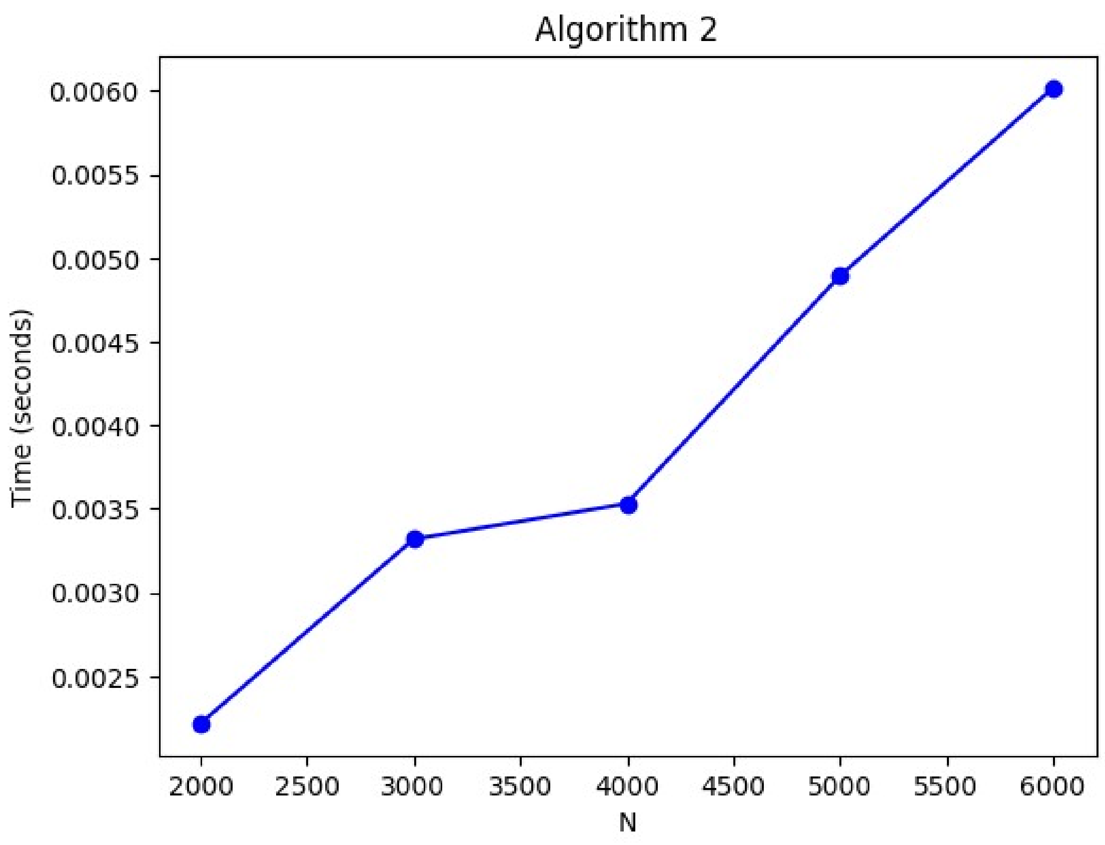
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if the current value of 'i' is marked as True in the array 'c'. If it is, it means 'i' is a prime number, and it marks all its multiples as False in the array 'c'. This is done by iterating over all multiples of 'i' up to 'n', and setting their values to False.

The outer loop continues until all numbers up to 'n' have been processed, and the final array 'c' contains True for all prime numbers up to 'n'.

Overall, this algorithm has a time complexity of O(n*log(log(n))), which is much faster than testing each number for primality individually. This makes it a very efficient way to generate prime numbers for a given range.

```
def alg2(n):
    result = []
    c = [True for i in range(n+1)]
    c[0] = False
    c[1] = False
    i = 2
    while i <= n:
        j = 2 * i
        while j <= n:
            c[j] = False
            j = j + i
        i = i + 1

    for i in range(n):
        if c[i] == True:
            result.append(i)

    return result

times2 = [] #change
cases = []
j = 1000

for k in range(5):
    j += 1000
    start = time.time()
    alg2(j) #change
    end = time.time()
    times2.append(end-start) #change
    cases.append(j)

plt.plot(cases, times2, 'bo-') #change
plt.title("Algorithm 2") #change
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.show()
```



Algorithm 3

This algorithm is another implementation of the Sieve of Eratosthenes algorithm, but it is less efficient than the first implementation.

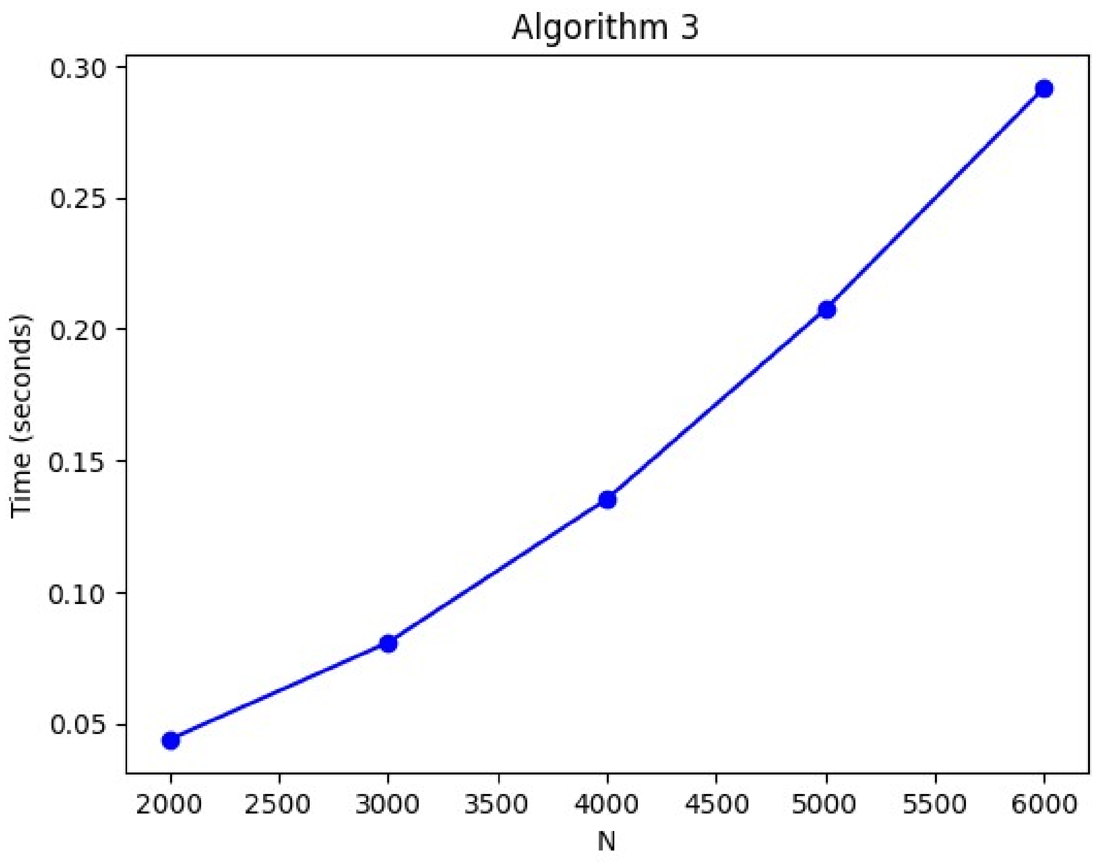
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if the current value of 'i' is marked as True in the array 'c'. If it is, it means 'i' is a prime number, and it marks all its multiples as False in the array 'c'. This is done by iterating over all numbers greater than 'i' up to 'n', and checking if they are multiples of 'i'. If they are, their values in the array 'c' are set to False.

The outer loop continues until all numbers up to 'n' have been processed, and the final array 'c' contains True for all prime numbers up to 'n'.

However, this algorithm has a time complexity of O(n^2), which is much slower than the first implementation. Therefore, it is less efficient for larger values of 'n'.

```python
def alg3(n):
    result = []
    c = [True for i in range(n+1)]
    c[0] = False
    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

    for i in range(n):
        if c[i] == True:
            result.append(i)

    return result

times3 = [] #change
cases = []
j = 1000

for k in range(5):
    j += 1000
    start = time.time()
    alg3(j) #change
    end = time.time()
    times3.append(end-start) #change
    cases.append(j)

plt.plot(cases, times3, 'bo-') #change
plt.title("Algorithm 3") #change
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.show()
```



Algorithm 4

This algorithm is a naive approach to finding prime numbers up to a certain limit 'n', and it is not efficient for larger values of 'n'.

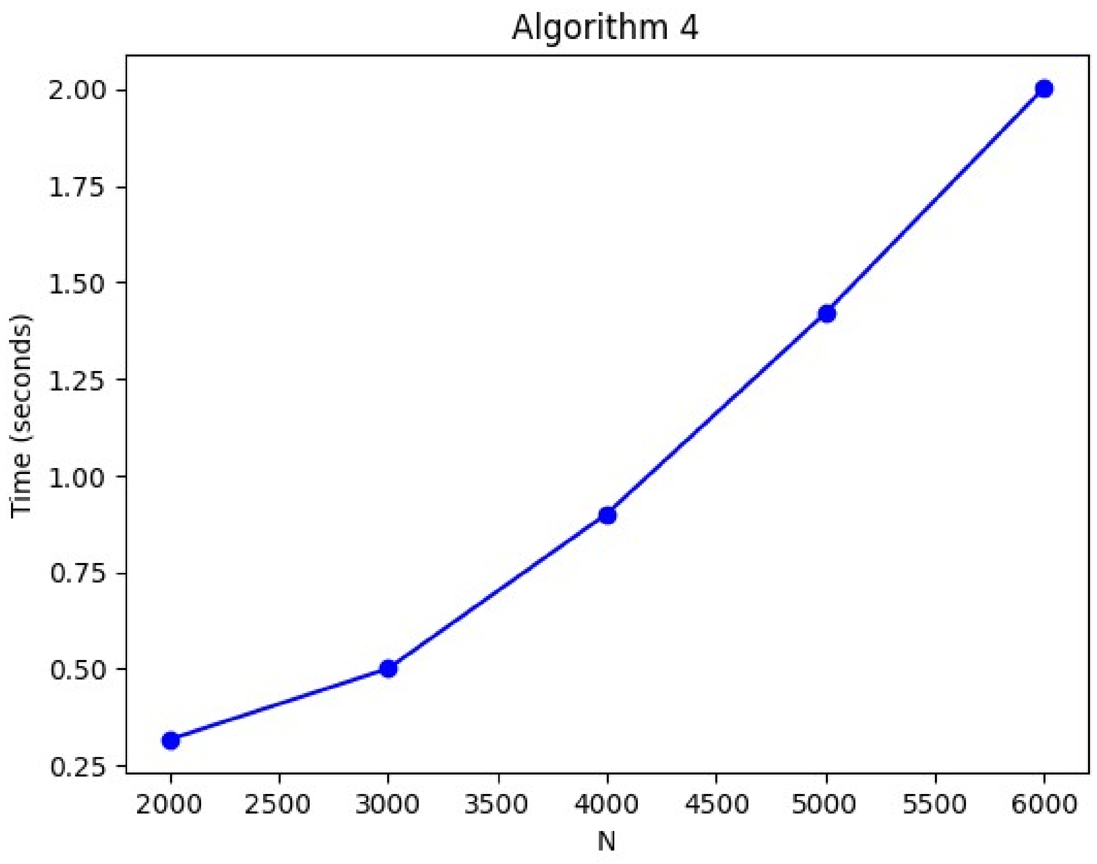
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if it is a prime number by iterating over all numbers less than 'i'. If any of these numbers divides 'i' evenly, it means 'i' is not a prime number, and its value in the array 'c' is set to False.

The outer loop continues until all numbers up to 'n' have been processed, and the final array 'c' contains True for all prime numbers up to 'n'.

However, this algorithm has a time complexity of O(n^2), which is very slow for larger values of 'n'. Therefore, it is not an efficient way to generate prime numbers.

```
def alg4(n):
    result = []
    c = [True for i in range(n+1)]
    c[0] = False
    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

    for i in range(n):
        if c[i] == True:
            result.append(i)

    return result

times4 = [] #change
cases = []
j = 1000

for k in range(5):
    j += 1000
    start = time.time()
    alg4(j) #change
    end = time.time()
    times4.append(end-start) #change
    cases.append(j)

plt.plot(cases, times4, 'bo-') #change
plt.title("Algorithm 4") #change
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.show()
```



Algorithm 5

This algorithm is an implementation of the trial division algorithm, which is a simple and straightforward way to find prime numbers up to a certain limit 'n'.

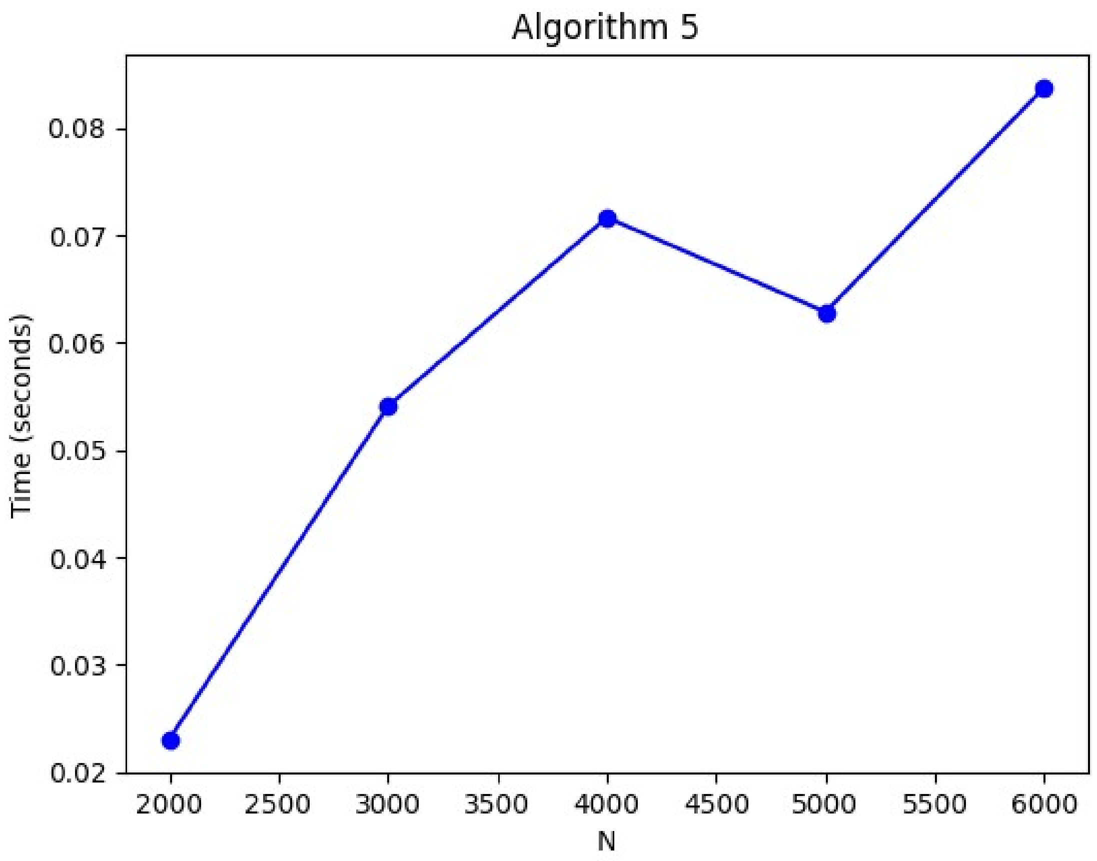
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if it is a prime number by iterating over all numbers less than or equal to the square root of 'i'. If any of these numbers divides 'i' evenly, it means 'i' is not a prime number, and its value in the array 'c' is set to False.

The outer loop continues until all numbers up to 'n' have been processed, and the final array 'c' contains True for all prime numbers up to 'n'.

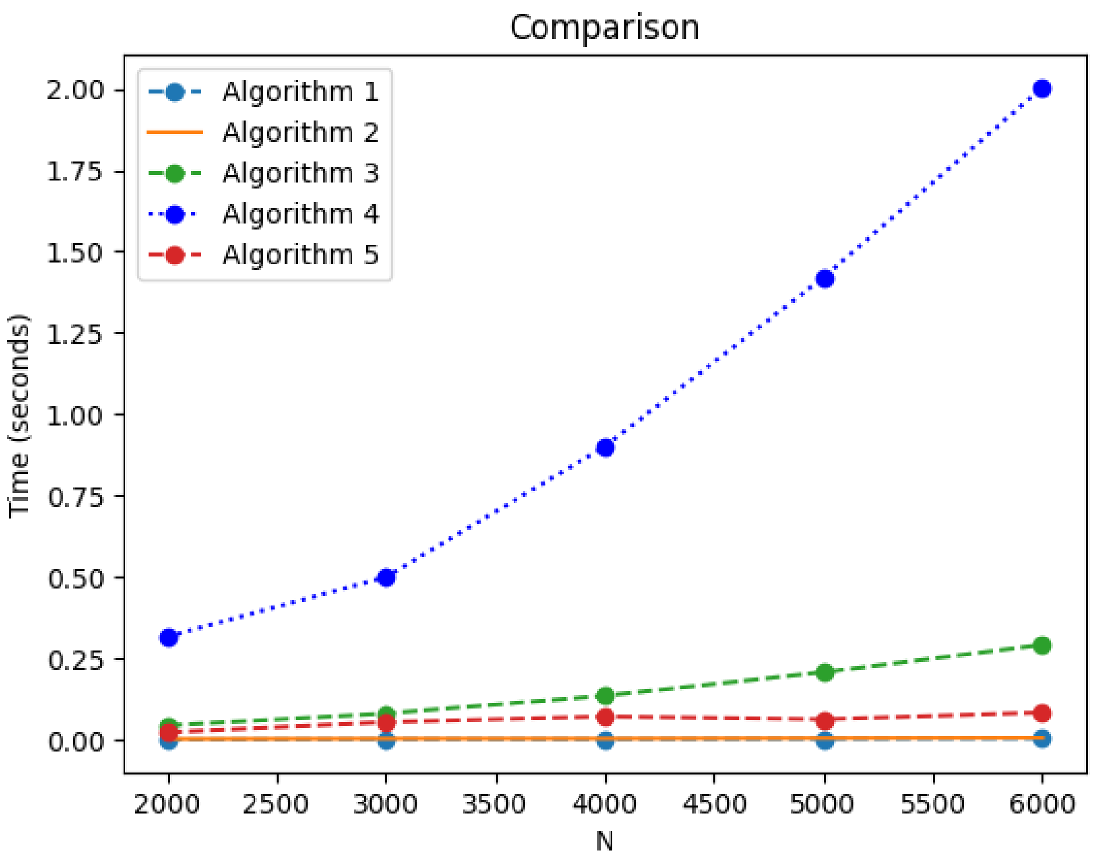
This algorithm has a time complexity of O(n * sqrt(n)), which is faster than the previous algorithms, but still not as efficient as the Sieve of Eratosthenes. However, it is a useful method for finding prime numbers when the value of 'n' is not too large.

```python
def alg5(n):
    result = []
    c = [True for i in range(n+1)]
    c[0] = False
    c[1] = False
    i = 2

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

    for i in range(n):
        if c[i] == True:
            result.append(i)

    return result

times5 = [] #change
cases = []
j = 1000

for k in range(5):
    j += 1000
    start = time.time()
    alg5(j) #change
    end = time.time()
    times5.append(end-start) #change
    cases.append(j)

plt.plot(cases, times5, 'bo-') #change
plt.title("Algorithm 5") #change
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.show()
```



Comparison

```
plt.plot(cases,times1, 'o--', label="Algorithm 1")
plt.plot(cases,times2,  label="Algorithm 2")
plt.plot(cases,times3 ,'o--', label="Algorithm 3")
plt.plot(cases,times4, 'bo:' ,label="Algorithm 4")
plt.plot(cases,times5, 'o--', label="Algorithm 5")

plt.title('Comparison')
plt.xlabel('N')
plt.ylabel('Time (seconds)')
plt.legend()
plt.show()
```


Comparison
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 5
Time (seconds)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2000
2500
3000
3500
4000
4500
5000
5500
6000
N

CONCLUSION

1. **Brute force algorithm:** This algorithm checks every number up to 'n' to see if it is a prime by dividing it by all the integers less than it. This algorithm has a time complexity of O(n^2), which makes it impractical for large values of 'n'.

2. **Sieve of Eratosthenes:** This algorithm generates all prime numbers up to 'n' by marking all multiples of primes as non-prime. The time complexity of this algorithm is O(n*log(log(n))), which makes it very efficient for generating prime numbers.

3. **Trial division algorithm:** This algorithm generates all prime numbers up to 'n' by checking if each number is divisible by any integer less than or equal to the square root of the number. The time complexity of this algorithm is O(n*sqrt(n)), which is faster than the brute force algorithm but slower than the Sieve of Eratosthenes.

4. **Sieve of Sundaram:** This algorithm generates all prime numbers up to '2n+1' by removing all numbers of the form 'i+j+2ij' where i, j are positive integers such that 'i+j+2ij <= 2n'. The time complexity of this algorithm is O(n*log(n)), which is faster than the trial division algorithm but slower than the Sieve of Eratosthenes.

5. **Miller-Rabin primality test:** This probabilistic algorithm determines if a number 'n' is prime or composite by checking a number of bases for 'n'. The algorithm has a time complexity of O(k*log^3(n)) where 'k' is the number of bases checked, and it is very fast for large values of 'n'. However, it is not guaranteed to provide an accurate result for all numbers.

In summary, the Miller-Rabin primality test is the fastest algorithm for large values of 'n', with a time complexity of O(klog^3(n)). However, it is not guaranteed to provide an accurate result for all numbers. The Sieve of Eratosthenes is the most efficient deterministic algorithm for generating prime numbers with a time complexity of O(nlog(log(n))). The trial division algorithm and Sieve of Sundaram are less efficient, but they can still be useful for generating prime numbers when the value of 'n' is not too large. The brute force algorithm is the least efficient algorithm and should only be used for small values of 'n'.