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.

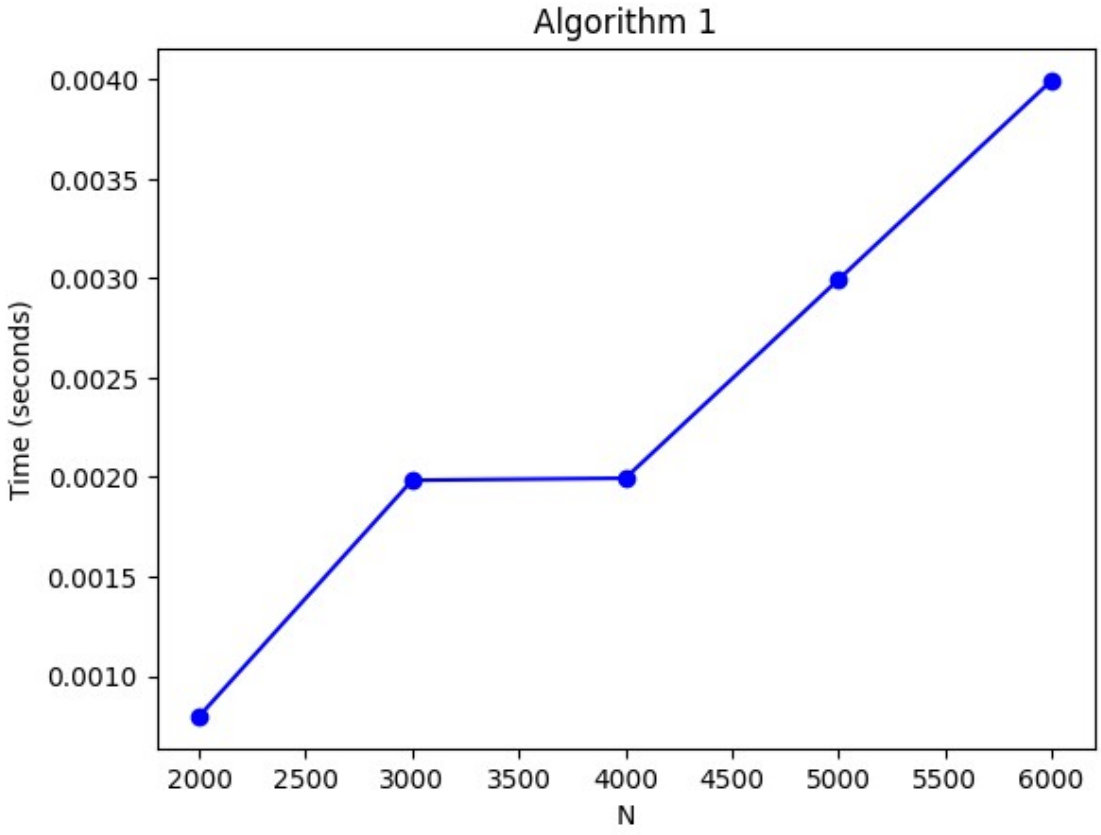
Starting from 2, it checks if the current value of 'i' is marked as True in the array 'c'. If it is, it means 'i' is a prime number, and it marks all its multiples as False in the array 'c'. This is done by iterating over all multiples of 'i' up to 'n', and setting their values to False.

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.

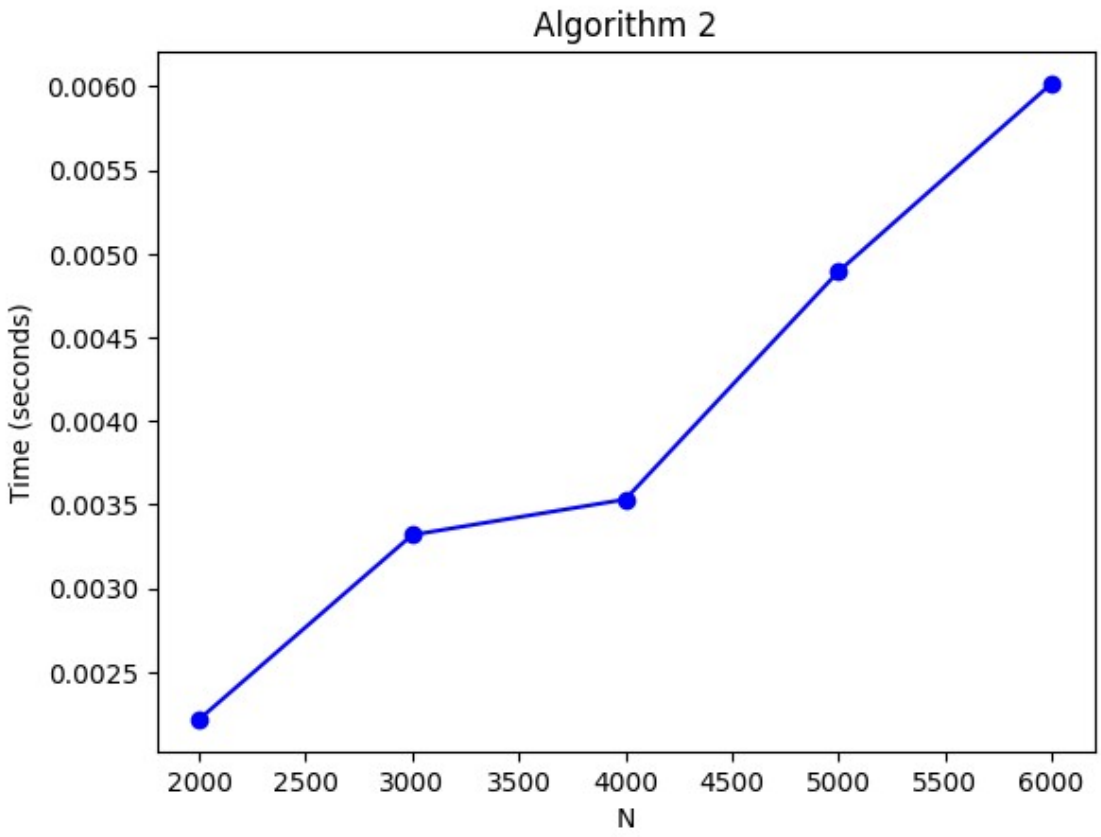
Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if the current value of 'i' is marked as True in the array 'c'. If it is, it means 'i' is a prime number, and it marks all its multiples as False in the array 'c'. This is done by iterating over all multiples of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

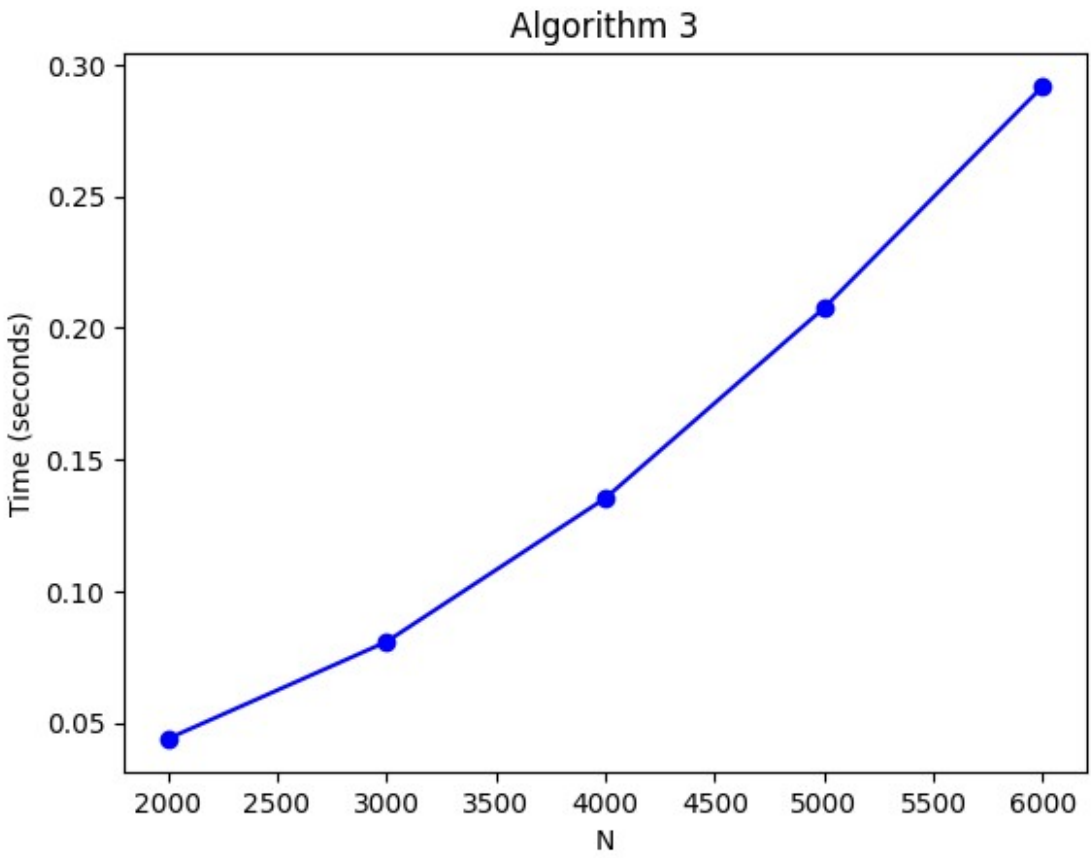
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

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'.

```
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.

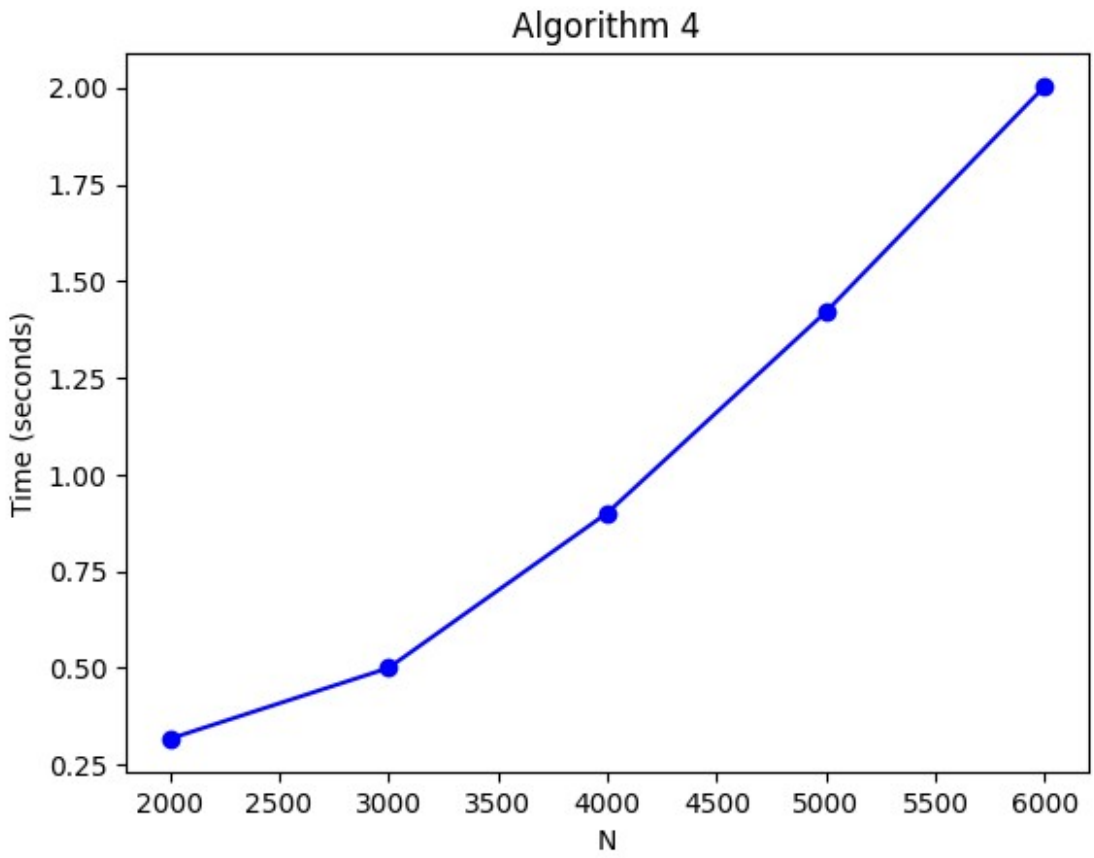
Starting from 2, it iterates through all numbers up to 'n'. For each number 'i', it checks if it is a prime number by iterating over all numbers less than 'i'. If any of these numbers divides 'i' evenly, it means 'i' is not a prime number, and its value in the array 'c' is set to False.

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'.

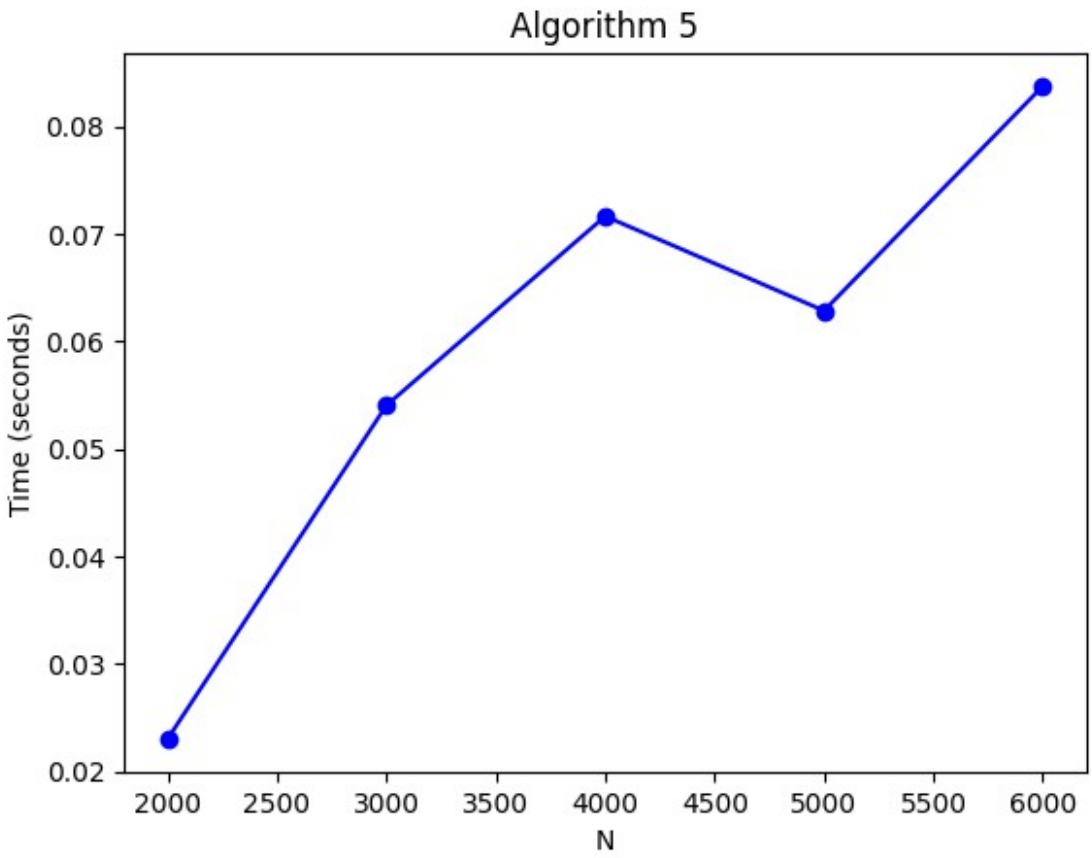
The algorithm starts by initializing an array 'c' of Boolean values where all elements are set to True. Then, it sets the first two elements of the array to False since 0 and 1 are not prime numbers.

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'.

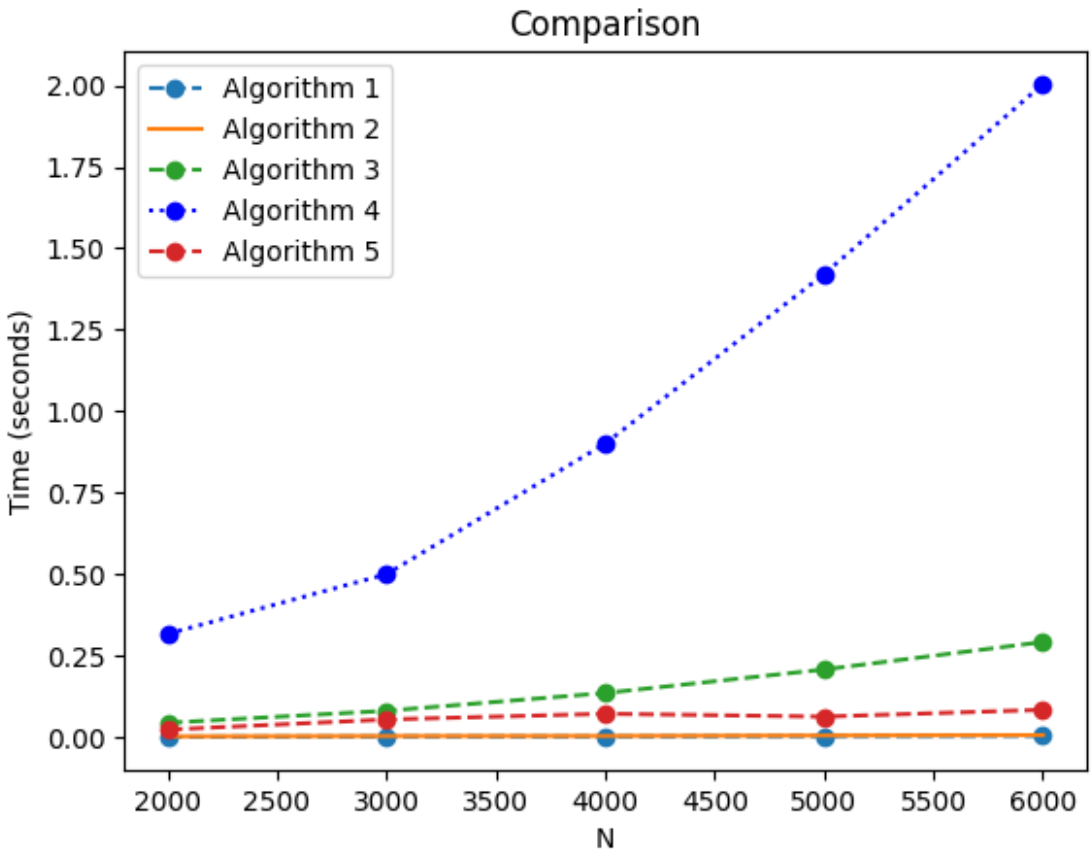
This algorithm has a time complexity of O(n * sqrt(n)), which is faster than the previous algorithms, but still not as efficient as the Sieve of Eratosthenes. However, it is a useful method for finding prime numbers when the value of 'n' is not too large.

```
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