Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department: **Software Engineering and Automation**

CC - Laboratory Work Nr 3

Sieve of Eratosthenes

Ciuş Iurie FAF-203

Chişinău 2022

Contents

1 Algorithm Analysis **3**

1.1 Introduction 3

1.2 Objectives 3

1.3 Theoretical Notes 4

1.4 Algorithms 6

1.4.1 Algorithm 1 6

1.4.2 Algorithm 2 6

1.4.3 Algorithm 3 6

1.4.4 Algorithm 4 6

1.4.5 Algorithm 5 7

2 Code **8**

2.1 Implementation 8

2.2 Algorithmic complexity 11

2.3 Graph / Results 11

3 Conclusion **12**

1 Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

1.1 Introduction

The current paper work represents my implementation of five Sieve of Eratosthenes algorithms.

The Sieve of Eratosthenes is a method for finding all primes up to (and possibly including) a given natural. This method works well when is relatively small, allowing us to determine whether any natural number less than or equal to is prime or composite.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime. Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are primes.

1.2 Objectives

- Study and empirical analysis of sorting algorithms. QuickSort, mergeSort, heapSort analysis, (one of your choice).
- Establish the properties of the input data in relation to which the analysis is made.
- Choose the metric for comparing algorithms.
- Perform empirical analysis of the proposed algorithms.
- Make a conclusion on the work done.

1.3 Theoretical Notes

A prime number is a natural number that has exactly two distinct natural number divisors: the number 1 and itself.

To find all the prime numbers less than or equal to a given integer n by *Eratosthenes' method*:

- Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
- Initially, let p equal 2, the smallest prime number.
- Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself should not be marked).
- Find the smallest number in the list greater than p that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
- When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

The main idea here is that every value given to p will be prime, because if it were composite it would be marked as a multiple of some other, smaller prime. Note that some of the numbers may be marked more than once (e.g., 15 will be marked both for 3 and 5).

As a refinement, it is sufficient to mark the numbers in step 3 starting from p^2, as all the smaller multiples of p will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4 when p2 is greater than n

Another refinement is to initially list odd numbers only, *(3, 5, ..., n)*, and count in increments of $2p$ from p^2 in step 3, thus marking only odd multiples of p. This actually appears in the original algorithm. This can be generalized with wheel factorization, forming the initial list only from numbers coprime with the first few primes and not just from odds (i.e., numbers coprime with 2), and counting in the correspondingly adjusted increments so that only such multiples of p are generated that are coprime with those small primes, in the first place.

Example

To find all the prime numbers less than or equal to 30, proceed as follows.

First, generate a list of integers from 2 to 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The first number in the list is 2; cross out every 2nd number in the list after 2 by counting up from 2 in increments of 2 (these will be all the multiples of 2 in the list):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~

The next number in the list after 2 is 3; cross out every 3rd number in the list after 3 by counting up from 3 in increments of 3 (these will be all the multiples of 3 in the list):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

The next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after 5 by counting up from 5 in increments of 5 (i.e. all the multiples of 5):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~

The next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after 7, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because 7×7 is greater than 30. The numbers not crossed out at this point in the list are all the prime numbers below 30:

2 3 5 7 11 13 17 19 23 29

1.4 Algorithms

Required algorithms to implement in a programming language.

1.4.1 Algorithm 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;
}
```

1.4.2 Algorithm 2

```
c[1] = false;
i = 2;
while (i <= n) {
  j=2*i;
  while (j <= n) {
    c[j] = false;
    j = j + i;
  }
  i = i + 1;
}
```

1.4.3 Algorithm 3

```
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;
}
```

1.4.4 Algorithm 4

```
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;
}
```

1.4.5 Algorithm 5

```
c[1] = faux;
i = 2;
while (i<=n) {
  j = 2;
  while (j <= sqrt(i)) {
    if (i % j == 0) {
      c[i] = false;
    }
    j++;
  }
  i++;
}
```

2 Code

This section shows my implementation and result for the 5 given Sieve of Eratosthenes algorithms.

2.1 Implementation

```python
import math


class SIEVE:
    def __init__(self, n) -> None:
        self.n = n
        self.prime = []

    def algorithm_1(self):
        self.prime = [True for i in range(self.n + 1)]
        self.prime[0] = False
        self.prime[1] = False
        i = 2

        while i <= self.n:
            if self.prime[i] == True:
                j = 2 * i
                while j <= self.n:
                    self.prime[j] = False
                    j = j + i
            i = i + 1

    def algorithm_2(self):
        self.prime = [True for i in range(self.n + 1)]
        self.prime[0] = False
        self.prime[1] = False
        i = 2

        while i <= self.n:
            j = 2 * i
            while j <= self.n:
                self.prime[j] = False
                j = j + i
            i = i + 1

    def algorithm_3(self):
        self.prime = [True for i in range(self.n + 1)]
        self.prime[0] = False
        self.prime[1] = False
        i = 2

        while i <= self.n:
            if self.prime[i] == True:
                j = i + 1
                while j <= self.n:
                    if j % i == 0:
                        self.prime[j] = False
                    j = j + 1
            i = i + 1

    def algorithm_4(self):
        self.prime = [True for i in range(self.n + 1)]
        self.prime[0] = False
        self.prime[1] = False
        i = 2

        while i <= self.n:
            j = 2
            while j < i:
                if i % j == 0:
                    self.prime[i] = False
                j = j + 1
            i = i + 1

    def algorithm_5(self):
        self.prime = [True for i in range(self.n + 1)]
        self.prime[0] = False
        self.prime[1] = False
        i = 2

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

    def get_prime(self):
        return self.prime
```

2.2 Algorithmic complexity

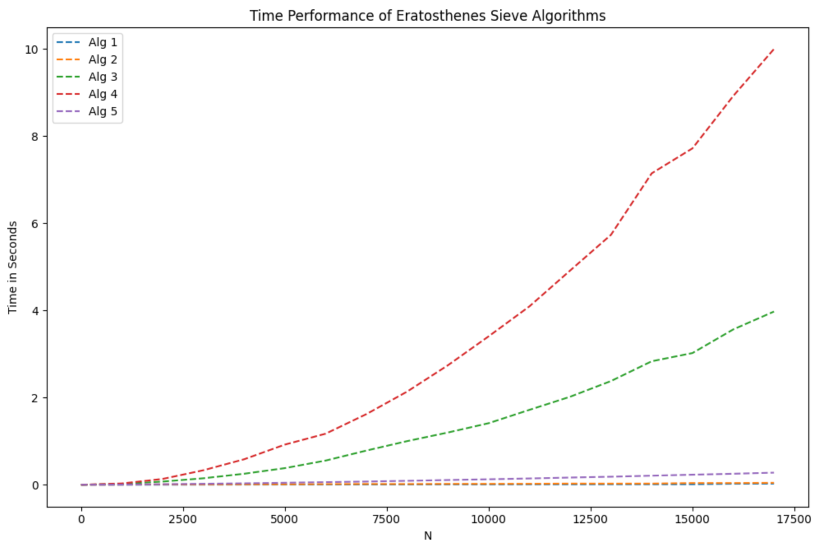
The sieve of Eratosthenes is a popular way to benchmark computer performance. The time complexity of calculating all primes below n in the random access machine model is O(n log log n) operations, a direct consequence of the fact that the prime harmonic series asymptotically approaches log log n. It has an exponential time complexity with regard to input size, though, which makes it a pseudo-polynomial algorithm. The basic algorithm requires O(n) of memory.

The bit complexity of the algorithm is O(n (log n) (log log n)) bit operations with a memory requirement of O(n).

The normally implemented page segmented version has the same operational complexity of O(n log log n) as the non-segmented version but reduces the space requirements to the very minimal size of the segment page plus the memory required to store the base primes less than the square root of the range used to cull composites from successive page segments of size $O(\frac{\sqrt{n}}{log(n)})$.

A special (rarely, if ever, implemented) segmented version of the sieve of Eratosthenes, with basic optimizations, uses O(n) operations and $O(\sqrt{n}\frac{log\ log\ n}{log\ n})$ bits of memory.

2.3 Graph / Results



3 Conclusion

To conclude, the current piece of work represents my personal implementation of 5 algorithms for obtaining Eratosthenes Sieve. The programming language I chose is Python. The results show the time performance in seconds based on the N limit. As you can see in the *Figure 1*, the first 3 algorithms performed the best with a complexity of O(0), while the last 2 - $O(n^2)$ and $O(log\, n)$.

References

- https://github.com/IuraCPersonal/FAF203-CC