Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 3

Discipline: Algorithms’ Analysis

Topic: Empirical analysis of algorithms for obtaining Eratosthenes Sieve

Elaborated: Țărnă Cristina,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Study and analyze different sorting algorithms

**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.

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.

**Introduction:**

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

**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 [1000, 1500, 2000, 2500, 5000, 10000, 15000, 20000, 50000, 100000, 500000, 1000000, 5000000, 10000000]

**IMPLEMENTATION**

All 5 algorithms will be implemented in python an analyzed empirically based on the time required for their completion.

**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;

}

**Implementation:**

The code initializes a boolean list c where each index i corresponds to the number i, and sets all values to True except for c[0] and c[1] which are marked as not prime.

The outer loop iterates over all integers from 2 to n. If c[i] is True, the inner loop starts at j=2\*i and marks all j values in the list c as False, since they are multiples of i and not prime.

Finally, the code loops over all integers from 2 to n and prints the ones that are still marked as True, which correspond to the prime numbers.

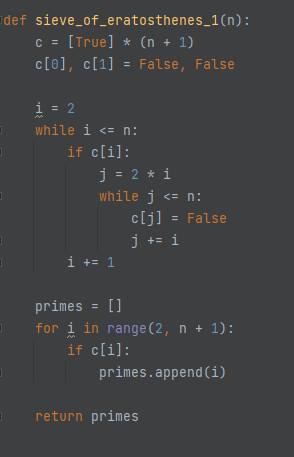


Figure 1. Implementation Algorithm 1

**Results:**

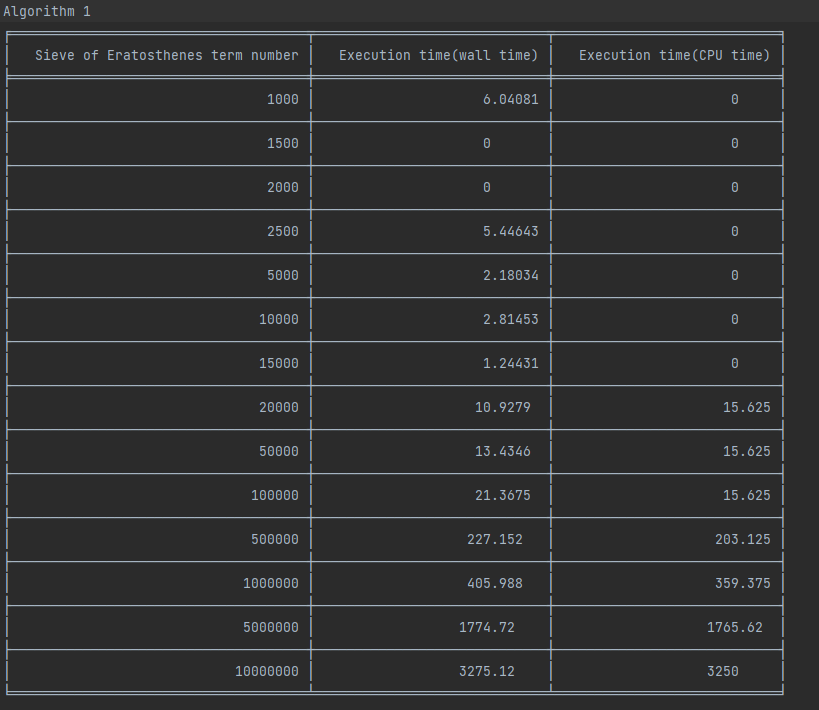


Figure 2. Result Algorithm 1

**The plot:**

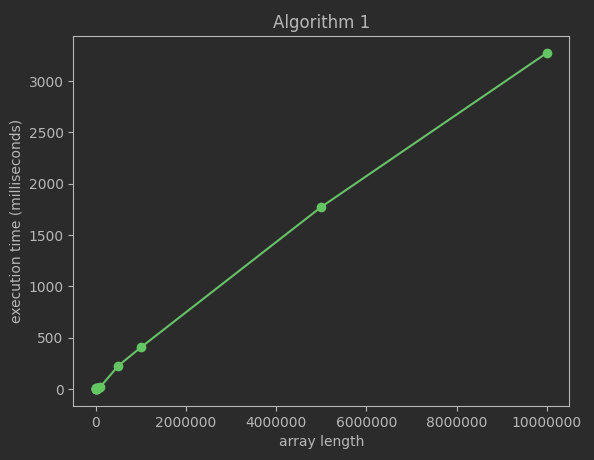


Figure 3. Plot of Algorithm 1

**Algorithm 2**

C[1] =false;

i=2;

while (i<=n){

j=2\*i;

while (j<=n){

c[j] =false;

j=j+i;

}

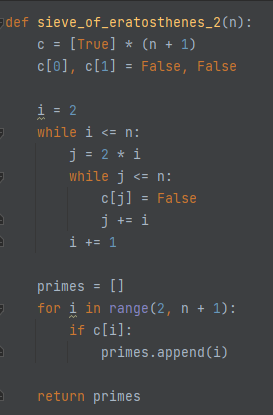
i=i+1;

**Implementation:**

The code initializes a boolean list c where each index i corresponds to the number i, and sets all values to True except for c[0] and c[1] which are marked as not prime.

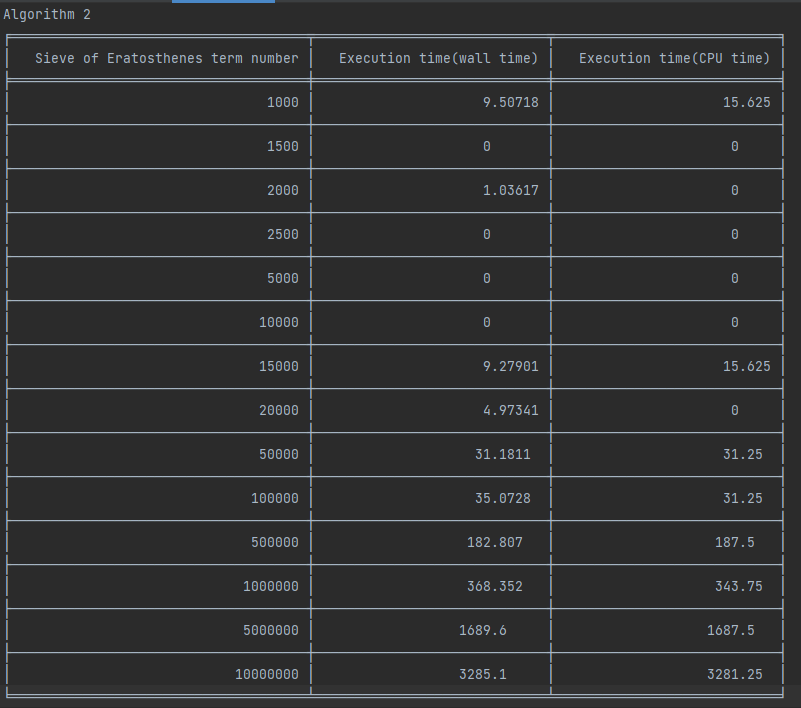
The outer loop iterates over all integers from 2 to n. The inner loop starts at j=2\*i and marks all j values in the list c as False, since they are multiples of i and not prime.

Finally, the code loops over all integers from 2 to n and prints the ones that are still marked as True, which correspond to the prime numbers.



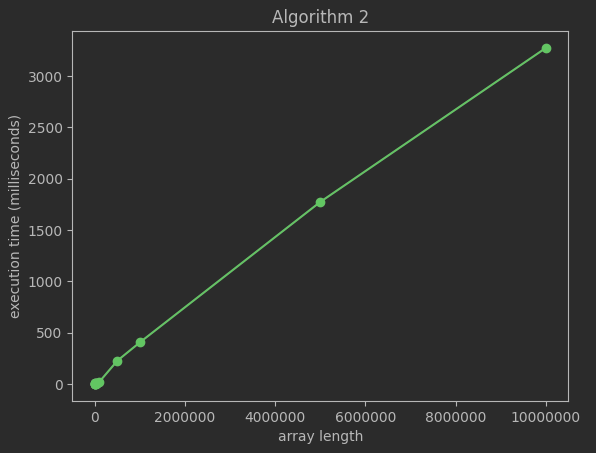
**Figure 4. Implementation Algorithm 2**

**Results:**



**Figure 5. Result Algorithm 2**

**The plot:**



**Figure 6. Plot of Algorithm 2**

**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;

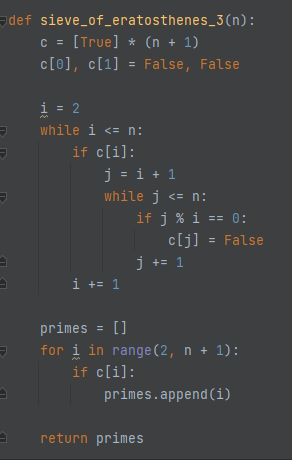
}

The function takes an integer n as input and returns a list of prime numbers up to n using a modified version of the Sieve of Eratosthenes algorithm.

The implementation is similar to the previous functions, but instead of marking all multiples of i as composite, the function only marks the multiples that are greater than i (starting from j=i+1). This is because all the multiples of i that are less than i have already been marked as composite by previous iterations.

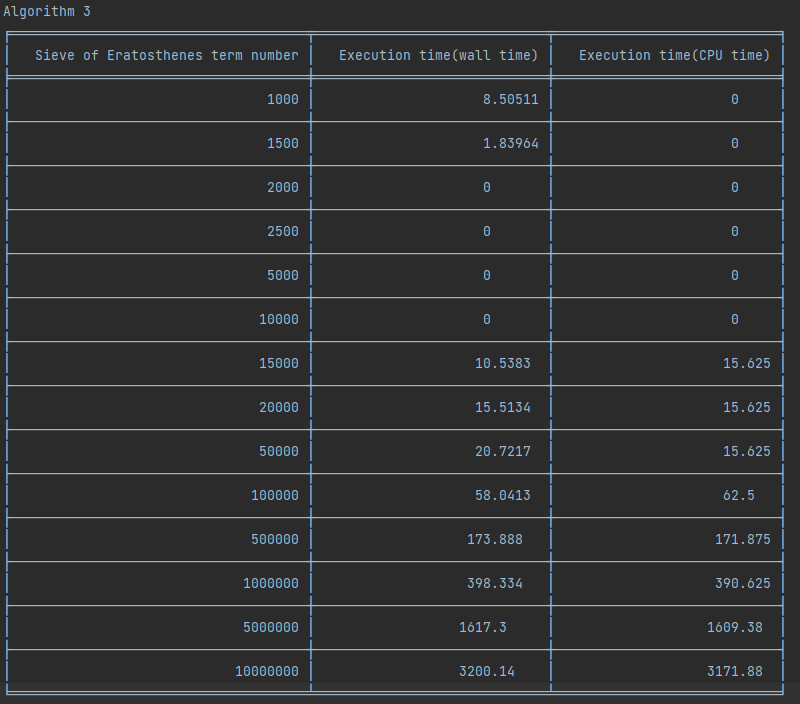
The function then stores the prime numbers in a list called primes and returns it at the end.

**Implementation:**



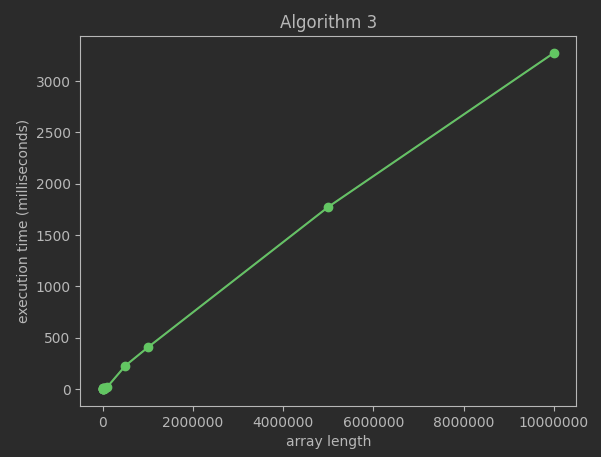
**Figure 7. Implementation Algorithm 3**

**Results:**



**Figure 8. Result Algorithm 3**

**The plot:**



**Figure 9. Plot Algorithm 3**

**Algorithm 4**

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;

}

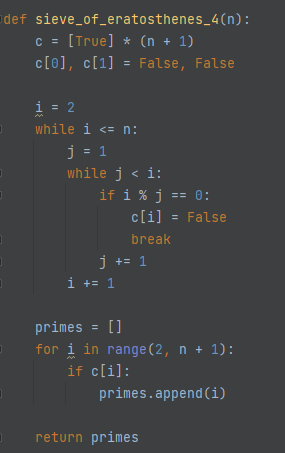
The function takes an integer n as input and returns a list of prime numbers up to n using a simple algorithm that checks all numbers less than i to see if they divide i evenly.

The implementation initializes a boolean list c where each index i corresponds to the number i, and sets all values to True except for c[0] and c[1] which are marked as not prime.

The outer loop iterates over all integers from 2 to n. The inner loop starts at j=1 and checks all numbers less than i to see if they divide i evenly. If any such number is found, c[i] is marked as False and the loop is exited using break.

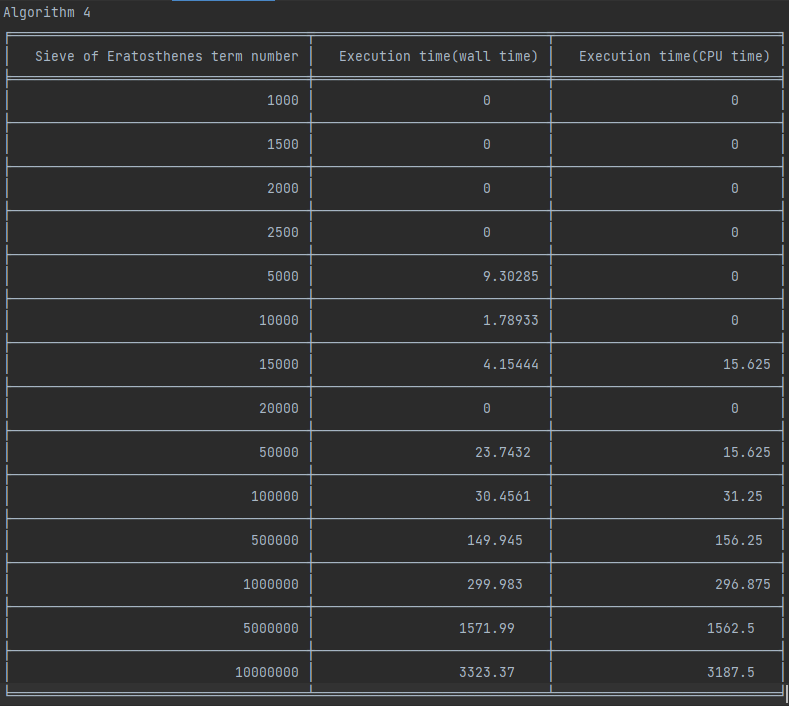
Finally, the code loops over all integers from 2 to n and stores the ones that are still marked as True in a list called primes, which is returned at the end of the function.

**Implementation:**



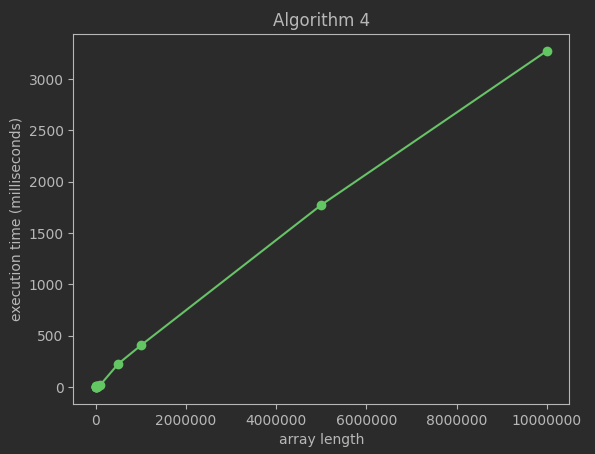
**Figure 10. Implementation Algorithm 4**

**Results:**



**Figure 11. Result Algorithm 4**

**The plot:**



**Figure 12. Plot Algorithm 4**

**Algorithm 5**

C[1] = false;

i=2;

while (i<=n){

j=2;

while (j<=sqrt(i)){

if (i % j == 0) {

c[i] = false;

}

j++;

}

i++;

}

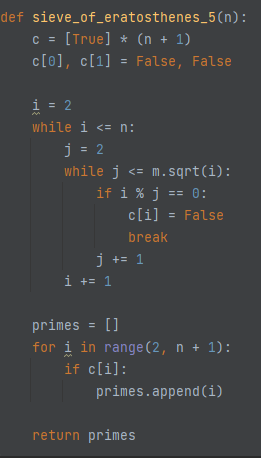
The function takes an integer n as input and returns a list of prime numbers up to n using a simple algorithm that checks all numbers less than or equal to the square root of i to see if they divide i evenly.

The implementation initializes a boolean list c where each index i corresponds to the number i, and sets all values to True except for c[0] and c[1] which are marked as not prime.

The outer loop iterates over all integers from 2 to n. The inner loop starts at j=2 and checks all numbers less than or equal to the square root of i to see if they divide i evenly. If any such number is found, c[i] is marked as False and the loop is exited using break.

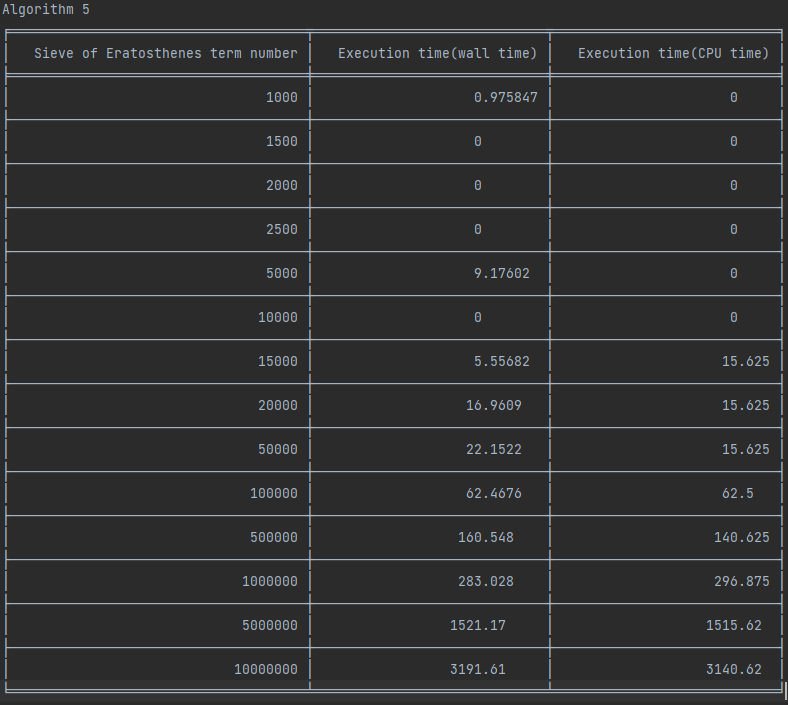
Finally, the code loops over all integers from 2 to n and stores the ones that are still marked as True in a list called primes, which is returned at the end of the function.

**Implementation:**



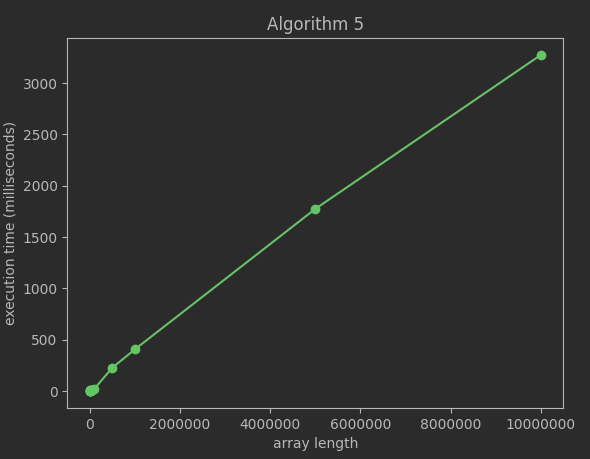
**Figure 13. Implementation Algorithm 5**

**Results:**

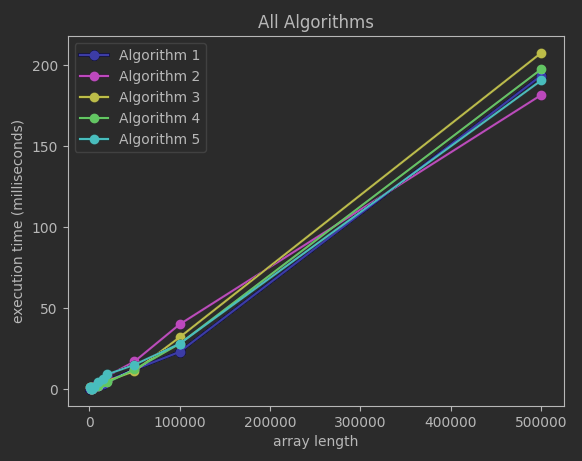


**Figure 14. Result Algorithm 5**

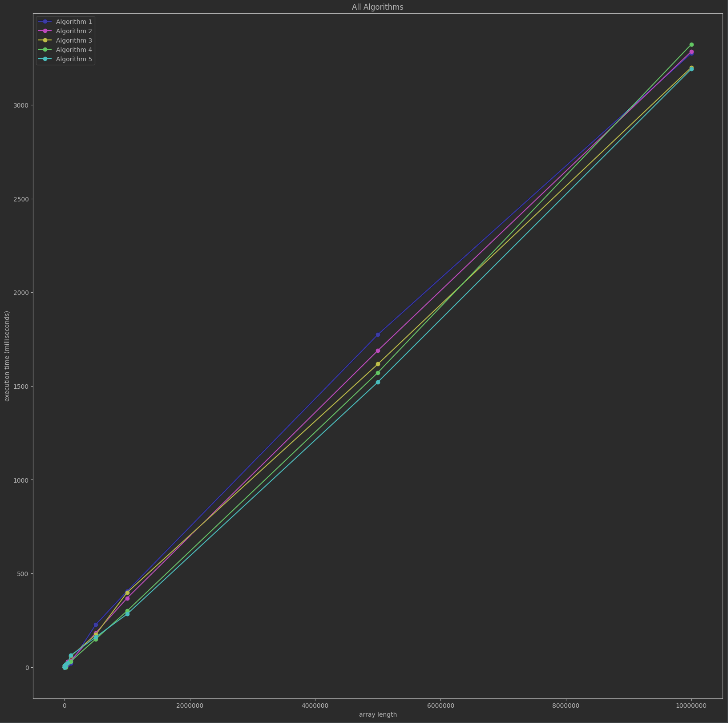
**The plot:**



**Figure 15. Plot Algorithm 5**



**Figure 16. All algorithms zoomed in**



**Figure 17. All algorithms**

**Conclusion:**

The algorithms are very similar based on time complexity, and each time perform differently. So, I cannot say which one is the best, but there will be no problem to choose either of them.

**Link to GitHub:** [**https://github.com/CristinaT21/APA\_LABS**](https://github.com/CristinaT21/APA_LABS)