**LABORATORY 1**

**GONZALO DE VARONA**

**JUAN JOSÉ RESTREPO**

**ESTEBAN YUSUNGUAIRA**

**A00358687**

**A00359137**

**A00358393**

**ALGORITHMS AND DATA STRUCTURES**

**ICESI UNIVERSITY**

**FEBRUARY 16 2020**

**Phase 1. Problem identification**

A small business is interested in improving its software's security by implementing prime numbers into encryption algorithms, so requires the development of a software capable of generate a certain quantity of prime numbers for their posterior security purposes. This application must implement three different algorithms to complete the task.

**Phase 2. Gathering information**

A prime number (or prime integer, often simply called a "prime" for short) is a [positive integer](http://mathworld.wolfram.com/PositiveInteger.html) *p>1* that has no positive integer [divisors](http://mathworld.wolfram.com/Divisor.html) other than *1* and *p* itself.

We use the exactly divisors for do the math operation in this case, we only use numbers and the result in integer. In general, Beginning in number one and the other prime numbers, we can possible create the rest of the numbers. That is what say the gregorian math Euclides, he proved that there is an infinite supply of prime numbers and all numbers greater than 1 or the result of multiplication of a combination unique of numbers primes. For this reason, In the actually, the enterprises use cryptography of public key, that is the reason why our data electronic is safe, because the time for searching prime numbers in the algorithm is very high, this is why as the data grows, it will be much slower and the next number may be the prime number or not. All of this information is giving for asymptotic law:

= x/Ln(x)

=x/(Ln(x)-1)

=Integral[dt/Ln(t),{t=2,x}].

That is why the distance of the prime numbers is more separated that the amount that we set. But one of the problems for this case is the distribution of the “program” because the program take so much for find two prime numbers, one solution is safe a collection of numbers that the company generated and select a two random numbers of that collection, other solution is the client do it in his computer when he installed the program, but takes the initial problem but is more faster than the first.

**Phase 3. Seek for creative solutions**

**Idea 1:** Use Java libraries in order to make an algorithm to get the first *n* prime numbers.

**Idea 2:** Use the Sieve of Eratosthenes algorithm in order to get all the prime numbers below an *n* number. The strategy of the algorithm is:

1. Create a list of consecutive integers from 2 to *n*: (2, 3, 4, …, *n*).

2. Initially, let *p* equal 2, the first prime number.

3. Starting from *p*2, count up in increments of *p* and mark each of these numbers greater than or equal to *p2* itself in the list. These numbers will be *p(p+1)*, *p(p+2)*, *p(p+3)*, …

4. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from step 3.

This sieve is one of the most efficient ways to obtain the prime numbers below a given integer lower than 10 million.

**Idea 3:** Use the segmented sieve. The basic idea of this sieve is to divide the range [0…n-1] in different segments and compute prime numbers in all segments one by one. It uses the sieve of Eratosthenes to find primes smaller or equal to √(n).

1. Use Sieve of Eratosthenes to find all primes up to square root of ‘n’ and store these primes in an array “prime[]”. Store the found primes in an array ‘prime[]’.

2. We need all primes in range [0..n-1]. We divide this range in different segments such that size of every segment is at-most √n

3. Do following for every segment [low…high]

○ Create an array mark[high-low+1]. Here we need only O(x) space where x is number of elements in given range.

○ Iterate through all primes found in step 1. For every prime, mark its multiples in given range [low...high].

In the sieve of Eratosthenes, we need O(n) space which may not be feasible for large n. Using this sieve, we only need O(√n )

**Idea 4:** The Sieve of Sundaram

1) The sieve of Sundaram, produces primes smaller than (2\*x + 2) for a number given number x. Since we want primes smaller than n, we reduce n-2 to half. We call it nNew.

nNew = (n-2)/2;

For example, if n = 102, then nNew = 50.

2) Create an array marked[n] that is going to be used to separate numbers of the form i+j+2ij from others where 1 <= i <= j

3) Initialize all entries of marked[] as false.

4) Mark all numbers of the form i + j + 2ij as true where 1 <= i <= j

Loop for i=1 to nNew

a) j = i;

b) Loop While (i + j + 2\*i\*j) 2, then print 2 as first prime.

6) Remaining primes are of the form 2i + 1 where i is index of NOT marked numbers. So print 2i + 1 for all i such that marked[i] is false.

**Idea 5:** Bitwise Sieve. Optimized implementation of the simple sieve (Eratosthenes).

The idea is the same as a normal Sieve of Eratosthenes but it can save space using bits. Instead of using 8 bits to store a true or false value, it only uses 1.

We can optimize space to n/8 by using individual bits of an integer to represent individual primes. We create an integer array of size n/64. Note that the size of array is reduced to n/64 from n/2 (Assuming that integers take 32 bits).

**Idea 6:** Sieve of Atkin.

The sieve of Atkin is a modern algorithm for finding all prime numbers lower than an n specified integer. Unlike the Eratosthenes sieve, the sieve of Atkin does some preliminary work and then marks off multiples of squares of primes, achieving a better theoretical asymptotic complexity (n/Log(Log n)).

1. Create a results list, filled with 2, 3, and 5.

2. Create a sieve list with an entry for each positive integer; all entries of this list should initially be marked non-prime.

3. For each entry number n in the sieve list, with modulus-sixty remainder r:

a. If r is 1, 13, 17, 29, 37, 41, 49, or 53, flip the entry for each possible solution to 4x2 + y2 = n.

* 1. If r is 7, 19, 31, or 43, flip the entry for each possible solution to 3x2 + y2 = n.

c. If r is 11, 23, 47, or 59, flip the entry for each possible solution to 3x2 – y2 = n when x > y.

* 1. If r is something else, ignore it completely.

4. Start with the lowest number in the sieve list.

5. Take the next number in the sieve list still marked prime.

6. Include the number in the results list.

7. Square the number and mark all multiples of that square as non prime. Note that the multiples that can be factored by 2, 3, or 5 need not be marked, as these will be ignored in the final enumeration of primes.

8. Repeat steps four through seven.

**Phase 4.** Transition from ideas to preliminary designs

**Idea 1: *Discarded*.** Java libraries aren’t enough to obtain the prime numbers below a specified integer. They can be as a tool to determine if an integer is prime or not but not to solve the entire problem. We could use the class *BigInteger* and its method bigInt.isProbablePrime or use *Apache Commons Math* and its method org.apache.commons.math3.primes.Primes.

**Idea 2:** The sieve of Eratosthenes is very simple and effective. The fastest algorithms that obtain prime numbers use optimized versions of this sieve, indeed, almost all of the ideas are based on the implementation of this simple sieve.

**Idea 3: *Discarded.*** It controls memory use, and also makes it run faster. This is because it splits the range into segments and analyze it separately. It’s really nice for large n (over a million) but in this case, it doesn’t matter too much because the app will only work with small integers. As we said before, the segmented sieve uses de Eratosthenes sieve, so it wouldn’t be so different than its simple implementation. That’s why we rather use the Eratosthenes sieve directly.

**Idea 4:** Considering that the sieve of Sundaram has a O(nLog n) time complexity, it has a great performance in small values. In fact, according to a paper made by the Universitas Malikussaleh in Indonesia that compares the basic algorithm of the sieve of Sundaram with the basic implementation of de sieve of Eratosthenes; Sundaram has better results with small values and Eratosthenes with big ones.

**Idea 5:** While the other sieves focuses on the time complexity, the Bitwise sieve focuses on optimization of the memory usage. This sieve is not that useful because even a basic Raspberry pi could handle the memory usage that any sieve requires for this problem but, for educational purposes, we should consider this one and focus on the bits usage and analyze its spatial complexity.

**Idea 6: *Discarded*.** The sieve of Atkin is indeed the newest method and under certain assumptions, it has the best asymptotic complexity but: 1) It does slow down as the size increases; 2) it has higher overhead and requires some non-trivial work to properly implement; and 3) Implementations using the pseudocode of the general idea are quite a bit slower than correct simple sieve of Eratosthenes implementations.

**Idea 7:** Create an algorithm that obtain all the prime numbers by analyzing just the odd integers from 3 to n.

**Phase 5. Evaluation and selection of the best solutions**

**Item 1: Ease of implementation**

**[0]** The algorithm needs a specific implementation and some changes to function properly with the mentioned pros/cons.

**[1]** The algorithm uses operations that are difficult to understand or implement due we’ve not seen it or worked with it before.

**[2]** The algorithm uses basic operations to solve the problem.

**[3]** The algorithm uses basic operations to solve the problem and can be implemented in a compact and short method.

**Item 2: Efficiency**

**[0]** O(n^2)

**[1]** O(nLog(Log n)), O(nLogn) or O(n) .

**Item 3: Course objectives**

**[1]** The algorithm focuses on time complexity but not on the spatial complexity

**[2]** The algorithm focuses on time complexity and the spatial complexity

**Item 4: Exact solution**

**[0]** The algorithm gives an approximated solucion

**[1]** The algorithm gives an exact solution.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Item 1** | **Item 2** | **Item 3** | **Item 4** | **Total** |
| **Sieve of Eratosthenes** | 2 | 1 | 1 | 1 | 4 |
| **sieve of Sundaram** | 0 | 1 | 1 | 1 | 3 |
| **Bitwise sieve** | 1 | 1 | 2 | 1 | 4 |
| **Custom implementation** | 3 | 0 | 1 | 1 | 4 |

**Phase 6: Reports and specifications**

**Pseudocode**

**Sieve of Eratosthenes**

**input: an integer *n* > 1.**

**output: all prime numbers from 2 through *n*.**

**let *A* be an array of** [**Boolean**](https://en.wikipedia.org/wiki/Boolean_data_type) **values, indexed by integers 2 to *n*,**

**initially all set to true.**

**for *i* = 2, 3, 4, ..., not exceeding *√n* do**

**if *A*[*i*] is true**

**for *j* = *i*2, *i*2+*i*, *i*2+2*i*, *i*2+3*i*, ..., not exceeding *n* do**

***A*[*j*] := false**

**return all *i* such that *A*[*i*] is true.**

**Our prime generator**

1) Initialize an integer x= 1

2) Initialize a list and add the number 2.

3) For (i=1; x<given number; i++)

Initialize a counter = 0

Inside the this for, the algorithm will only evaluate the odd integers >2, that’s why our new x will be (2\*i)+1

4) Verify if x is a prime analyzing the modulus of x and all the odd integers lower than x. If the result of that modulus is 0, count+1

5) If the counter is exactly 2, the algorithm add x to the list

6) Repeat

**Bitwise Sieve**

Boolean[] finalArray <- new boolean[index+1]

If index equals 1

finalArray <- new Boolean[2]

finalArray[1] <- false

Else

Boolean[] primeArray<- new boolean[index/2 +1]

For i = 0 to primeArray.length

primeArray[i] <- false

For composite =3, composite \* composite to index, composite +=2

If primeArray[composite/2]= false

For múltiple = composite \* composite, múltiple to Index, multiple += composite\*2

primeArray[multiple/2] = true

finalArray[1] <- false

finalArray[2] <- true

For i =3 to finalArray[].length, i+=2

If primeArray[i/2] <- false

finalArray[i] <- false

return finalArray

**Phase 7: Design implementation**

**Functional Requirements**

|  |  |
| --- | --- |
| **F.R1** | **Generate prime numbers** |
| **Abstract** | The application is able to generate all the prime numbers lower than *n*. |
| **Input** | The highest number that the application will generate (*n*). Beyond this point, the application won’t search for more prime numbers. |
| **Output** | All the prime numbers lower than *n*. |

|  |  |
| --- | --- |
| **FR.2** | **Generate matrix** |
| **Abstract** | The application is able to generate a matrix that represent all numbers lower than *n*. |
| **Input** | - |
| **Output** | Generates the required matrix. |

|  |  |
| --- | --- |
| **FR.3** | **Highlight the prime numbers in the matrix** |
| **Abstract** | The application is able to highlight the prime numbers in the matrix by colouring them with green and red the non prime numbers. |
| **Input** | - |
| **Output** | Prime numbers are highlighted in the matrix |

|  |  |
| --- | --- |
| **FR.4** | **Show in real time the process of algorithm** |
| **Abstract** | The application is able to show in real time the prime numbers lower than n in a matrix and highlight them |
| **Input** | - |
| **Output** | The process is shown in real time |

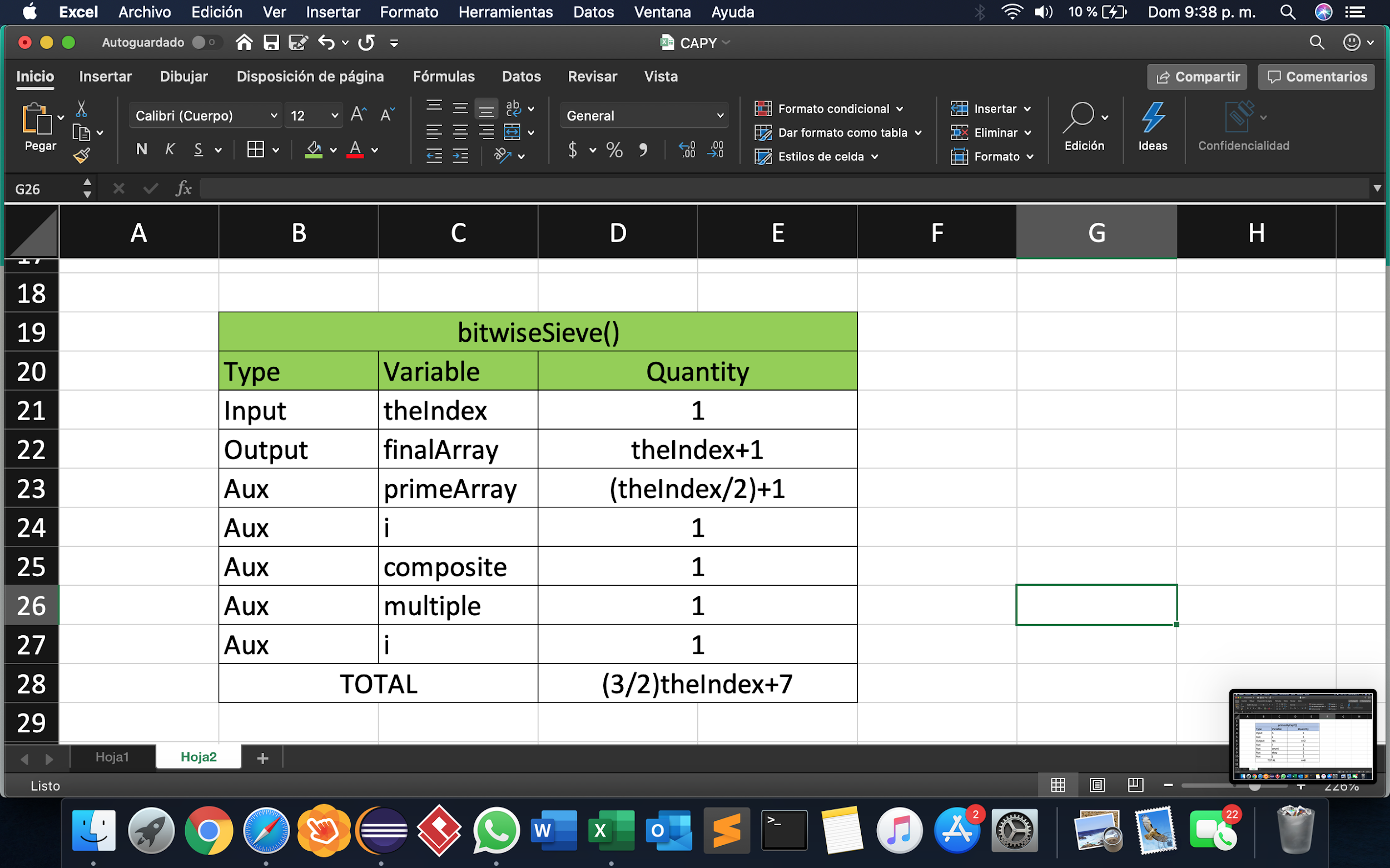
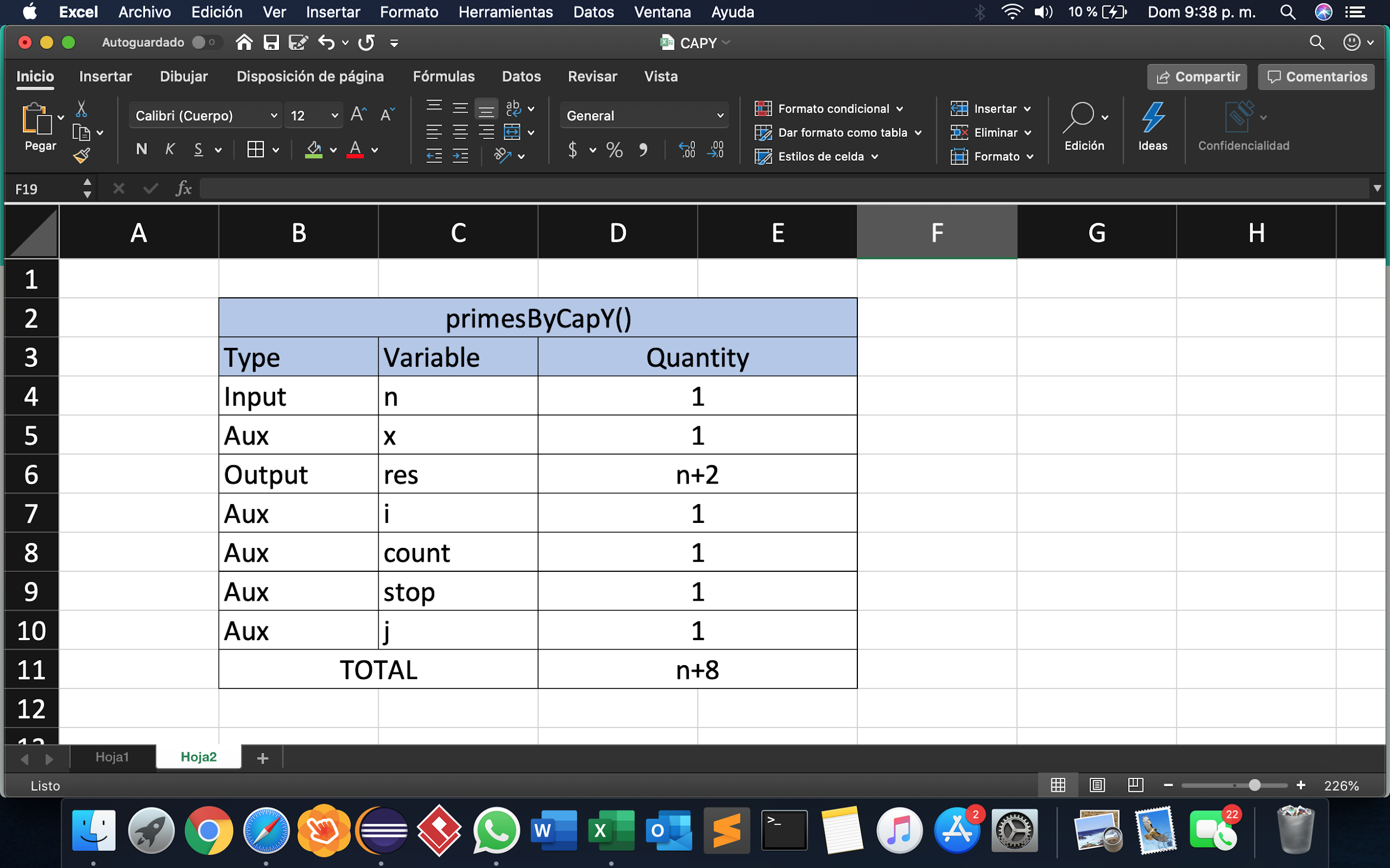
**Non-Functional Requirements**

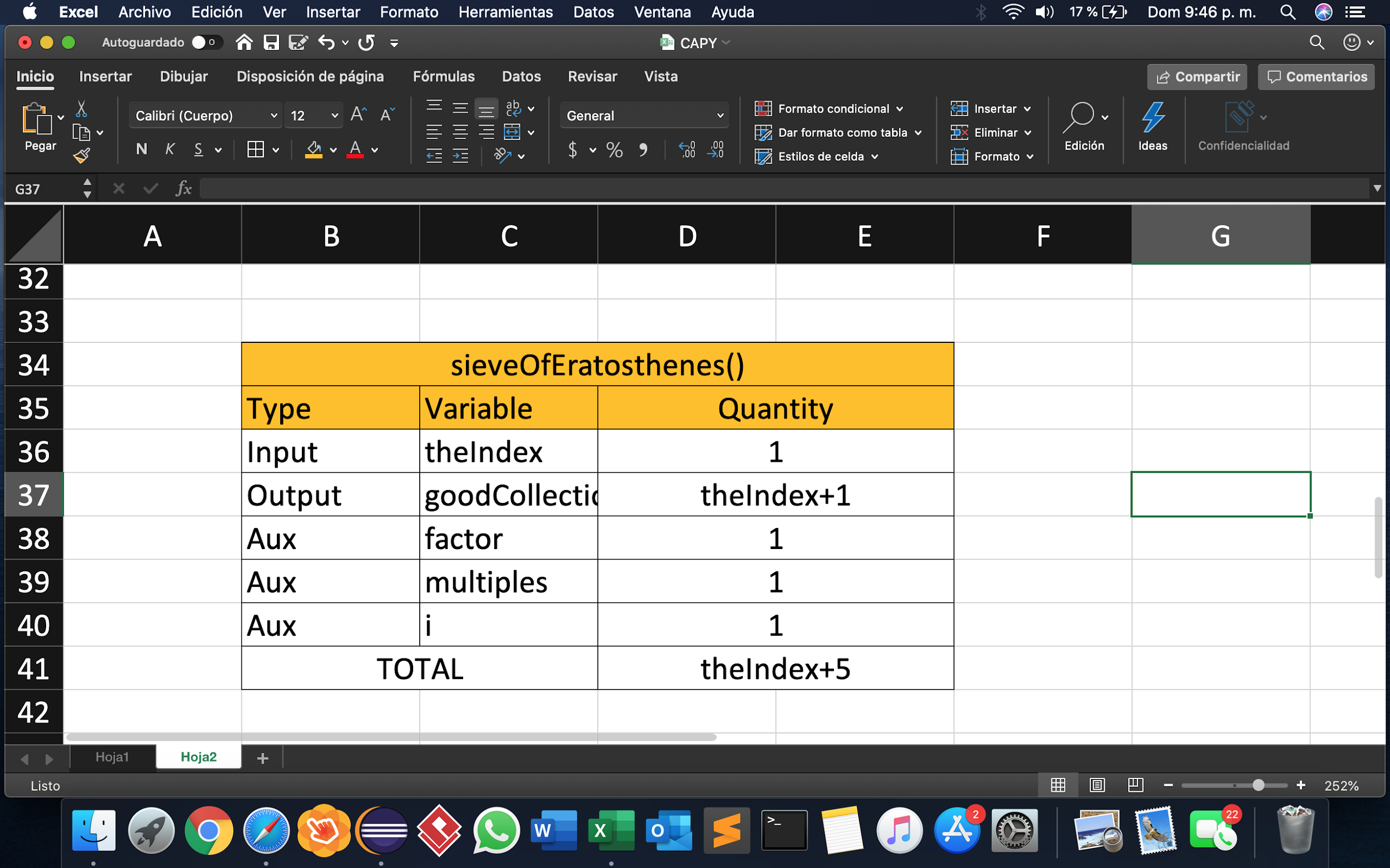
|  |  |
| --- | --- |
| **NFR.1** | **Matrix must be as squared as possible** |
| **Abstract** | The generated matrix must be as squared as possible, for instance, if there are 100 numbers, the matrix must be 10x10 |
| **Output** | Square matrix |

|  |  |
| --- | --- |
| **NFR.2** | **Implement 3 algorithms for finding prime numbers** |
| **Abstract** | There must be 3 algorithms for finding prime numbers in order to make the user choose which algorithm wants to use. |
| **Output** | Prime numbers can be found with 3 different algorithms |

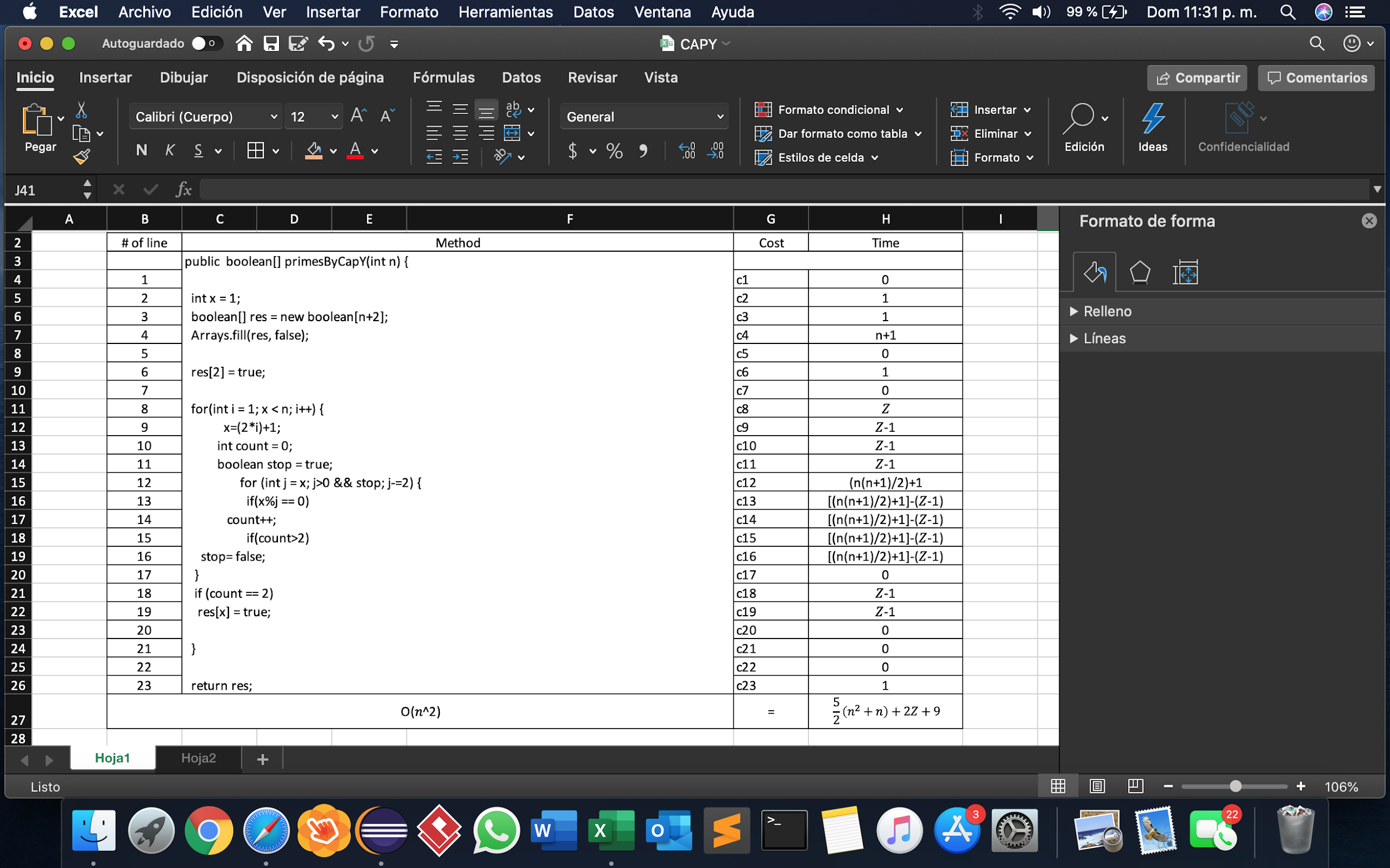
|  |  |
| --- | --- |
| **NFR.3** | **Application must have Graphic User Interface** |
| **Abstract** | The application must have a Graphic User Interface |
| **Output** | GUI is made for the application with given standards |

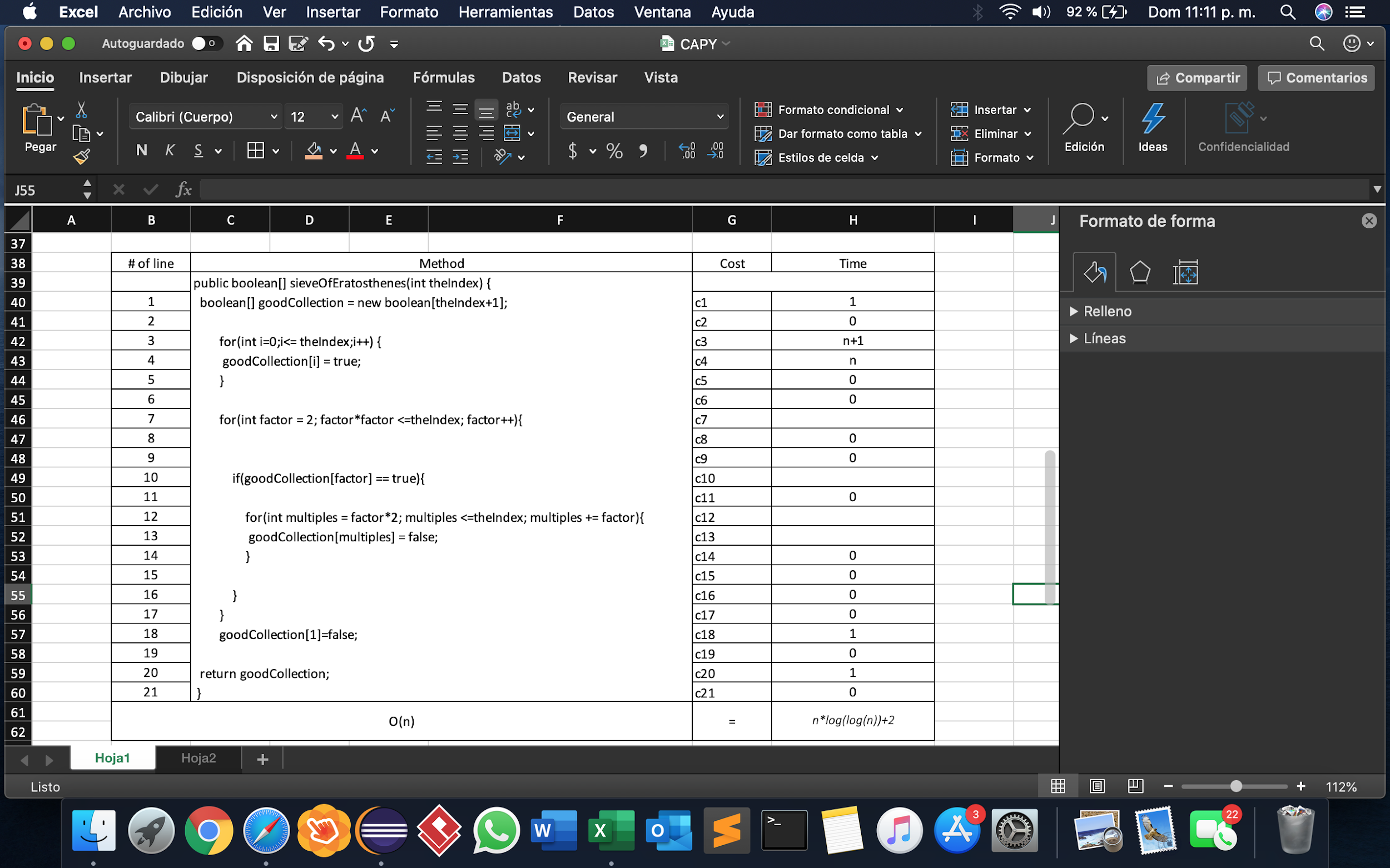
**Space complexity**

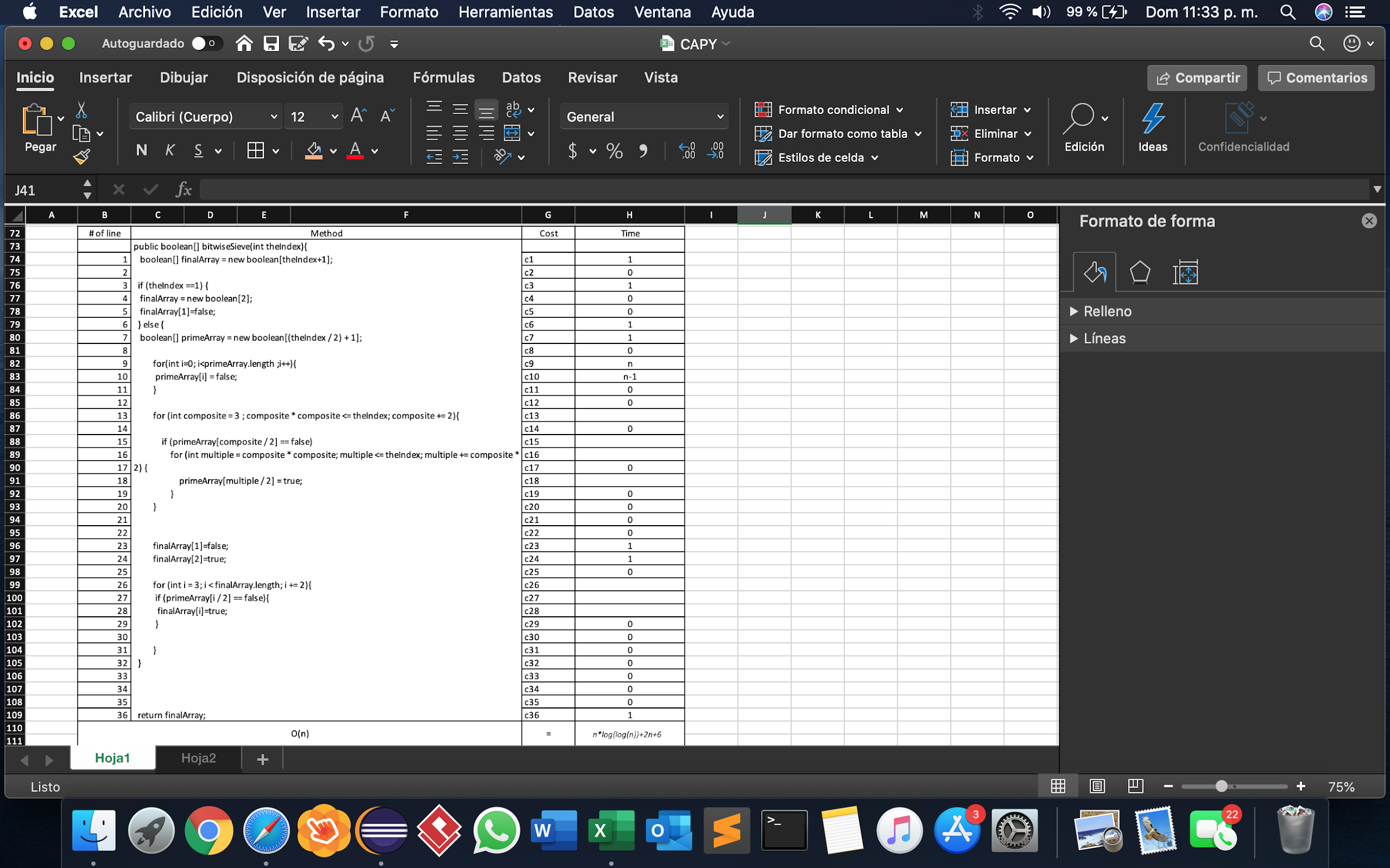
****

****

**Time Complexity**

****

****

****

**Bibliography**

[Weisstein, Eric W.](http://mathworld.wolfram.com/about/author.html) "Prime Number." From [*MathWorld*](http://mathworld.wolfram.com/)--A Wolfram Web Resource.<http://mathworld.wolfram.com/PrimeNumber.html>

Chaudhary, N. (n.d.). *Prime Numbers - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/prime-numbers/ [Accessed Feb. 2020].

Jacobsen, D. (2016). What are the pros and cons of various sieving methods for finding and testing prime numbers?. Retrieved 17 February 2020, from <https://www.quora.com/What-are-the-pros-and-cons-of-various-sieving-methods-for-finding-and-testing-prime-numbers>

Abdullah, D. et al 2018 J. Phys.: Conf. Ser. 978 012123, Retrieved 17 February 2020, from <https://www.researchgate.net/publication/323788948_Prime_Numbers_Comparison_using_Sieve_of_Eratosthenes_and_Sieve_of_Sundaram_Algorithm>

How is the time complexity of Sieve of Eratosthenes is n\*log(log(n))? - GeeksforGeeks. (2020). Retrieved 17 February 2020, from https://www.geeksforgeeks.org/how-is-the-time-complexity-of-sieve-of-eratosthenes-is-nloglogn/