

Notación Big O

1. Introducción

La notación binaria y lineal buscan el número máximo de intentos necesarios para completar un algoritmo, pero otro dato de interés es el tiempo y para calcularlo es necesario tener dos ideas principales; primero que determinamos el tiempo del algoritmo en base al tamaño de su entrada y la segunda es que tan rápido crece una función en base a su entrada, a esto se le conoce como **tasa de crecimiento**, cuya función es " $an^2 + bn + c$ ", donde:

$$a > 0$$

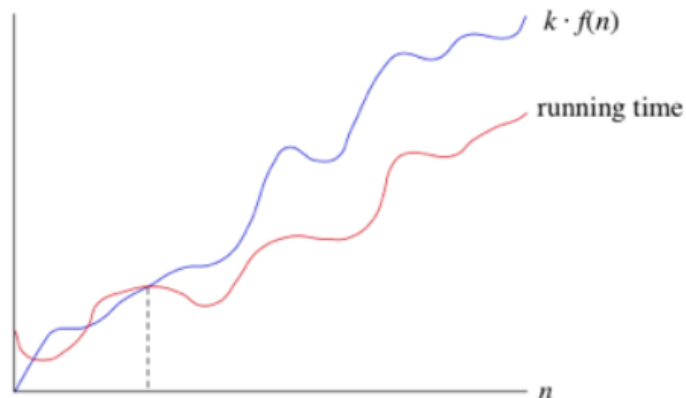
$an^2 > bn + c$ donde siempre existirá un valor an^2 mayor que $bn + c$ y conforme aumente n la diferencia se intensifica.

Al descartar los términos poco significativos y las constantes, podemos centrarnos en la parte importante del tiempo de ejecución de un algoritmo (su tasa de crecimiento) sin necesidad de tomar en cuenta detalles que complican el entendimiento. Descartar los coeficientes constantes y los términos menos significativos es **notación asintótica**.

2. ¿Qué es la notación Big O?

La notación Big O es una notación asintótica que nos dice cuánto se tarda un algoritmo en el peor de los casos. Si el tiempo de una función es $O(f(n))$, por consiguiente para una n lo suficientemente grande, el tiempo de ejecución es a lo máximo $k \cdot f(n)$ para alguna constante k , donde k es el tiempo que tu computadora, lenguaje y compilador, entre otros factores tardan en hacer un cálculo matemático.

Ejemplo:



Por lo que se entiende que se usa la notación Big O para cotas superiores asintóticas, dado que delimita el aumento del tiempo de ejecución por arriba para entradas lo suficientemente grandes.

3. Complejidad de algoritmos

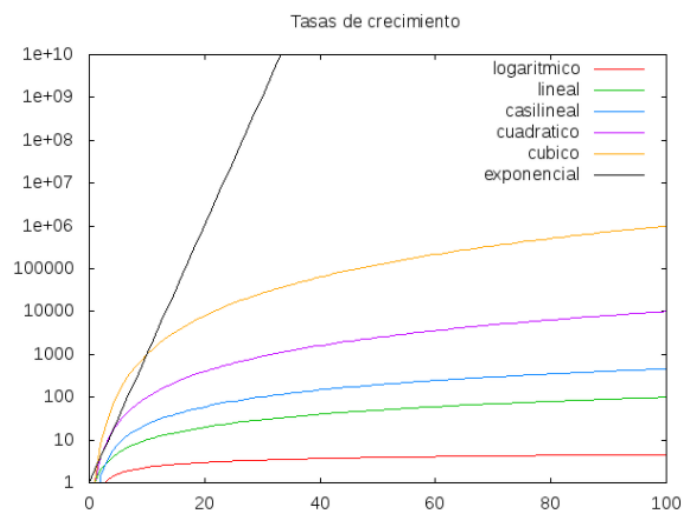
La complejidad algorítmica representa la cantidad de recursos (temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo.

Una vez creado un algoritmo se definen criterios para conocer el rendimiento y comportamiento. Estos criterios se dirigen a su simplicidad y al uso eficiente de recursos.

El uso eficiente de recursos se mide en base a dos parámetros: el espacio(memoria que utiliza) y el tiempo(lo que tarda en ejecutarse).Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de entre varios que solucionan un mismo problema.

Órdenes de complejidad:

Orden	Nombre
$O(1)$	Constante
$O(\log n)$	Logarítmica
$O(n)$	Lineal
$O(n \log n)$	Casi lineal
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(a^n)$	Exponencial



4. Problemas tratables e intratables.

Problema Tratable: Se puede resolver por un algoritmo en tiempo polinomial

– La cota superior (upper bound) es polinomial

Problema Intratable: No se puede resolver por un algoritmo en tiempo polinomial

– La cota inferior (lower bound) es exponencial

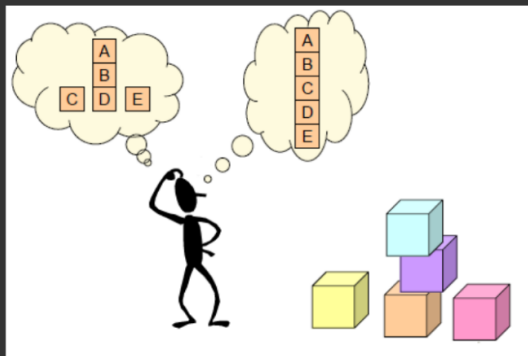
Clasificación P:

Cuando el tiempo de ejecución de un algoritmo (mediante el cual se obtiene una solución al problema) es menor que un cierto valor calculado a partir del número de variables implicadas (generalmente variables de entrada) usando una fórmula polinómica, se dice que dicho problema se puede resolver en un tiempo polinómico.

EJEMPLO CASE P: Multiplicación matricial

Dadas dos matrices A y B de tamaño $n \times n$, hallar C, el producto de las dos.

La solución del algoritmo seria:



```
int i,j,k;
int A[n][n],B[n][n],C[n][n];

// Dar valores a A y B.

for(i=0;i<n;i++)
{
  for(j=0;j<n;j++)
  {
    for(k=0;k<n;k++)
      C[i][j]+=A[i][k]*B[k][j];
  }
}
```

Clasificación NP:

Muchos de estos problemas pueden caracterizarse por el hecho de que puede aplicarse a un algoritmo polinómico para ser comprobados sus posibles soluciones, de esta forma se determina si es válida o inválida. Por medio de esta característica podemos ejecutar un método de resolución no determinista, que consiste en aplicar métodos heurísticos para obtener soluciones hipotéticas que se van desestimando aceptando al ritmo polinómico.

Los problemas NP (N- no determinista y P - polinómico). Los problemas P son subconjunto de los NP.

EJEMPLO NP: Problema CLIQUE

Dado un grafo G y un entero k , ¿es posible encontrar un subgrafo de G completo de tamaño k ?

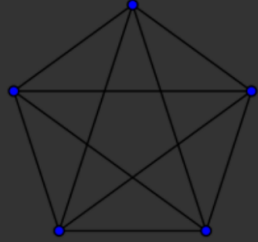
¿Porque?

- Claramente CLIQUE pertenece a NP.
- Ahora deberemos hacer una reducción de SAT a NP.
- Supongamos que tenemos una fórmula en (FNC):

Ahora nombramos:
 $C_1 \vee C_2 \vee \dots \vee C_k$ con n variables proposicionales.

Formaremos un grafo G con un nodo por cada literal que aparece en cada cláusula. Cada nodo está etiquetado con el literal que le dio origen. Agregaremos un arco entre un nodo etiquetado con l y un nodo etiquetado con l_0 si y solo si:

- l y l_0 están en cláusulas distintas.
- l no es el literal complementario de l_0 .



Clasificación NP-Completo:

Problemas de tipo NP de los cuales algunos destacan por su extrema complejidad.

Gráficamente se puede decir que algunos de estos problemas se encuentran en la frontera de la clase NP. Estos problemas se caracterizan por ser iguales en sentido de que si se descubriera una solución P en alguno de ellos, ésta solución sería aplicado para todos ellos.

5. Describa tres problemas que sean NP-completos

En teoría de la complejidad computacional, la clase de complejidad NP-completo es el subconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo. Se puede decir que los problemas de NP-completo son los problemas más difíciles de NP y muy probablemente no formen parte de la clase de complejidad P. La razón es que de tenerse una solución polinómica para un problema NP-completo, todos los problemas de NP tendrían también una solución en tiempo polinómico. Si se demostrara que un problema NP-completo, llamémoslo A, no se pudiese resolver en tiempo polinómico, el resto de los problemas NP-completos tampoco se podrían resolver en tiempo polinómico. Esto se debe a que si uno de los problemas NP-completos distintos de A, digamos X, se pudiese resolver en tiempo polinómico, entonces A se podría resolver en tiempo polinómico, por definición de

NP-completo. Ahora, pueden existir problemas en NP y que no sean NP-completos para los cuales exista solución polinómica, aun no existiendo solución para A.

1. Problema de satisfacibilidad booleana: En teoría de la complejidad computacional, el Problema de satisfacibilidad booleana (también llamado SAT) fue el primer problema identificado como perteneciente a la clase de complejidad NP-completo. El problema SAT es el problema de saber si, dada una expresión booleana con variables y sin cuantificadores, hay alguna asignación de valores para sus variables que hace que la expresión sea verdadera. Por ejemplo, una instancia de SAT sería el saber si existen valores para x_1, x_2, x_3, x_4 tales que la expresión:

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$$
 sea cierta.

Por el contrario, el problema de si la expresión en cuestión adquiere valor falso para todas las combinaciones de sus variables, se denomina UNSAT.

2. Problema de Programación Lineal Entera: En algunos casos se requiere que la solución óptima se componga de valores enteros para algunas de las variables. La resolución de este problema se obtiene analizando las posibles alternativas de valores enteros de esas variables en un entorno alrededor de la solución obtenida considerando las variables reales. Muchas veces la solución del programa lineal truncado está lejos de ser el óptimo entero, por lo que se hace necesario usar algún algoritmo para hallar esta solución de forma exacta. El más famoso es el método de 'Ramificar y Acotar' o Branch and Bound. El método de Ramificar y Acotar parte de la adición de nuevas restricciones para cada variable de decisión (acotar) que al ser evaluado independientemente (ramificar) lleva al óptimo entero.
3. Problema del Conjunto Independiente: En teoría de grafos, un conjunto independiente o estable es un conjunto de vértices en un grafo tal que ninguno de sus vértices es adyacente a otro. Es decir, es un conjunto V de vértices tal que para ningún par de ellos existe alguna arista que los conecte. El tamaño de un conjunto independiente es el número de vértices que contiene. Dado un grafo G , un subconjunto de vértices $H \subset V(G)$ se llama conjunto independiente si no hay dos vértices en H que sean adyacentes.
El número de independencia de G es el tamaño de un conjunto independiente más grande, y se denota por $\alpha(G)$.

6. Pruebe sus conocimientos, incluya al menos uno de los ejercicios y comente sobre la manera en que llegó a la solución

<http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html>

Problema: Es tu cumple y quieres saber si alguien mas en el salon tiene tu edad

```
salonedades=[20,21,22,23,22,20,21,25,24,23,19,18]
edadmia=22
acumulacion=[]
for i in salonedades: #Recorro todas las personas del salon y les pregunto su edad
    #comparo la edad de cada uno con la mia y si es igual, entonces voy acumulando cuantas
    #personas tienen mi edad para saber. Pero debo recorrer todo
    #si el salon es de 10, debo preguntarle a 10 personas
    if i == edadmia:
        acumulacion.append(1)

print('En el salon, ',len(acumulacion),' personas tienen la misma edad que yo')

En el salon,  2  personas tienen la misma edad que yo
```

Es un problema lineal, donde si hay N entradas, va a haber N operaciones

```
In [58]: #vamos a hacer una ventana para un problema de una red neuronal recursiva
#Necesitamos un arreglo de ventanas que se vaya recorriendo n(window) valores
#Ejemplo:
#[1,2,3,4,5] donde la ventana es 3
#necesito[1,2,3] => [2,3,4] => [3,4,5] etc.
window=3
windowarray=[]
array=[1,2,3,4,5]
for i in array:
    auxarray=[]
    for j in range(i,window+i):
        val = j
        auxarray.append(val)
    windowarray.append(auxarray)
print(windowarray)

[[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], [5, 6, 7]]
```

Es un problema cuadrático, donde N son los valores de entrada y $N*N$ van a ser la salida máxima, dependiendo el caso se puede romper antes la iteración.

7. Conclusiones

La complejidad es un factor importante a tener en cuenta a la hora de diseñar un algoritmo, dado que definirá el tiempo que este tarde en ejecutar y la cantidad de recursos empleados de la computadora para resolverlo, como se corrobora en en este proyecto, a mayor cantidad de entrada mas se esfuerza la computadora en poder ejecutar un algoritmo, señalando que a partir de cien mil entradas 16GB de RAM empiezan a notarse superadas por lo que el tiempo en algunos de estos casos también se ve incrementado. Y con un millón de datos de entrada no se pudieron ejecutar dado que la computadora dejó de funcionar correctamente.

Santiago Rivera Gonzalez

Constante :

```
[3]: time=0
start=0
end=0
def Oconstante(array):
    tamaño=0
    tamaño=len(array)
    start=timer()
    val = array[tamaño - 1]
    end=timer()
    #El resultado final va a tener valor del ultimo numero que exista en el array
    print("Ultimo valor: ",val)
    print("Tamano del valor: ", 1)#No hay iteraciones, solo es un numero constante de 1
    time = end-start
    print('tiempo de ejecucion: ',time)
    plt.axhline(y=val, color='r', linestyle='-')
    plt.show()
    return start,end

start,end=Oconstante(array);
#EN este caso solo hay una constante que es sacar un dato del array
#Si la entrada de tu funcion es n, La salida va a ser una constante
```

Lineal:

```
In [5]: buscar = random.randint(1, 25)
iteraciones=[]
cont=0
start=0
end=0
print('F(x): antes del 0: ',len(array))
def Olineal(a,cont):
    for i in a:
        start=timer()
        if buscar == i:
            b=True
            break
        else:
            b=False
        cont+=1
        iteraciones.append(cont)
    if b==True:
        print('Existe el dato: ', buscar, ' en el arreglo ')
    else:
        print('NO existe el dato:' , buscar,' en el arreglo')
    end=timer()
    time=end-start
```



```
File Edit View Insert Cell Kernel Widgets Help Trusted
iteraciones.append(cont)
if b==True:
    print('Existe el dato: ', buscar, ' en el arreglo ')
else:
    print('NO existe el dato:' , buscar,' en el arreglo')
end=timer()
time=end-start
plt.title('Operaciones vs datos',color='white')
plt.xlabel('Datos',color='red')
plt.ylabel('Iteraciones',color='darkblue')
plt.plot(iteraciones,color='darkblue')
plt.scatter(a,np.zeros(len(a)),color='red')
return start,end
#En este caso si tienes n entradas, va a hacer n iteraciones
#solo hay un for loop y por ello sus datos de entrada son iguales a los de salida
#No es necesario que complete todo el recorrido, en este caso hace una búsqueda donde las i
#son n = n si esta hasta el final del arreglo, si logra encontrarlo antes regresara
#Si tampoco logra encontrar el dato, pues recorrera todos los elemtnos por igual
start,end=Olineal(array,cont)
print('F(x): despues del 0: ',len(iteraciones))
time = end-start
print(time)
```

Cuadrática:

```

File Edit View Insert Cell Kernel Widgets Help

cont =0
print("F(x): antes del algoritmo ",len(array))
mult=0
def OCuadratica(a,iteraciones,cont):
    for i in a:
        start=timer()
        for j in a:
            mult=j*i
            cont+=1
            iteraciones.append(cont)
        end =timer()
    plt.title('Operaciones vs datos',color='white')
    plt.xlabel('Datos',color='red')
    plt.ylabel('Iteraciones',color='darkblue')
    plt.plot(iteraciones,color='darkblue')
    plt.scatter(a,np.zeros(len(a)),color='red')
    return start,end

start,end=OCuadratica(array,iteraciones,cont)
print("F(x) despues del algoritmo",len(iteraciones))
#Tiene complejidad cuadratica, puesto que si tenemos n elementos tendremos que hacer n^2 comparaciones
#[1,2,3]=> multiplicas

```

```

plt.xlabel('Datos',color='red')
plt.ylabel('Iteraciones',color='darkblue')
plt.plot(iteraciones,color='darkblue')
plt.scatter(a,np.zeros(len(a)),color='red')
return start,end

start,end=OCuadratica(array,iteraciones,cont)
print("F(x) despues del algoritmo",len(iteraciones))
#Tiene complejidad cuadratica, puesto que si tenemos n elementos tendremos que hacer n^2 comparaciones
#[1,2,3]=> multiplicas
#iteracion 1 => (1x1)(1x2)(1x3)
#iteracion 2=>(2x1)(2x2)(2x3)
#iteracion 3=>(3x1)(3x2)(3x3)
#El algoritmo podria poner un break antes de que se complete como una busqueda, pero en caso de no encontrar
#seria lo mismo
time = end-start

```

2

```
In [58]: #vamos a hacer una ventana para un problema de una red neuronal recursiva
#Necesitamos un arreglo de ventanas que se vaya recorriendo n(window) valores
#Ejemplo:
#[1,2,3,4,5] donde la ventana es 3
#necesito[1,2,3] => [2,3,4] => [3,4,5] etc.
window=3
windowarray=[]
array=[1,2,3,4,5]
for i in array:
    auxarray=[]
    for j in range(i,window+i):
        val = j
        auxarray.append(val)
    windowarray.append(auxarray)
print(windowarray)

[[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], [5, 6, 7]]
```

Logarítmica:

```
Complejidad Logarítmica  $O(\log(n))$ 

In [42]: cont=0
iteraciones=[]
start=timer()
def binary_search(arr, low, high, x,cont,itera):

    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # Si el elemento a buscar esta en medio
        if arr[mid] == x:
            cont+=1
            itera.append(cont)
            return mid,itera

        # Si el elemento es mas chico que en medio, voy a a izquierda
        elif arr[mid] > x:
```

```
File Edit View Insert Cell Kernel Widgets Help

# Si el elemento es mas chico que en medio, voy a a izquierda
elif arr[mid] > x:
    cont+=1
    itera.append(cont)
    return binary_search(arr, low, mid - 1, x,cont,itera)

# Si el elemento es mas grande que en medio, voy a la derecha
else:
    cont+=1
    itera.append(cont)
    return binary_search(arr, mid + 1, high, x,cont,itera)

else:
    # Si el elemento no existe en el array
    cont+=1
    itera.append(cont)
    return -1,itera

result,iteraciones = binary_search(array, 0, len(array)-1, 25,cont,iteraciones)
end=timer()
if result!= -1:
    print('Existe el numero en el array')
```

```
else:
    # Si el elemento no existe en el array
    cont+=1
    itera.append(cont)
    return -1,itera

result,iteraciones = binary_search(array, 0, len(array)-1, 25,cont,iteraciones)
end=timer()
if result!= -1:
    print('Existe el numero en el array')
else:
    print('No existe el numero en el array')

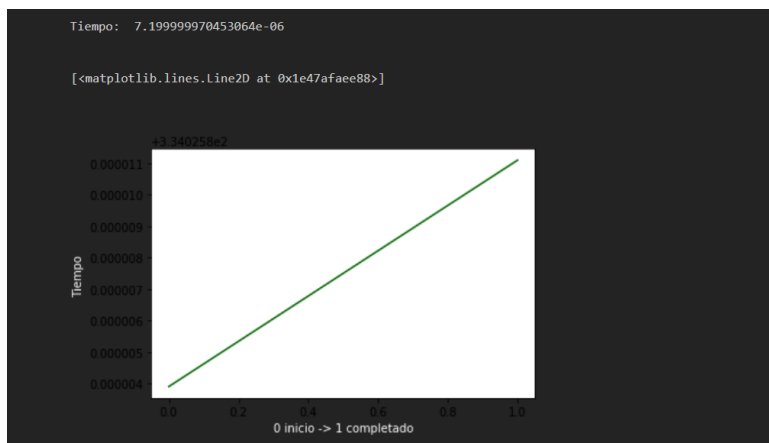
time=end-start
print(time)

#ejemplo: Si tenemos 10 datos, nos debería tomar 1 segundo realizarlo Log(10)
#Si tenemos 100 datos, nos debería tomar 2 segundos Log(100)
```

Santiago Rivera Gonzalez

Tablas con 10 datos:

Constante:



Lineal:

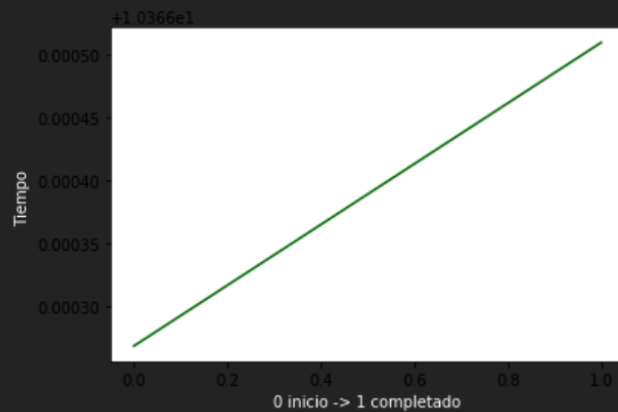
Santiago Rivera Gonzalez

```
F(x): antes del 0: 4  
NO existe el dato: 2 en el arreglo  
F(x): despues del 0: 4  
0.00024100000000008239
```



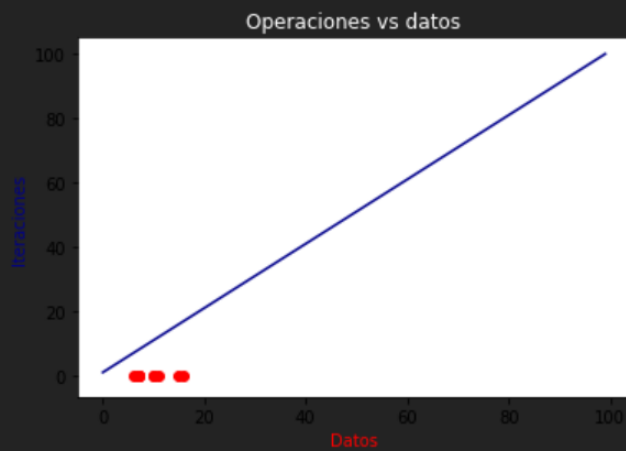
Tiempo: 0.00024100000000008239

[<matplotlib.lines.Line2D at 0x15e7b6de088>]



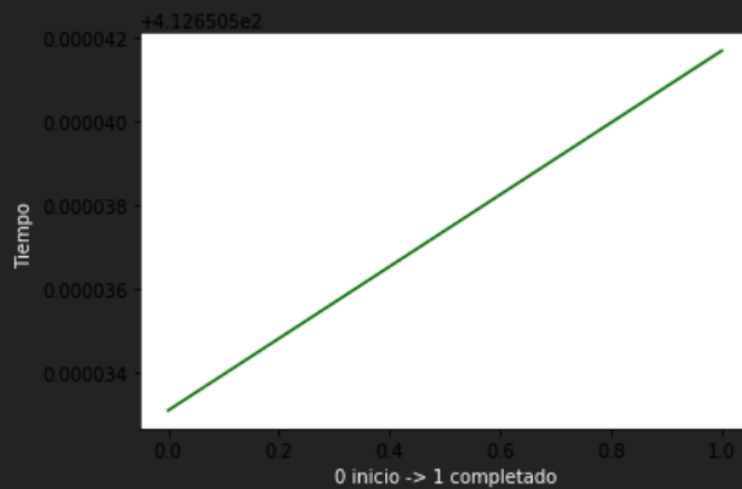
Cuadrático:

```
F(x): antes del algoritmo 10  
F(x) despues del algoritmo 100
```



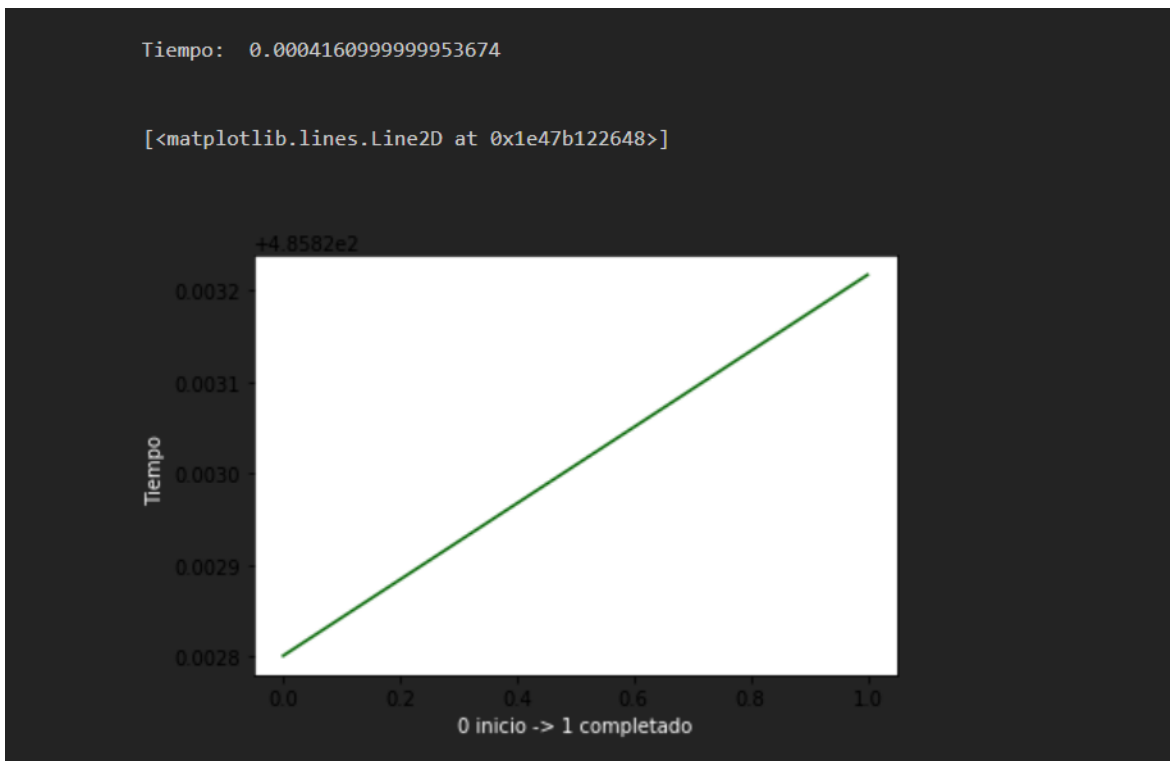
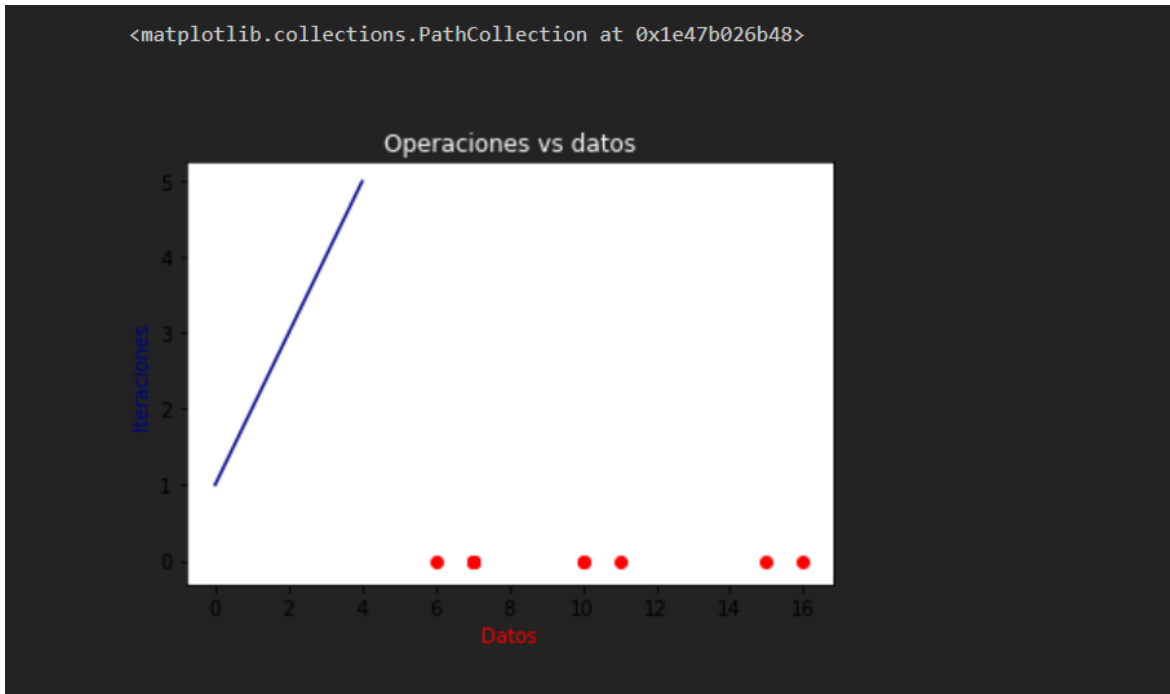
```
Tiempo: 8.600000001024455e-06
```

```
[<matplotlib.lines.Line2D at 0x1e47b0e98c8>]
```



Santiago Rivera Gonzalez

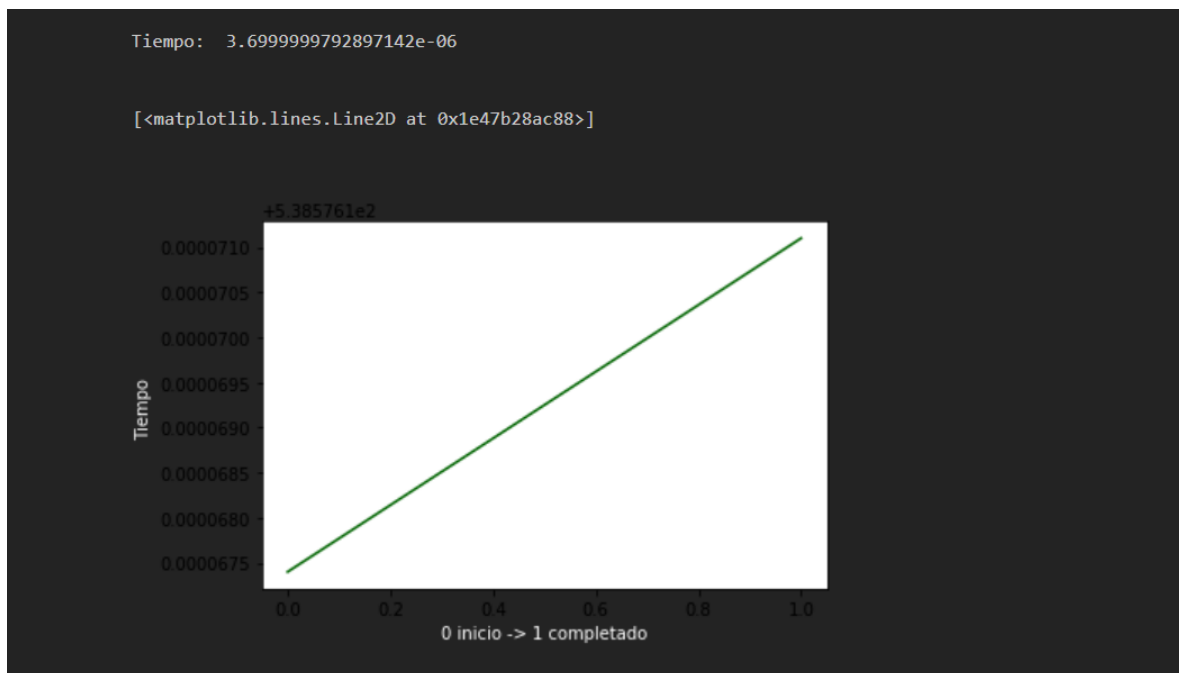
Logarítmico:



Santiago Rivera Gonzalez

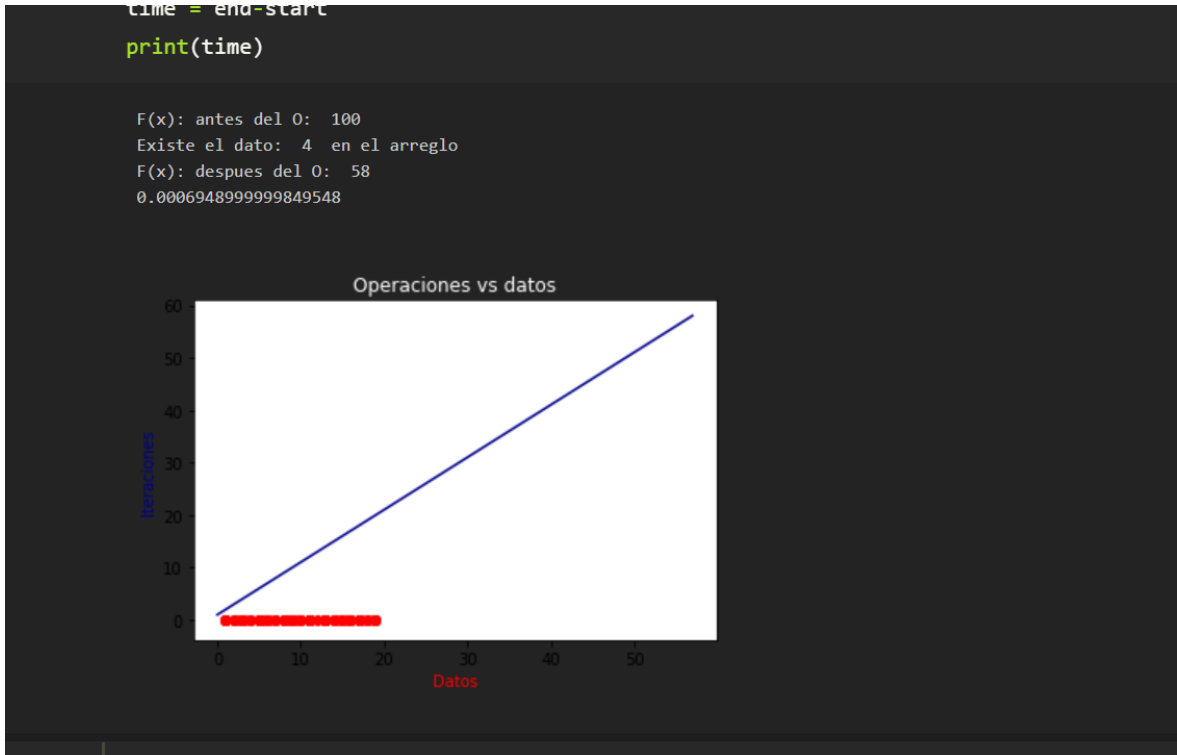
Tablas con 100 datos:

Constante:

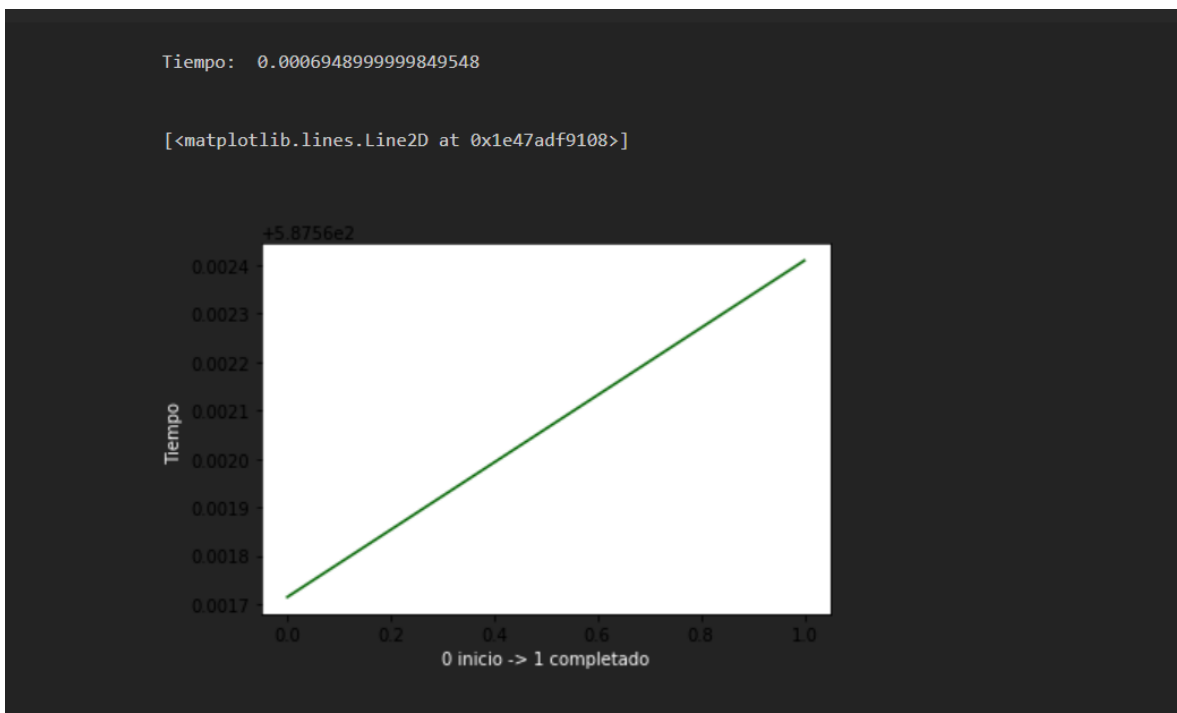


Santiago Rivera Gonzalez

Lineal:

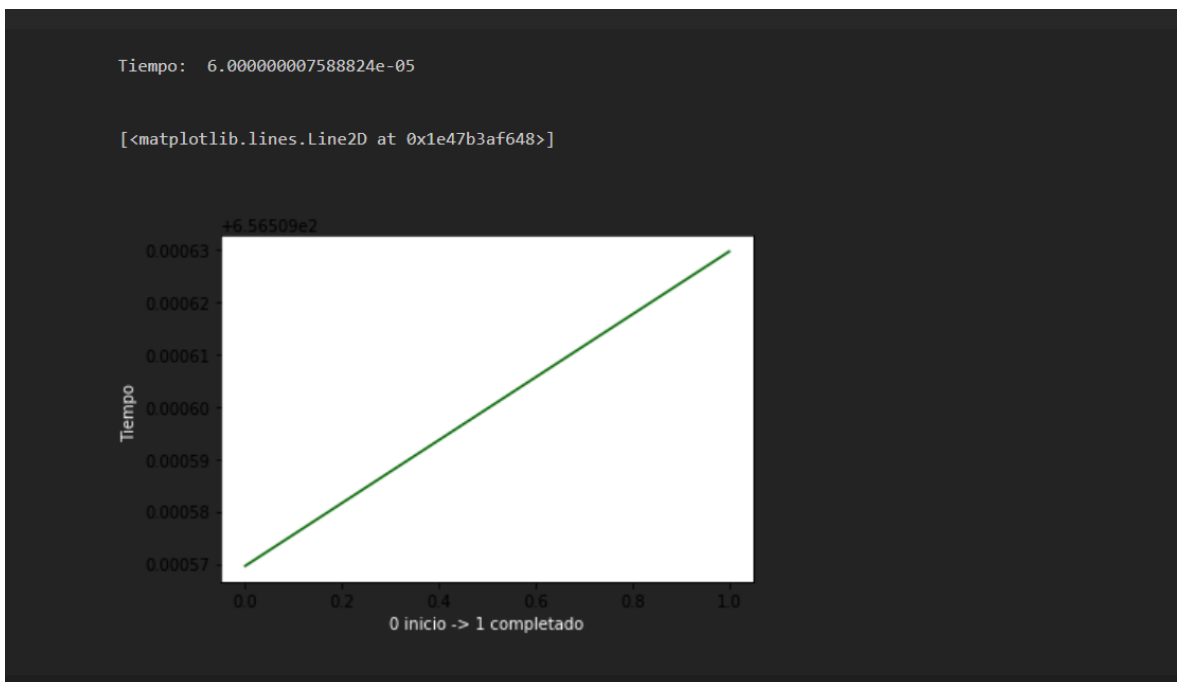
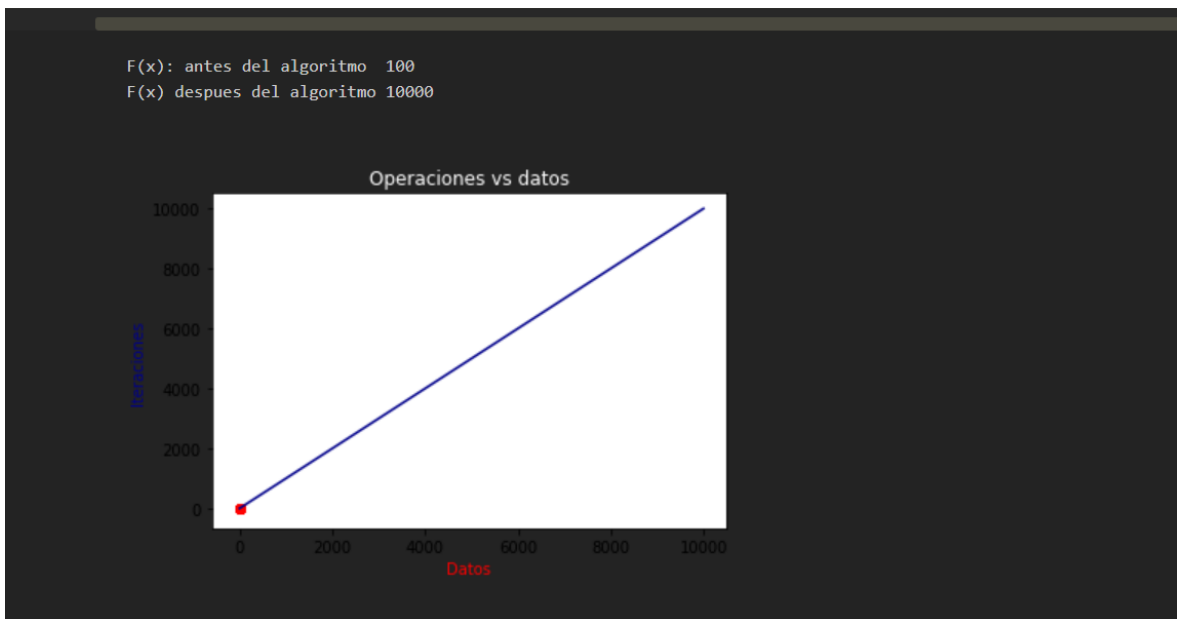


Encontró antes el número a buscar



Santiago Rivera Gonzalez

Cuadrático:



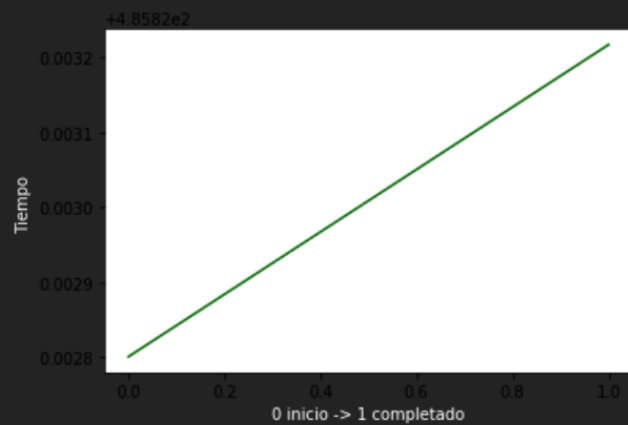
Logarítmico:

```
<matplotlib.collections.PathCollection at 0x1e47b354fc8>
```



Tiempo: 0.0004160999999953674

```
[<matplotlib.lines.Line2D at 0x1e47b122648>]
```

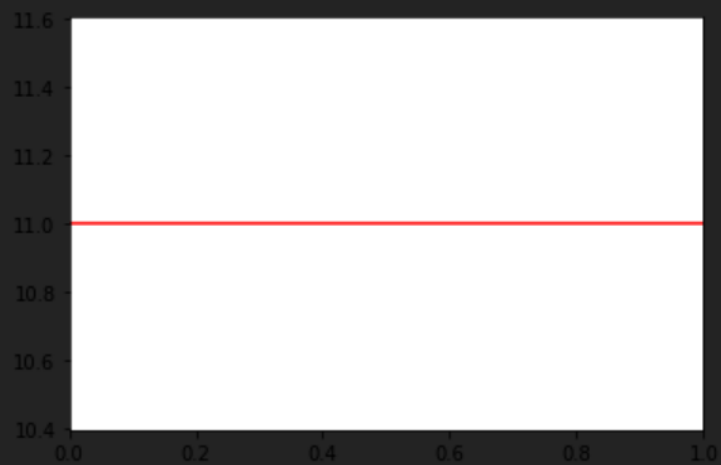


Tablas con 1000 datos:

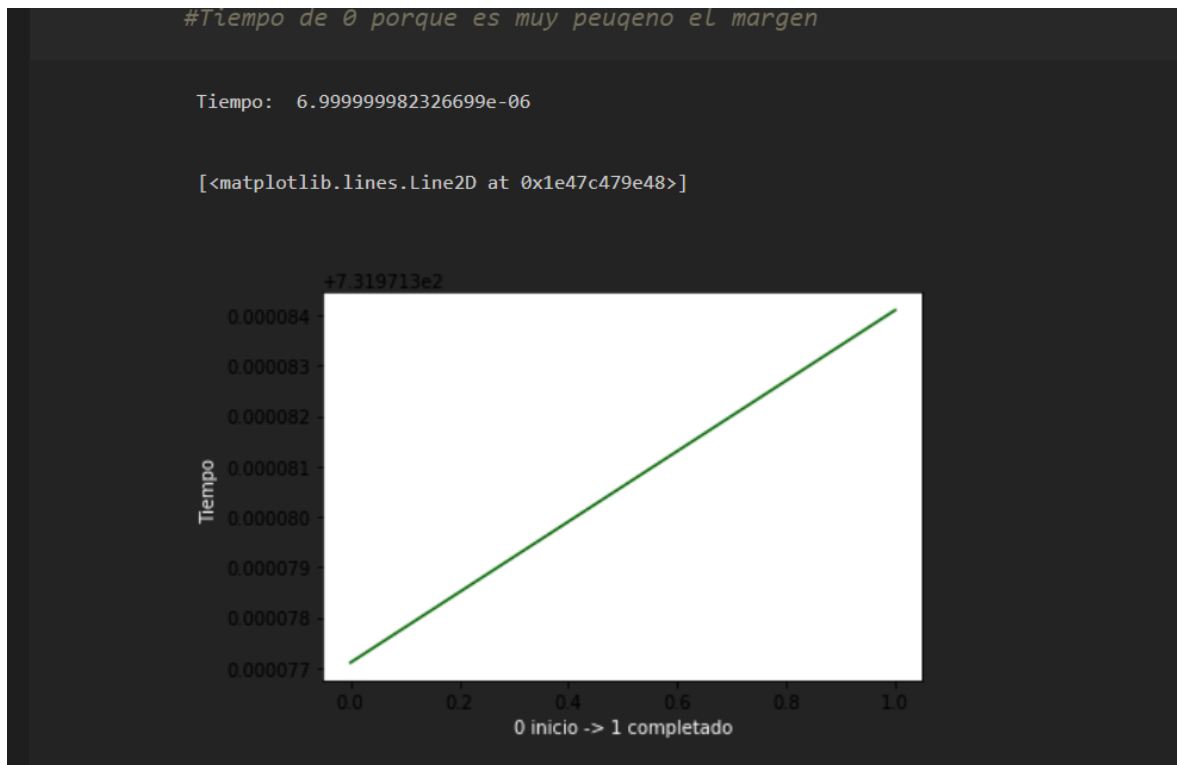
Constante:

```
#si quiero la posicion 5, voy a array[pos5]  
time = end-start
```

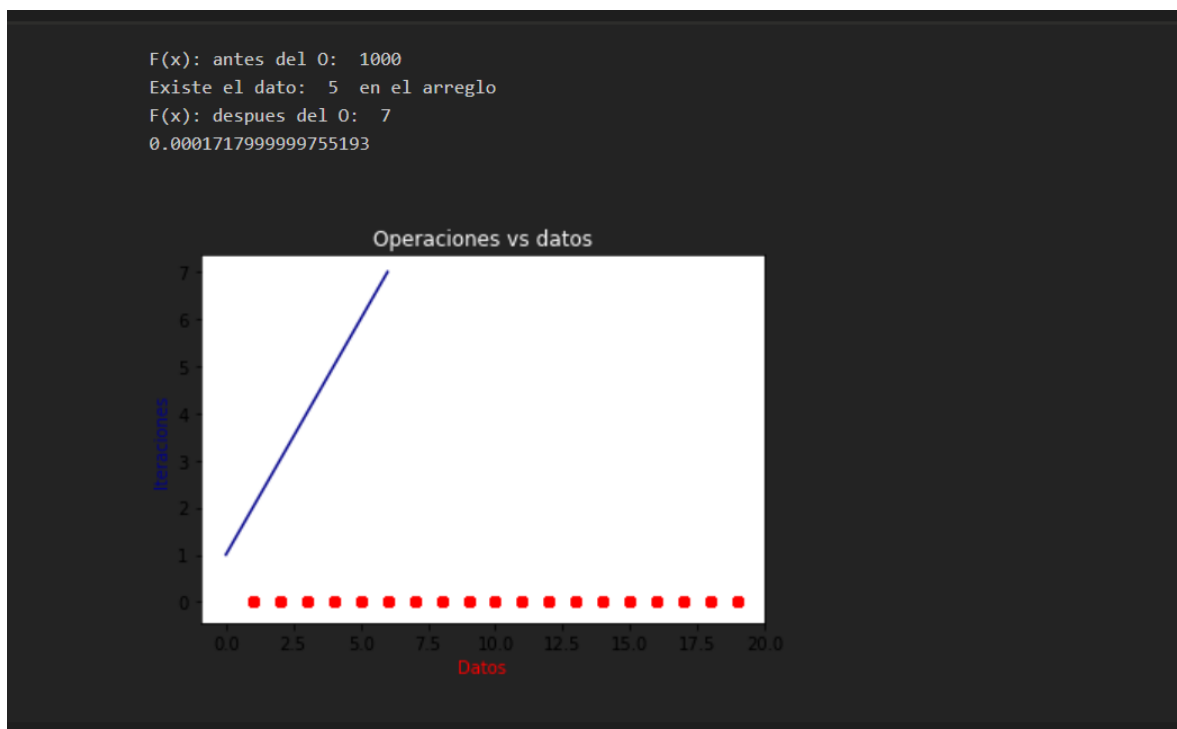
```
Ultimo valor: 11  
Tamano del valor: 1  
tiempo de ejecucion: 6.999999982326699e-06
```

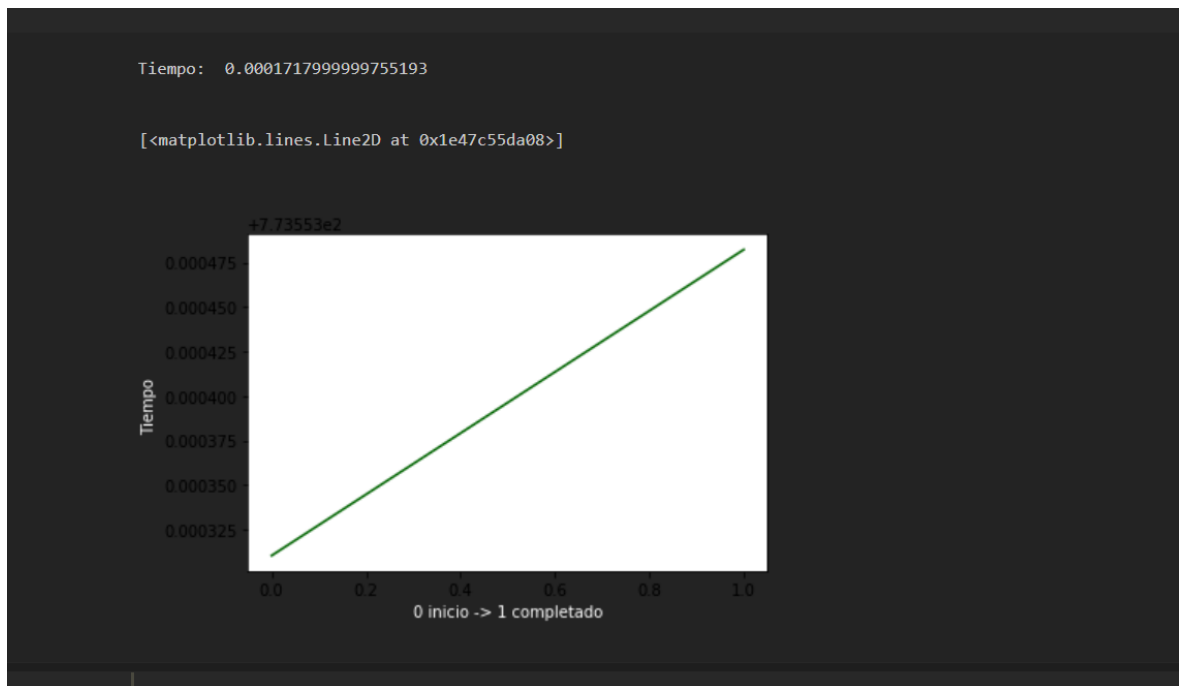


Santiago Rivera Gonzalez

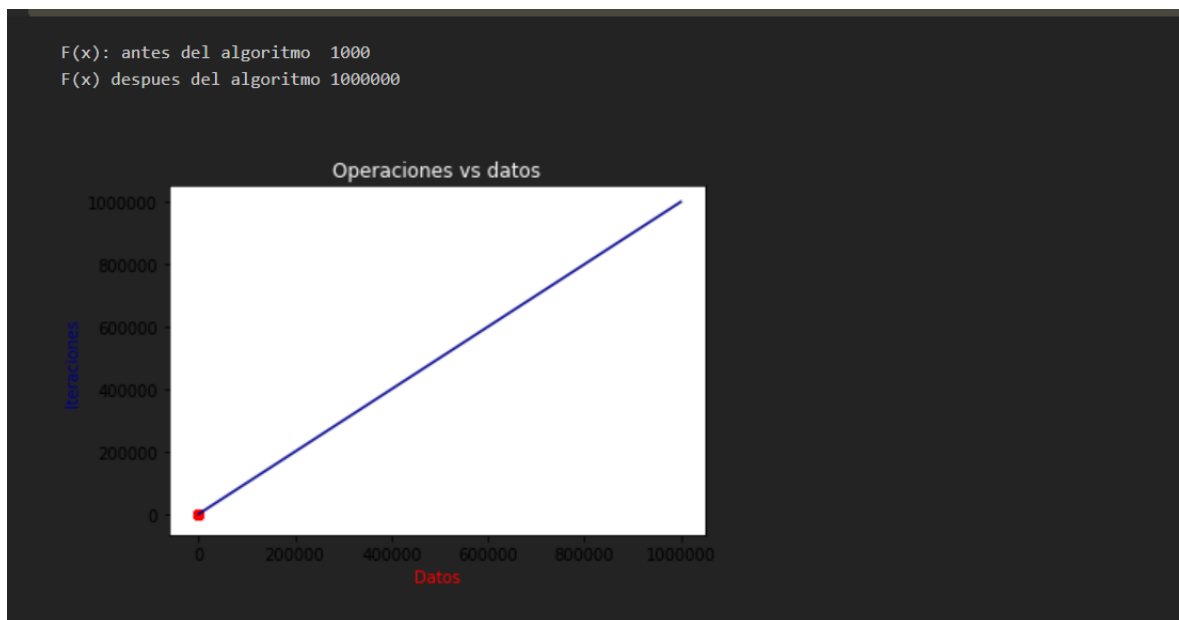


Lineal:





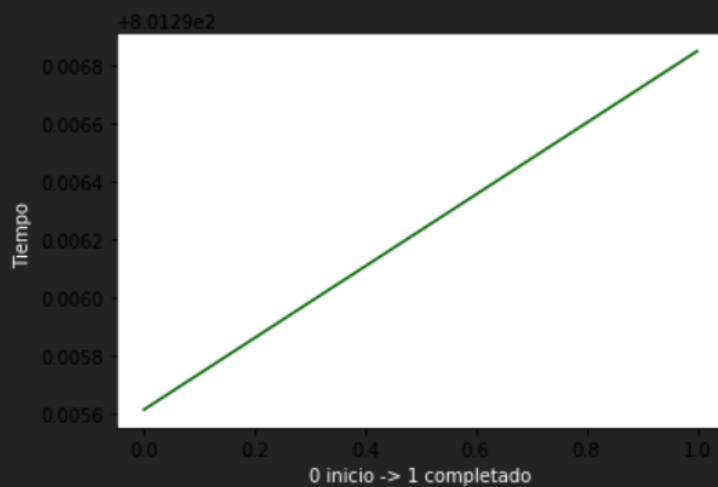
Cuadrático:



Santiago Rivera Gonzalez

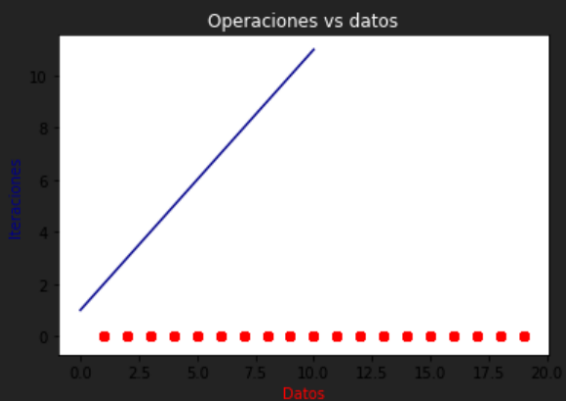
Tiempo: 0.001237400000363594

[<matplotlib.lines.Line2D at 0x1e47c42ae08>]

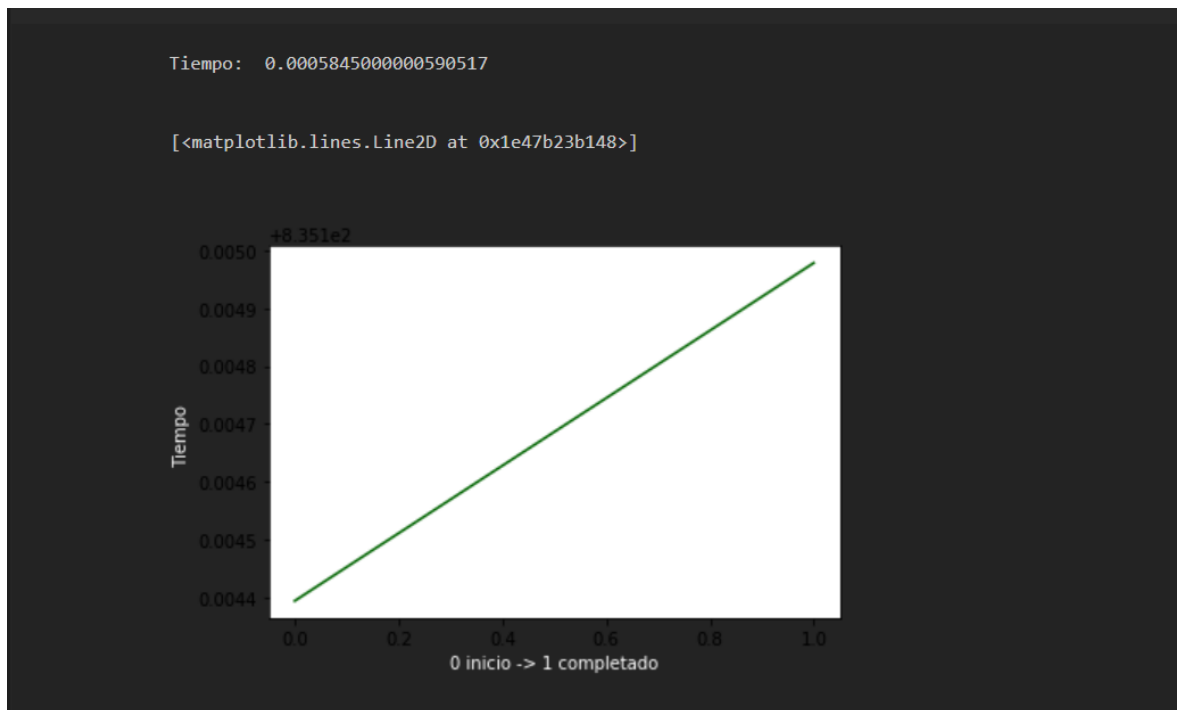


Logarítmico:

<matplotlib.collections.PathCollection at 0x1e47c51a308>

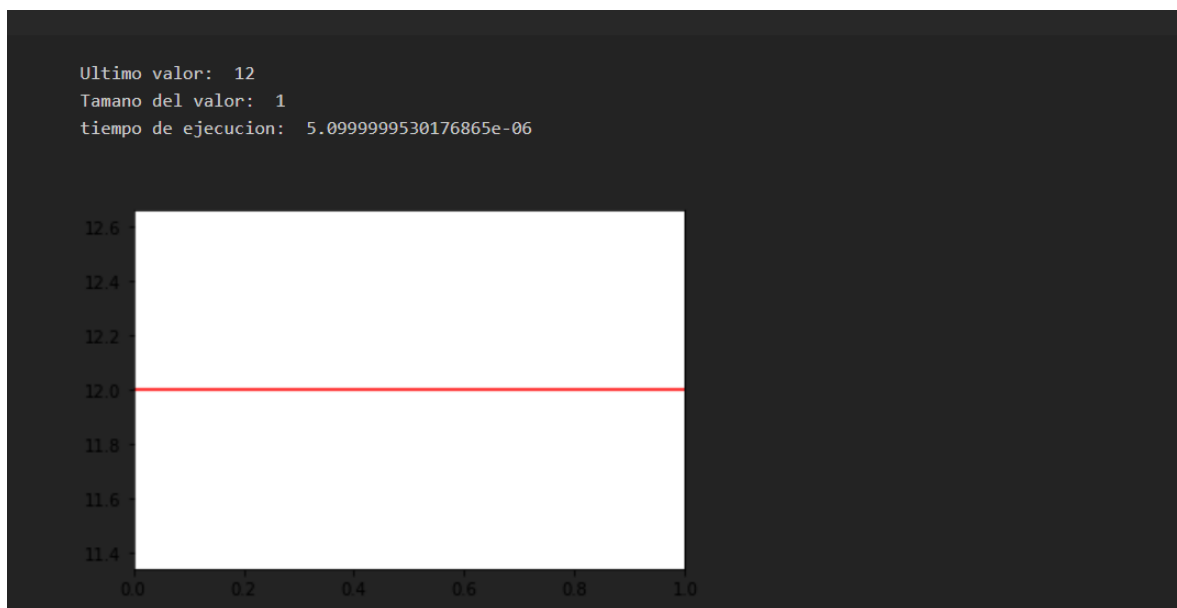


Santiago Rivera Gonzalez



Tablas con 10,000 datos:

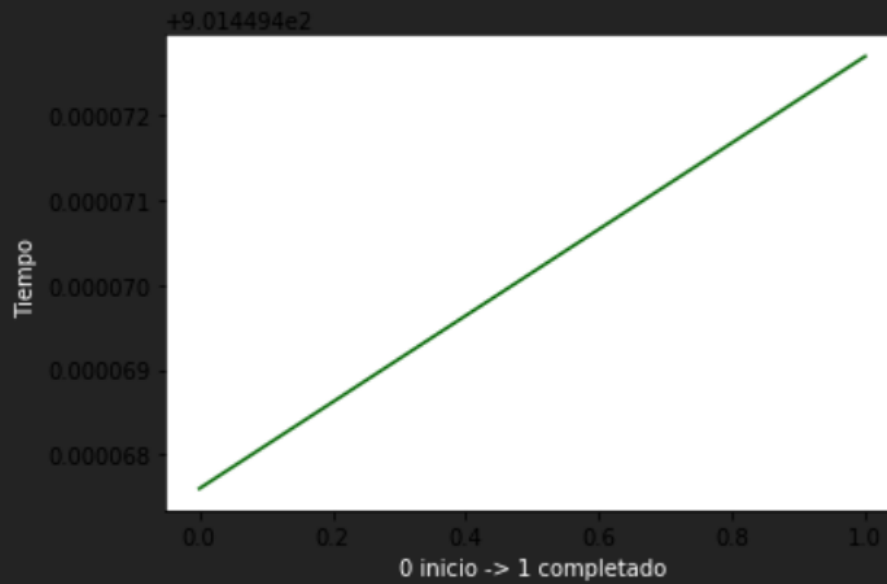
Constante:



Santiago Rivera Gonzalez

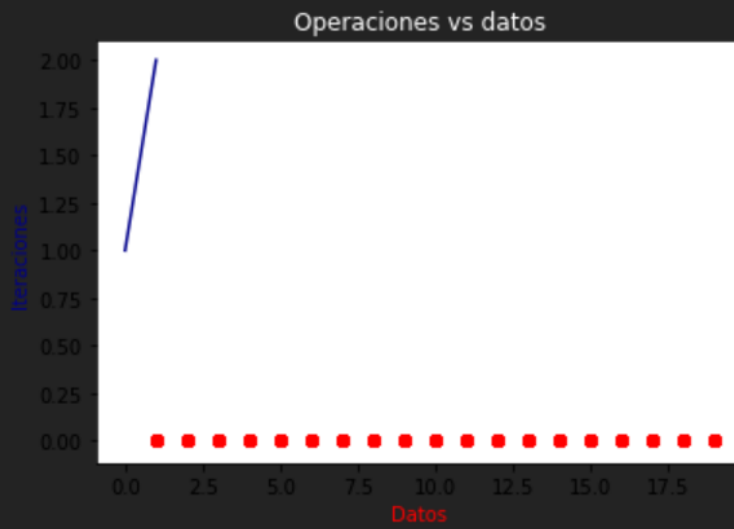
```
Tiempo: 5.0999999530176865e-06
```

```
[<matplotlib.lines.Line2D at 0x1e418fbbcc8>]
```



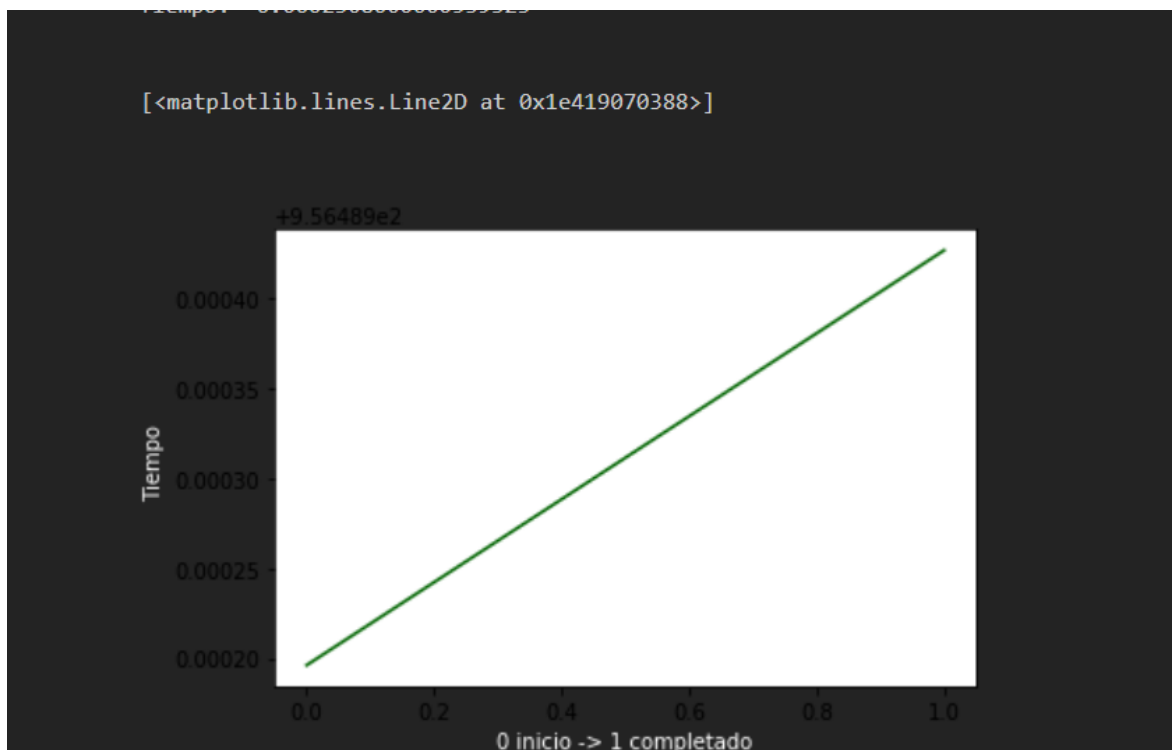
Lineal:

```
F(x): antes del 0: 10000  
Existe el dato: 6 en el arreglo  
F(x): despues del 0: 2  
0.0002308000000539323
```



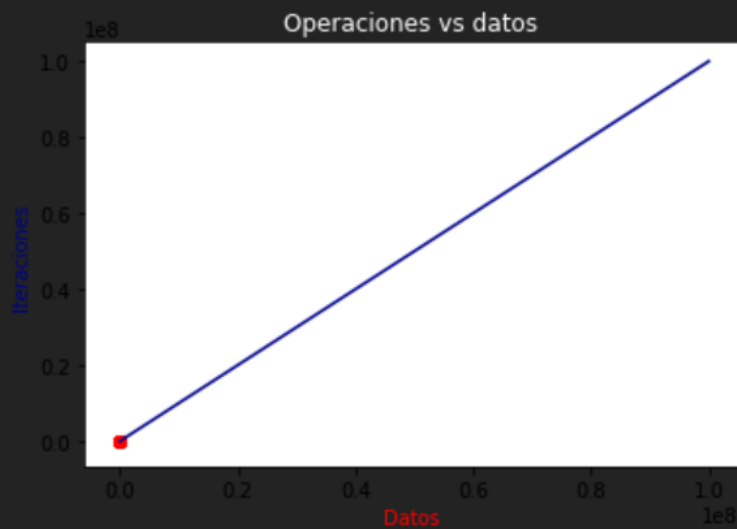
Algoritmo que salió rápido puesto que encontró el número a inicios del arreglo

Santiago Rivera Gonzalez



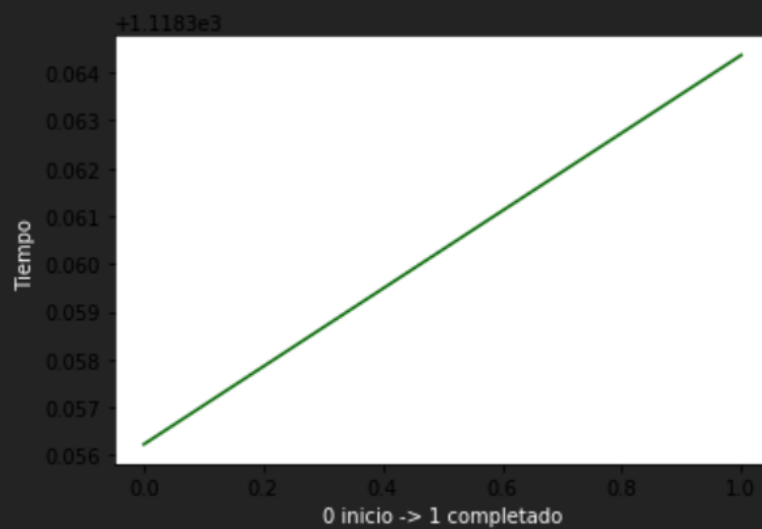
Cuadrático:

```
F(x): antes del algoritmo 10000  
F(x) despues del algoritmo 100000000
```



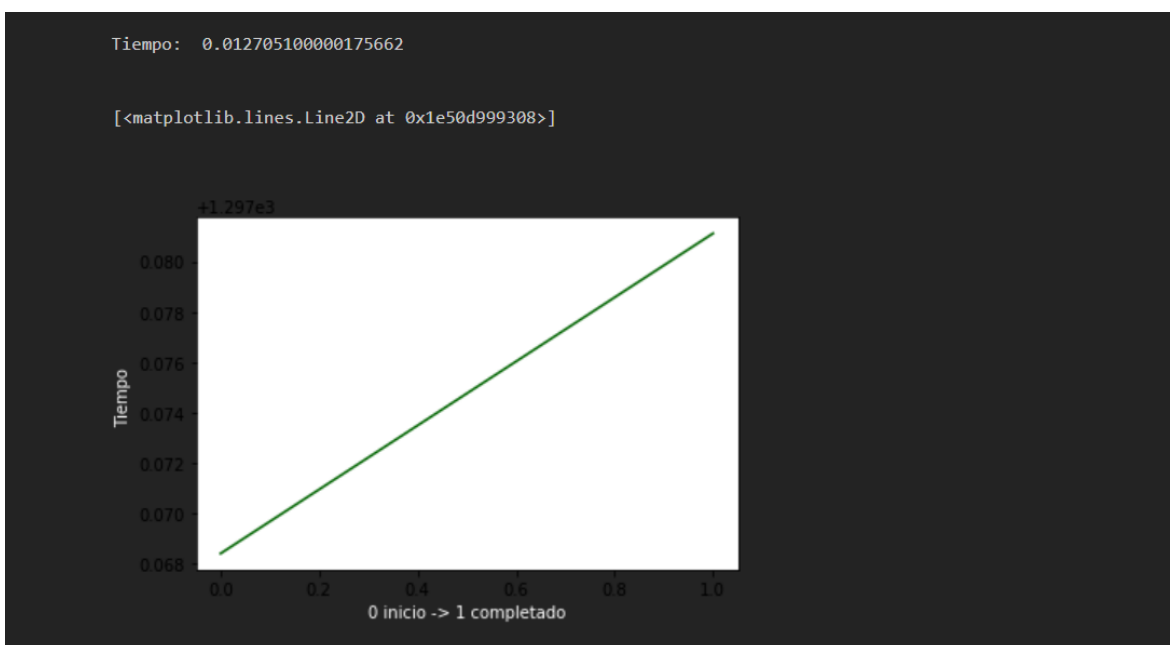
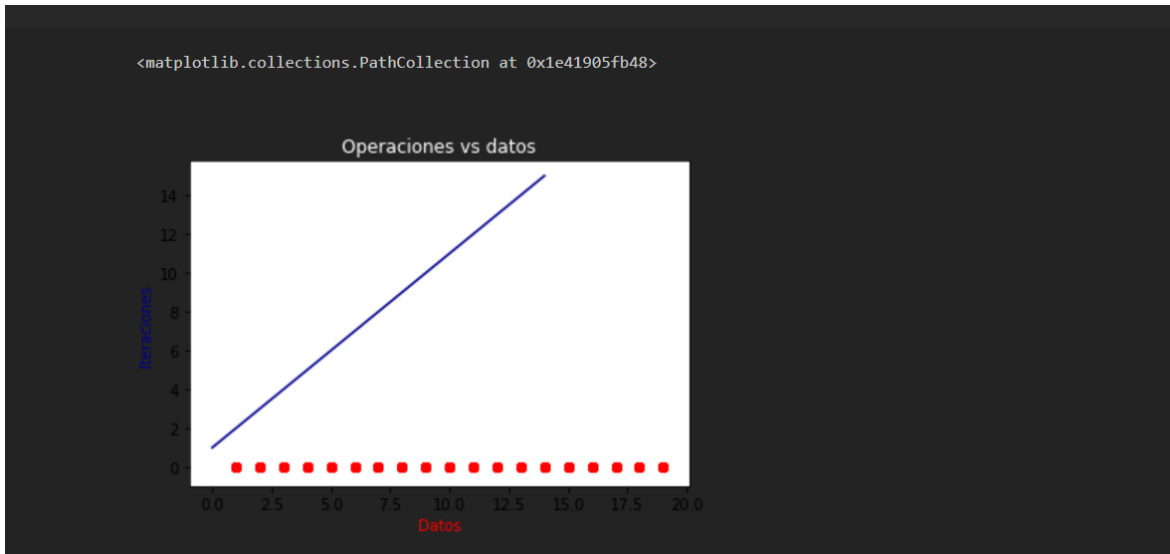
```
Tiempo: 0.00814170000012382
```

```
[<matplotlib.lines.Line2D at 0x1e47b2dd648>]
```



Santiago Rivera Gonzalez

Logarítmico:



Santiago Rivera Gonzalez

Tablas con 100,000 datos:

Constante:

```
F(x): antes del algoritmo 100000

-----
MemoryError                                Traceback (most recent call last)
<ipython-input-7-d9d300a57920> in <module>
    18     return start,end
    19
---> 20 start,end=OCuadratica(array,iteraciones,cont)
    21 print("F(x) despues del algoritmo",len(iteraciones))
    22 #Tiene complejidad cuadratica, puesto que si tenemos n elementos tendremos que hacer n^2 comparaciones

<ipython-input-7-d9d300a57920> in OCuadratica(a, iteraciones, cont)
     9         mult=j*i
    10         cont+=1
---> 11         iteraciones.append(cont)
    12     end =timer()
    13     plt.title('Operaciones vs datos',color='white')

MemoryError:
```

Lineal:

Cuadrático:

Logarítmico:

Tablas con 1,000,000 datos:

Constante:

Lineal:

Cuadrático:

Logarítmico:

Con cien mil y un millón de datos nuestras computadoras dejaron de funcionar y no pudimos ejecutar los algoritmos por un memory error, que es que no tenía suficiente memoria para procesar todos los datos.

Referencias

1. <https://www.lnds.net/blog/2013/11/la-notacion-big-o.html>
2. Kenneth H. Rosen (2012). Discrete Mathematics and Its Applications. Mc Graw Hill. Seventh Edition
3. Deitel & Deitel (2015). How to Program. Pearson Tenth Edition.
4. <https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/>
5. <https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation>
6. <https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>
7. <http://www.lcc.uma.es/~av/Libro/CAP1.pdf>
8. <https://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-28.html#principales-%C3%B3rdenes-de-complejidad>