EJERCICIO PRÁCTICO INTRODUCTORIO

· HITOS 1 Y 2 ·

22/02/2021 · UC3M · GRUPO 50

Hugo Romero 100 386 322

Irene Fernández 100 386 294

Maribel Medrano 100 386 149

Contenido

HITO 1

ALGORITMO 0

En primer lugar, se pedía un estudio de un algoritmo de primalidad lo más básico posible, con el fin de evaluar su rendimiento y estudiar su complejidad computacional.

ALGORITMO DISEÑADO

Para evaluar si un número es primo o compuesto se ha diseñado, en primer lugar, un algoritmo sin optimizaciones que consiste en comprobar para cada uno de los números evaluados, si es divisible entre todos los números entre el 2 y el número a evaluar menos 1. En caso de que no sea divisible entre alguno de estos números, el número evaluado es primo, mientras que si es divisible entre alguno de ellos el número evaluado es compuesto.

Para este caso, al no incluir ninguna optimización, se hará la comprobación para cada uno de los números dentro del intervalo [2, número a evaluar -1], de forma que el coste computacional no se deberá a si un número es primo o no.

El algoritmo diseñado sin optimizaciones es el siguiente:

```
TEST DE PRIMALIDAD (NUMERO_A_EVALUAR: Entero positivo)
Establecer ES_PRIMO a Verdadero
Para cada número CANDIDATO_A_DIVISOR entre 2 y NUMERO_A_EVALUAR – 1:
      Si CANDIDATO_A_DIVISOR es divisor de NUMERO A EVALUAR:
            Establecer ES_PRIMO a Falso
Fin del bucle
Devolver el valor de ES_PRIMO
```

Su implementación en código Java queda de la siguiente manera:

```java
public static boolean primalityTest(BigInteger test) {
	boolean isPrime = true;
	for (BigInteger bi = BigInteger.valueOf(2); bi.compareTo(test) < 0; bi  =
bi.add(BigInteger.ONE)) {
		if (test.mod(bi).equals(BigInteger.ZERO)) {
				isPrime = false;
		}
	}
	return isPrime;
}
```

Este código ha sido verificado antes de realizar las comprobaciones, constatando que identifica a los números primos correctamente.

PRUEBAS EXPERIMENTALES REALIZADAS

El algoritmo se ha probado y medido pasándole números aleatorios de diferentes tamaños, entendiendo por tamaño la cantidad de dígitos que tiene el número.

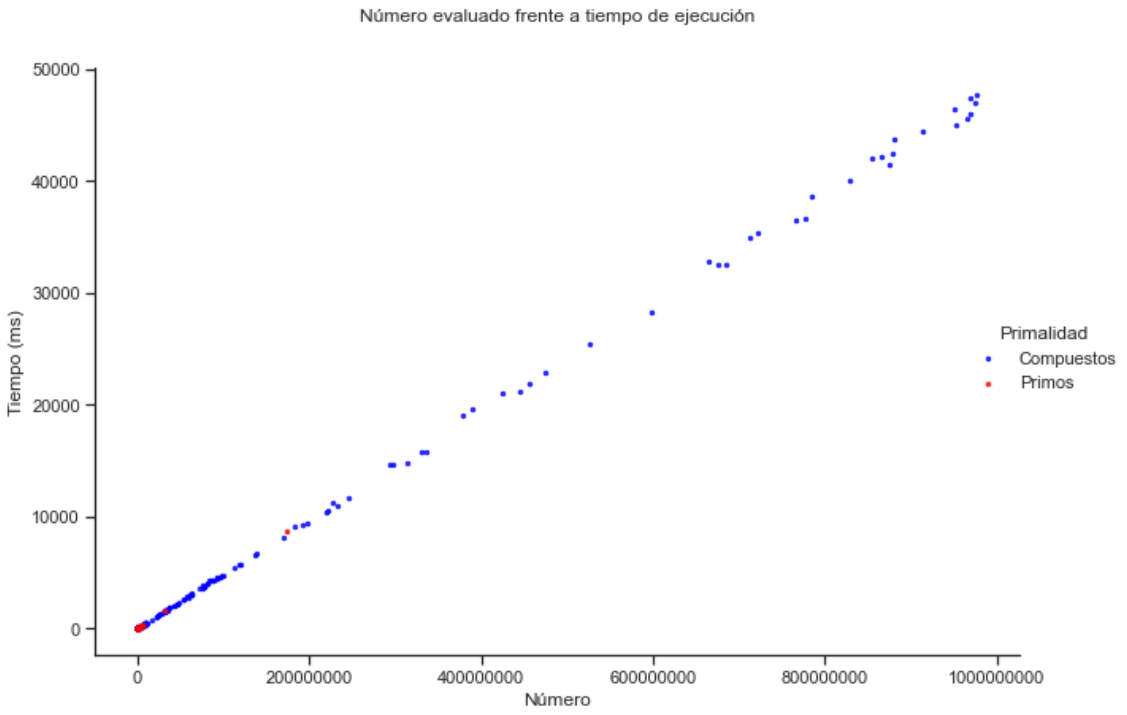
Además, dado que en general esta familia de algoritmos relacionados con la primalidad suele tener por peor caso cuando se le pasa un número primo, se ha duplicado el experimento forzando la generación de números aleatorios a que estos sean primos.

La generación ha consistido en 50 números de cada tamaño para ambos casos, tanto el de números aleatorios como el de primos aleatorios. Los tamaños son los del rango entre 3 dígitos y 9 dígitos, ambos incluidos. El límite inferior ha sido 3 porque cualquier número de 1 y 2 cifras arrojaría tiempo de evaluación 0, por lo que no aportarían demasiada información a la investigación. Por otro lado, el límite superior es 9 dígitos, porque los números por encima de 10^{10} tienen un coste de computación demasiado elevado y no se puede elevar el tiempo de ejecución al orden de días.

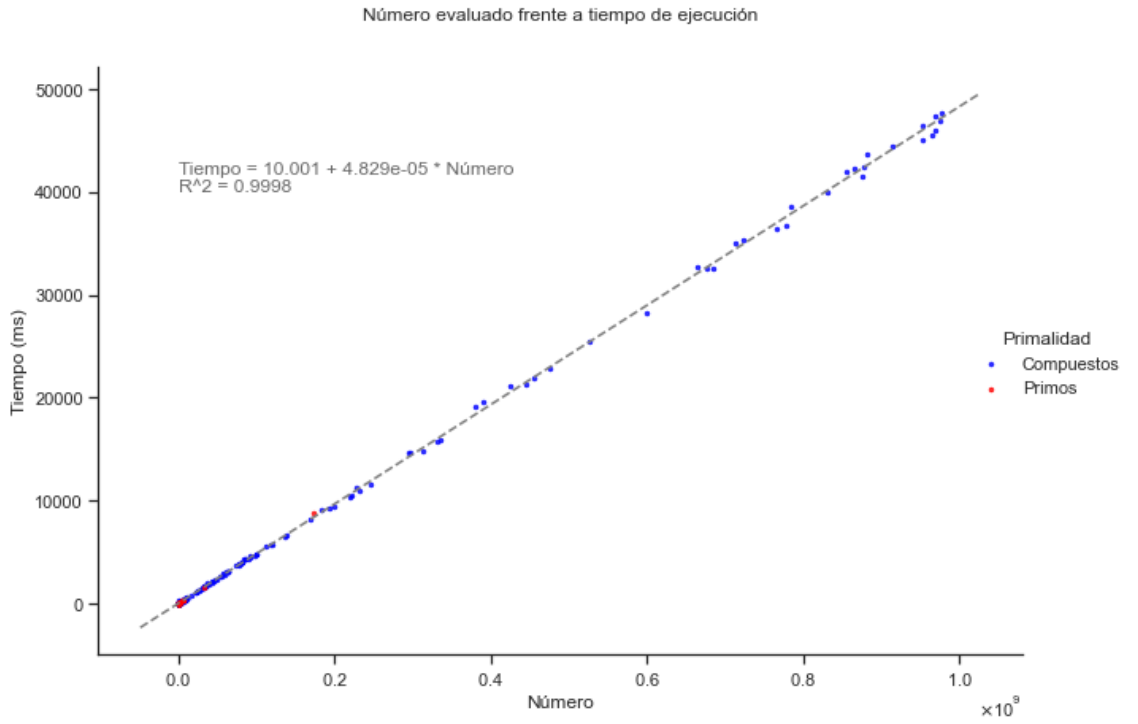
Se han pasado por tanto 350 números al algoritmo en cada experimento, estos se han medido y posteriormente se ha hecho un estudio empírico mediante el uso de programas de autoría propia hechos en Python.

RELACIÓN TIEMPO Y NÚMERO EVALUADO

La siguiente gráfica muestra relación entre el número evaluado y el tiempo que ha tardado el algoritmo en establecer su primalidad, en azul se representan los números que han sido evaluados como complejos y en rojo los primos. Ambos ejes siguen escalas lineales.

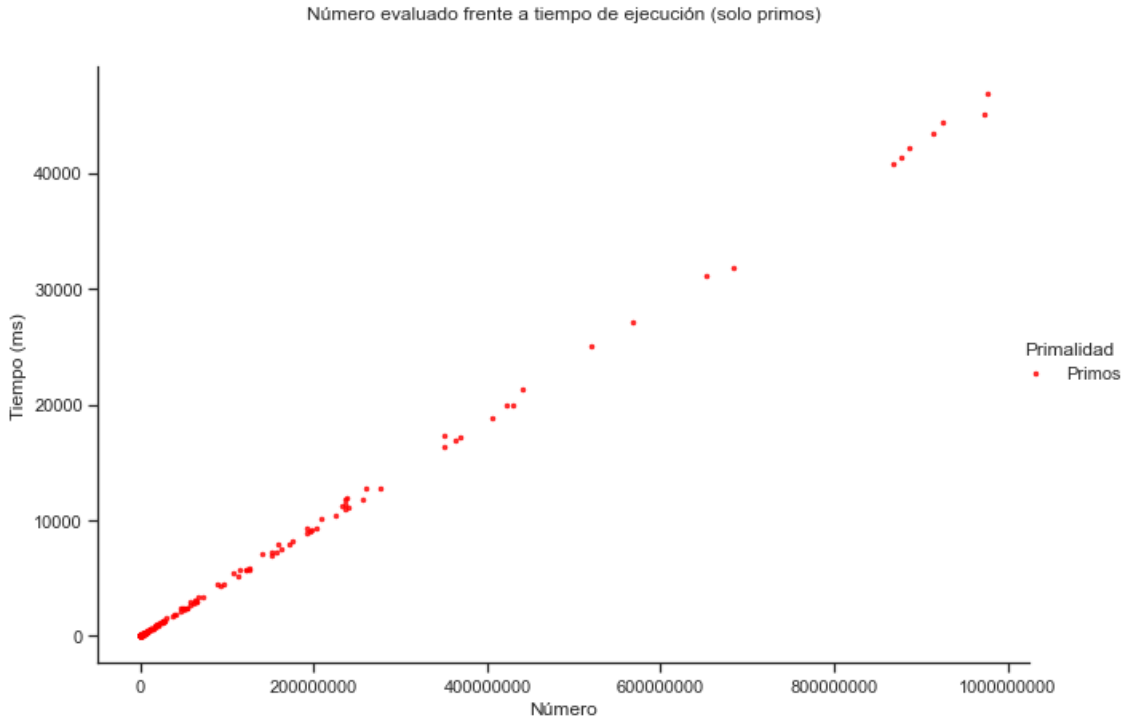


Se puede observar que el ajuste es claramente lineal, de hecho, se puede ir un paso más y tratar de sacar la ecuación de regresión junto con el R^2.

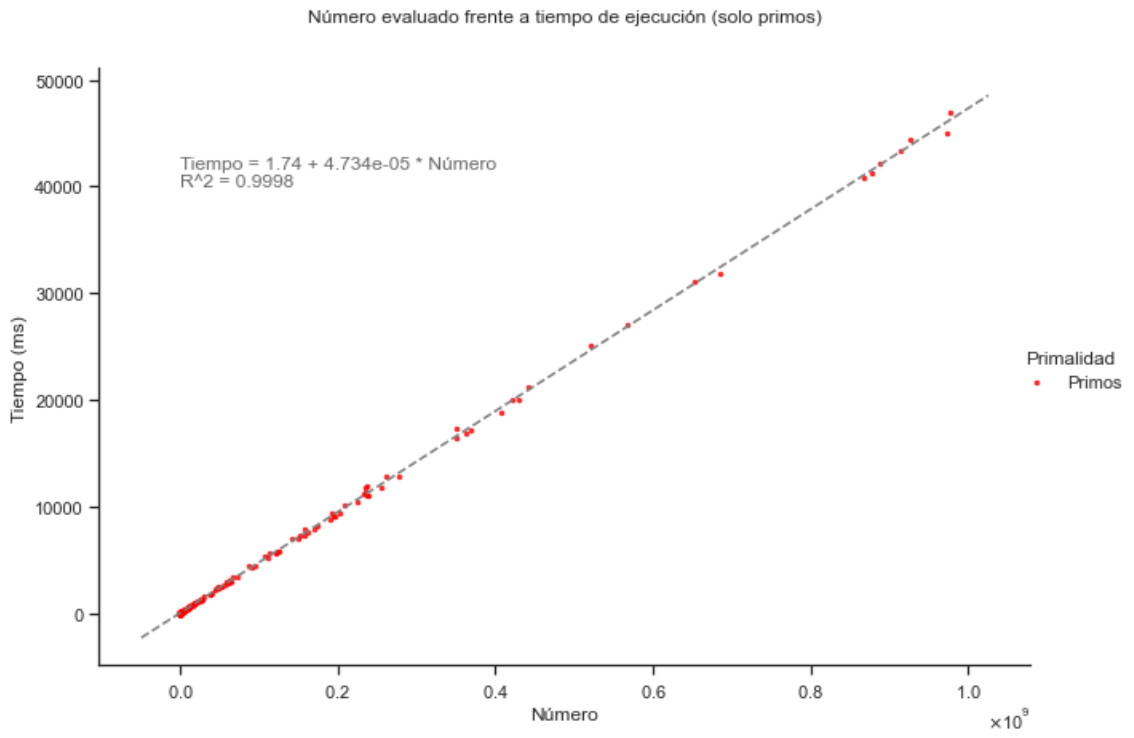


Se puede observar que la fiabilidad del ajuste, el coeficiente R^2, es extremadamente cercano a 1, por lo que el ajuste es casi perfecto mediante la ecuación lineal. De esta forma concluimos así, de forma empírica, que la complejidad del algoritmo es lineal, O(n), o de orden 1, al menos para los números compuestos.

Con el fin de profundizar un poco más se ha comprobado también con la restricción de generación únicamente de números primos. Esto se hace dado que este es el peor caso de los algoritmos de primalidad y en el anterior set de datos no se presentaban primos suficientes como para ser concluyentes.



De nuevo el ajuste es muy semejante a una línea recta, con la ayuda de la representación de la línea de regresión y el coeficiente R^2 se podrá estudiar la fiabilidad de esta afirmación.



Efectivamente el coeficiente R^2 es prácticamente 1, demostrando que el ajuste es casi perfecto mediante el modelo lineal y que la complejidad es lineal, O(n) o de orden 1, también para el caso de los números primos.

Con estas pruebas podemos afirmar empíricamente de forma concluyente que, para ambos tipos de números, primos y compuestos, el algoritmo antes empleado tiene complejidad O(n).

ANÁLISIS DEL COSTE COMPUTACIONAL

El estudio de complejidad se puede hacer de forma analítica.

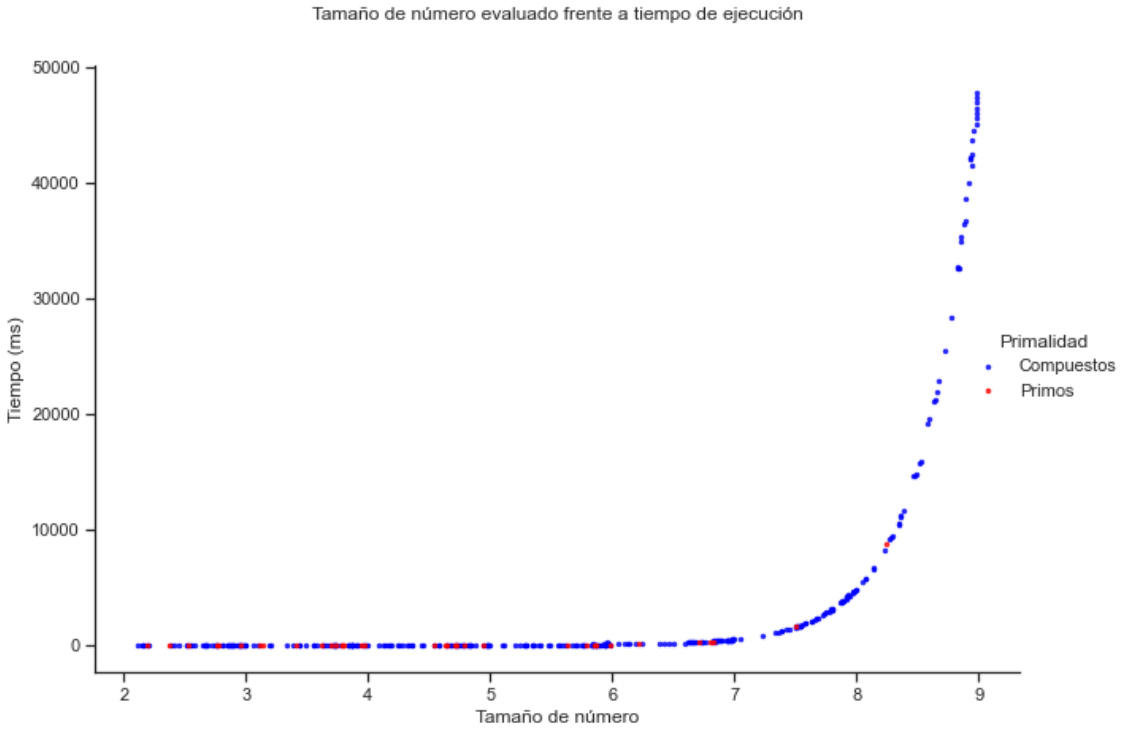
1. Llamada a función: 1
2. Asignación a una variable: 1
3. Bucle: 2+2*(n-2) + condicional: (2+Max (1,0)) *(n-2)
4. Retorno de función: 1

Total = 1 + 1 + 2+2*(n-2) + (2+Max (1,0)) *(n-2) + 1 = 5n -5

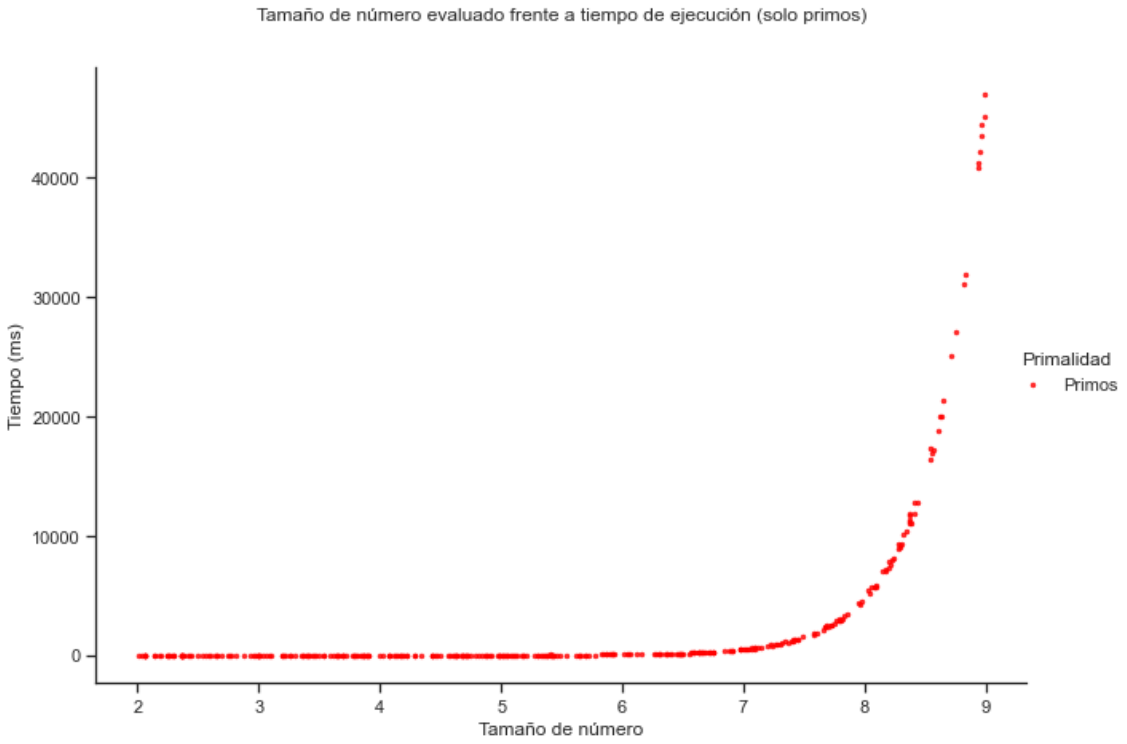
Tras el análisis del coste computacional concluimos que los datos se corresponden completamente con lo esperado, debido a que el algoritmo empleado tiene una complejidad de O(n), es decir, complejidad lineal, que resulta en el ajuste lineal casi perfecto de los resultados.

RELACIÓN TIEMPO Y TAMAÑO DEL NÚMERO EVALUADO

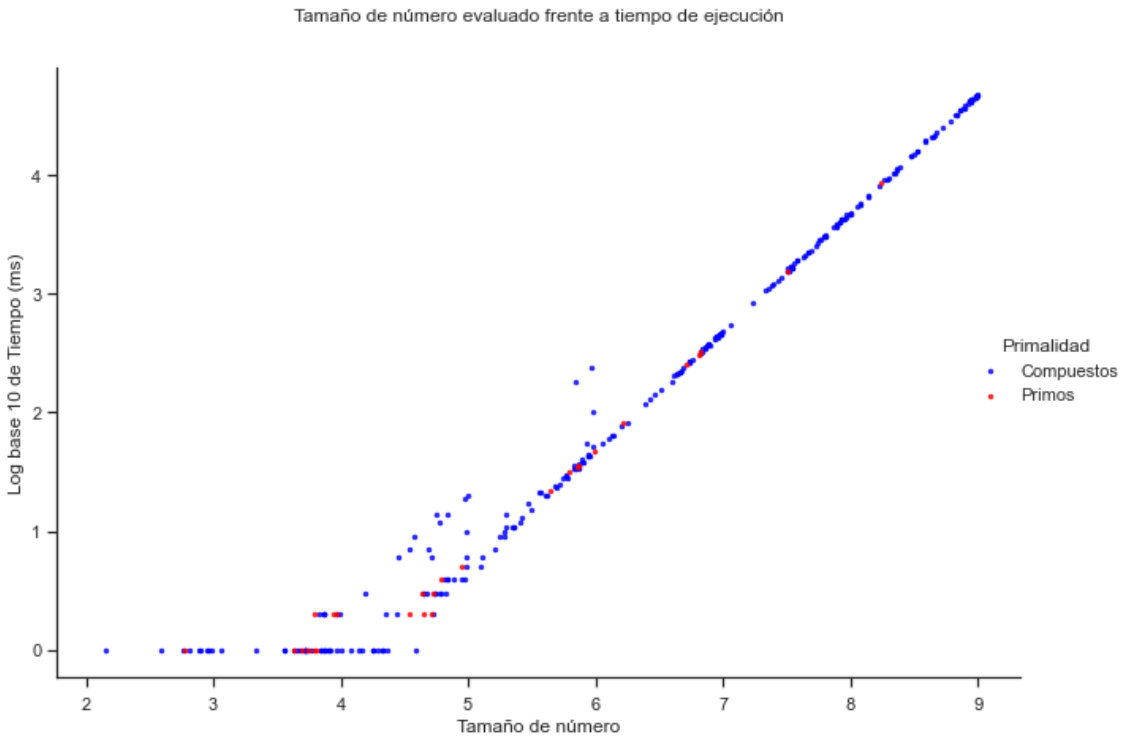
Un análisis que puede resultar interesante es el que relaciona el tamaño del número con el tiempo que tarda el algoritmo de primalidad en evaluarlo. Entendiendo por tamaño del número el logaritmo en base 10 del mismo. A continuación, se procede al estudio empírico de esta relación.



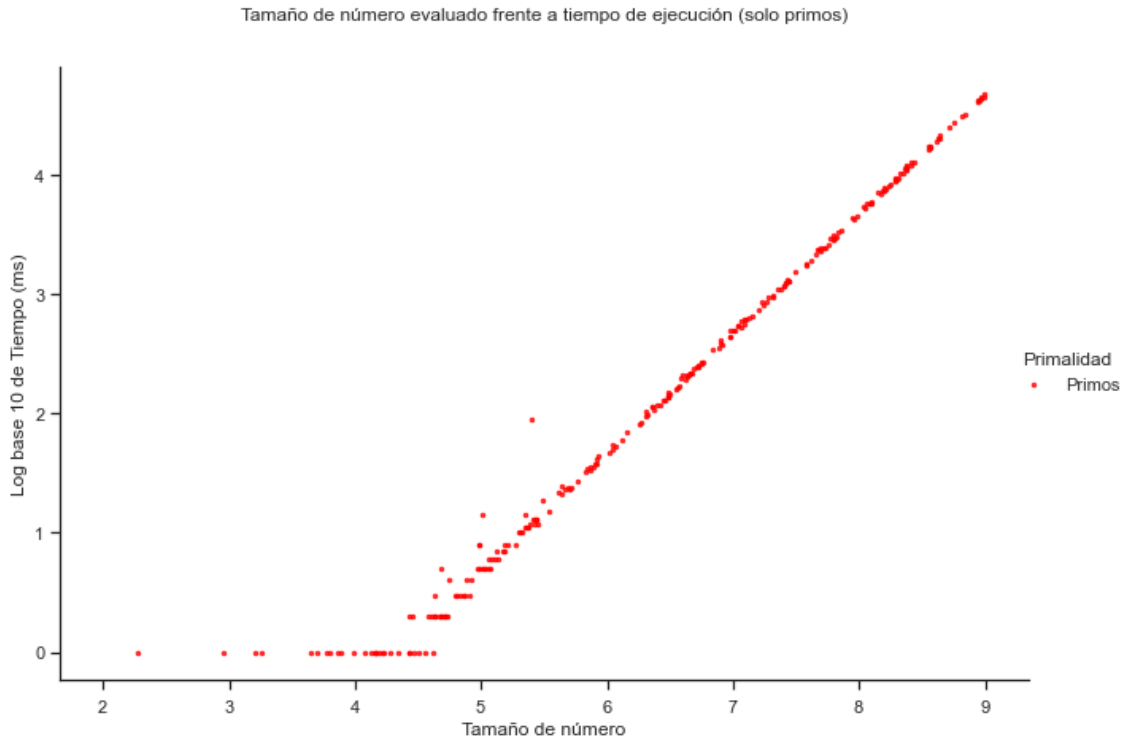
La gráfica apunta a que la relación entre las variables es exponencial, o al menos para el caso de los números compuestos, para los primos no se puede sacar conclusiones dado que el tamaño del dataset de primos es demasiado reducido. Para salvar esta situación se fuerza la generación de números a producir únicamente números primos, arrojando los siguientes datos.



El resultado es muy similar, dado que ambos casos presentan el mismo patrón, para verificar si la gráfica es realmente exponencial se realiza una transformación logarítmica al eje de ordenadas, si esta la gráfica tiene aspecto lineal se podrá afirmar que efectivamente la original era exponencial. El resultado es el siguiente:



Efectivamente, aunque hay mucho ruido en los valores más pequeños, rápidamente la gráfica cobra el aspecto lineal.



Ocurre lo mismo cuando reducimos el dataset a números primos, hay ruido inicial pero conforme aumenta el número la gráfica presenta claramente el aspecto de un modelo lineal. Se puede concluir de esta forma que la relación entre el tamaño del número y el tiempo de evaluación de primalidad es exponencial.

ALGORITMO 1

ALGORITMO DISEÑADO

Para la optimización del algoritmo inicial se han tenido en cuenta tres factores, el primero consiste en hacer la comprobación únicamente hasta la raíz del número evaluado ya que los números siguientes a la raíz multiplicado por alguno de los que se han evaluado anteriormente no puede dar como resultado el número que está siendo evaluado. Para añadir esta mejora se ha incluido un método con el que se calcula la raíz cuadrada.

El segundo factor que se ha considerado consiste en que solo se compruebe hasta que se encuentre un divisor puesto que si esto ocurre hay que descartar que el número sea primo.

Por último, dado que, sin contar el número dos, todos los primos son impares, la cuenta del bucle aumenta de dos en dos, y no de uno en uno, logrando reducir a aproximadamente la mitad el número de comprobaciones.

```
TEST DE PRIMALIDAD (NUMERO_A_EVALUAR: Entero positivo)
Si NUMERO_A_EVALUAR es múltiplo de 2:
      Devolver Falso
Para cada número CANDIDATO_A_DIVISOR entre 3 y la raíz de
NUMERO_A_EVALUAR aumentando de 2 en 2:
      Si CANDIDATO_A_DIVISOR es divisor de NUMERO A EVALUAR:
                  Devolver falso
Fin del bucle
Devolver verdadero
```

La implementación en java del algoritmo queda de la siguiente manera:

```
public static boolean primalityTestImprove1(BigInteger test) {
      BigInteger sqroot = Tools.sqrt(test).add(BigInteger.ONE);
      BigInteger two = BigInteger.valueOf(2L);
      if (test.mod(two).equals(BigInteger.ZERO)) {
            return false;
      }
      for (BigInteger bi = BigInteger.valueOf(3); bi.compareTo(sqroot) < 0;
      bi = bi.add(two)) {
            if (test.mod(bi).equals(BigInteger.ZERO)) {
                  return false;
            }
      }
      return true;
}
```

Este código ha sido verificado antes de realizar las comprobaciones, constatando que identifica a los números primos correctamente.

Pese a que existe una optimización extra, sencilla de añadir y cuyo algoritmo se basa en el hecho de que todos los primos mayores de 3 son congruentes con 1 o -1 modulo 6, se ha optado por no realizar la implementación. Esto se debe a que la complejidad del algoritmo no se reduce, solo se reduce ligeramente el tiempo de ejecución, pero el estudio concluiría con los mismos resultados que sugieren que la complejidad del algoritmo es O(sqrt(N)).

PRUEBAS EXPERIMENTALES REALIZADAS

De manera análoga al algoritmo 0, el algoritmo 1 se ha probado y medido pasándole números aleatorios de diferentes tamaños. Gracias a las optimizaciones de este algoritmo, se ha podido aumentar de manera significativa el rango de números evaluados.

Nuevamente se ha duplicado el experimento, realizando una segunda evaluación en la que solo se le pasan números primos al algoritmo, con el fin de poder mejorar la calidad

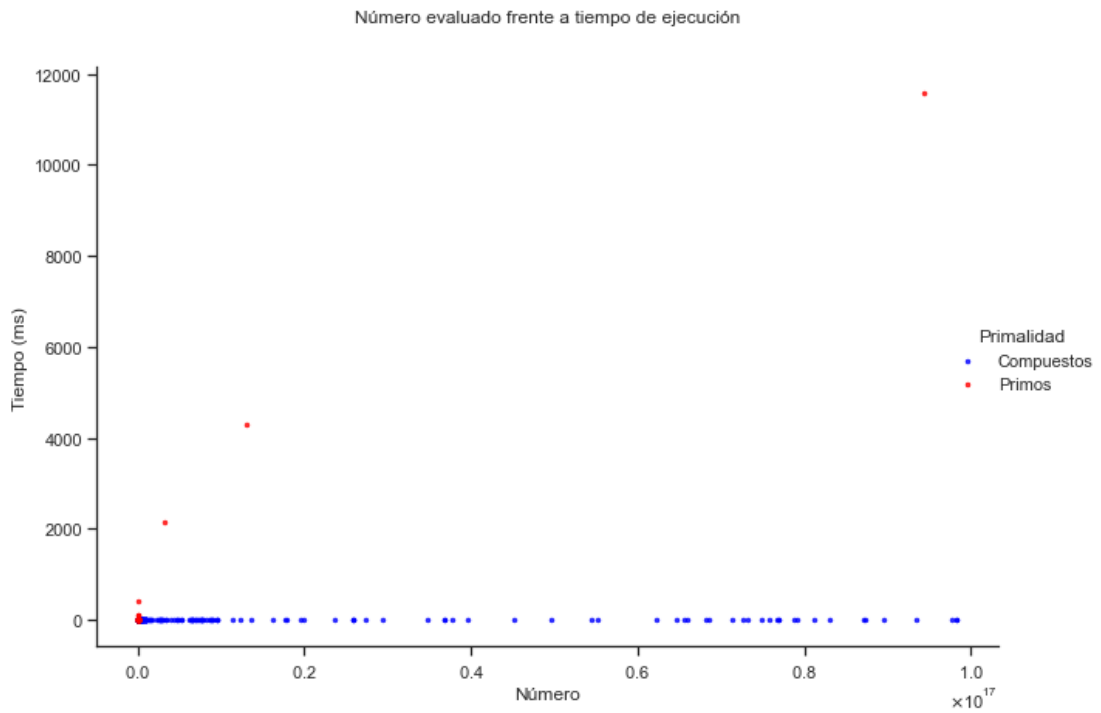
y fiabilidad de las conclusiones. En este caso es incluso más importante porque este algoritmo finaliza en el momento que se encuentra un divisor del número, por lo que la diferencia de evaluación entre los números primos y los compuestos aumenta drásticamente.

La generación ha consistido en 50 números de cada tamaño para ambos casos, tanto el de números aleatorios como el de primos aleatorios. Los tamaños son los del rango entre 3 dígitos y 17 dígitos, ambos incluidos. El límite inferior ha sido 3 porque cualquier número de 1 y 2 cifras arrojaría tiempo de evaluación 0, por lo que no aportan demasiada información a la investigación. Por otro lado, el límite superior es 17 dígitos, porque los números por encima de 10^{18} tienen un coste de computación demasiado elevado y no se puede elevar el tiempo de ejecución al orden de días.

Se han pasado por tanto 650 números al algoritmo en cada experimento, estos se han medido y posteriormente se ha hecho un estudio empírico mediante el uso de programas de autoría propia hechos en Python.

RELACIÓN TIEMPO Y NÚMERO EVALUADO

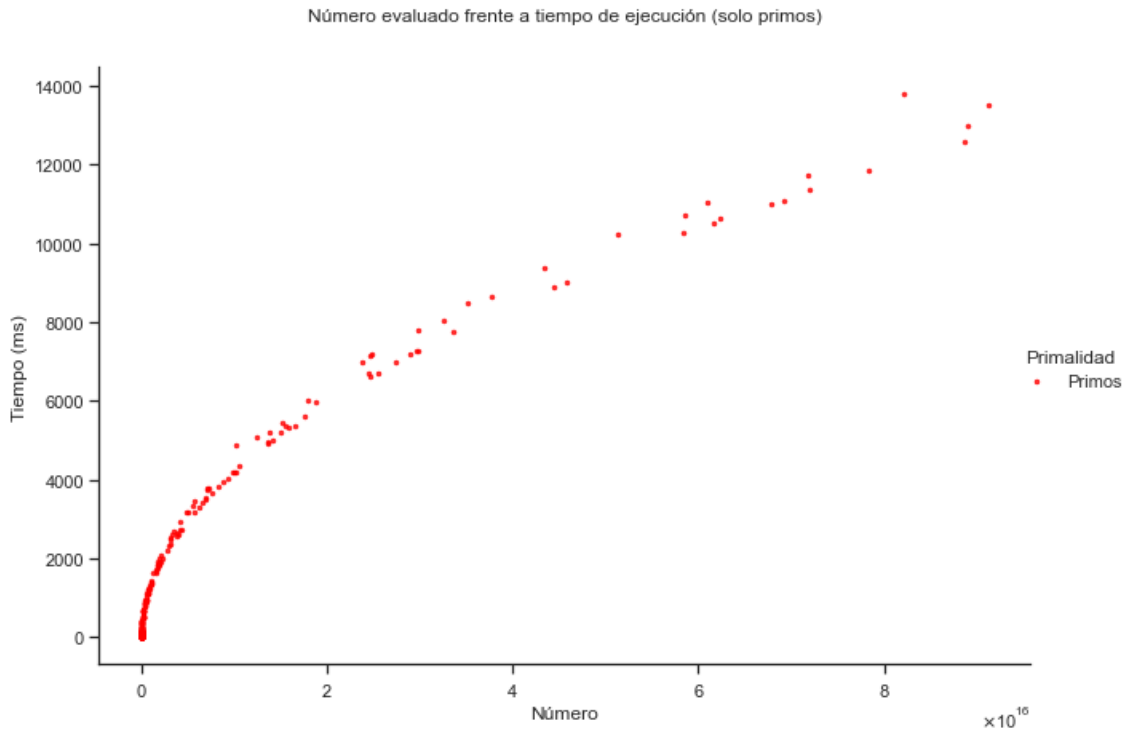
La siguiente gráfica muestra relación entre el número evaluado y el tiempo que ha tardado el algoritmo mejorado en evaluar la primalidad de este. En rojo se representa los números cuya clasificación ha resultado de número primo y en azul los que han resultado ser compuestos.



En esta gráfica se puede observar que en gran parte de los números el coste computacional es muy cercano a 0, ya que se habrán encontrado números por los que son divisibles cercanos al 0. Por otra parte, cabe destacar como el tiempo de ejecución para los números primos y compuestos varia en gran medida, ya que, a diferencia del

algoritmo inicial, al incluir las optimizaciones, sí que existe una diferencia entre la evaluación de un número primo y uno compuesto.

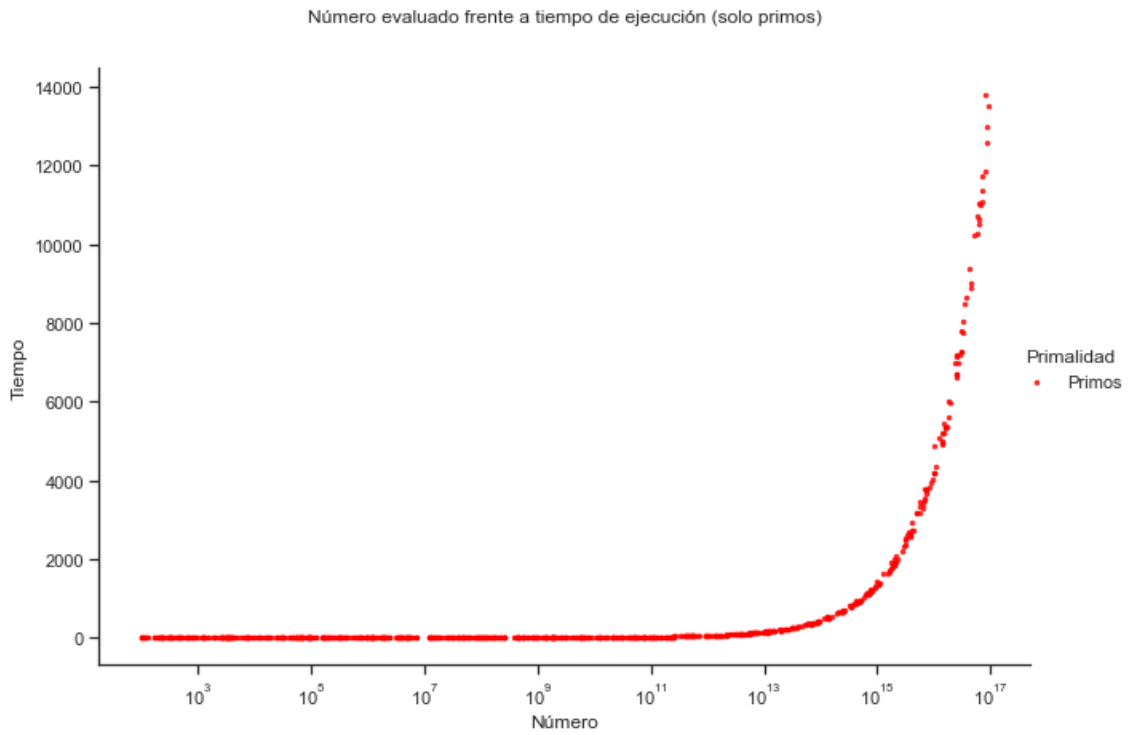
Nuevamente dado que el peor caso del algoritmo de primalidad se da para los números primos, se fuerza a que se generen exclusivamente estos, debido a que en el set de datos anterior no se obtienen suficientes ejemplos como para sacar conclusiones. Los resultados son los siguientes:



Es de esperar que el resultado sea un modelo cuya complejidad se ajuste a O(sqrt(N)), no obstante, dado que durante este proceso la metodología deber ser puramente empírica, la gráfica podría ser tanto un modelo de raíz cuadrada como logarítmico.

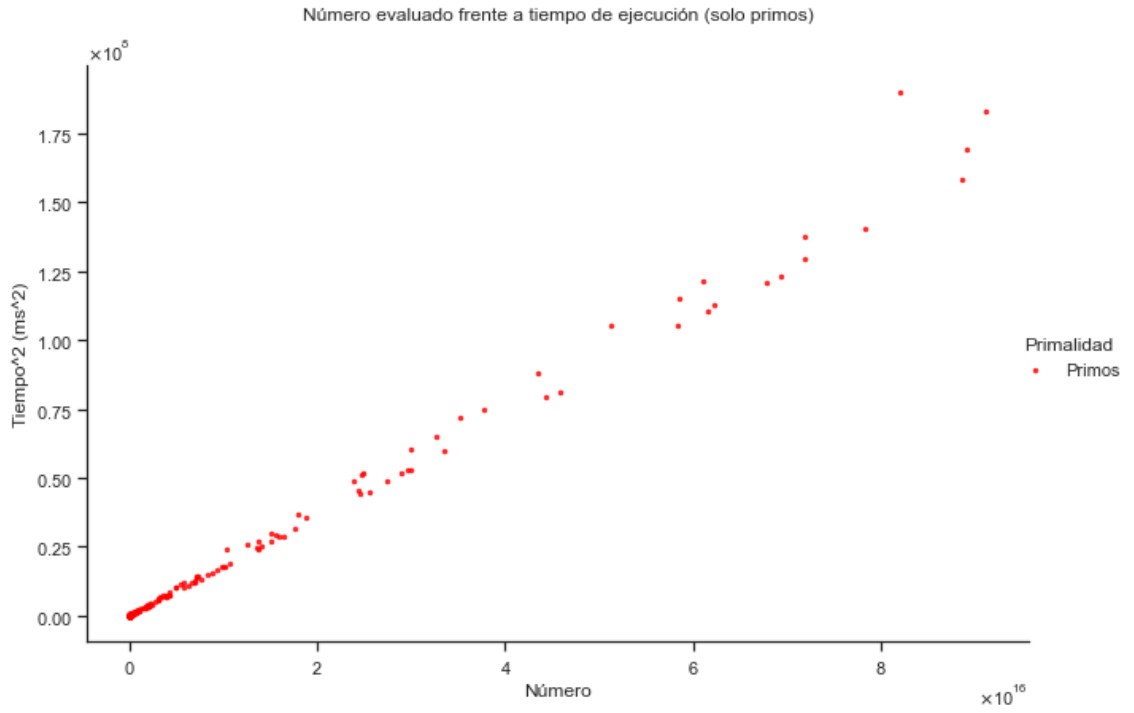
Realizando una transformación exponencial al eje de ordenadas o una logarítmica al de abscisas, si el resultado es una función lineal se podrá concluir que el ajuste original era logarítmico. Optamos por la transformación logarítmica al eje de abscisas, dado que la otra transformación trabaja con números excesivamente grandes.

El resultado es el siguiente:

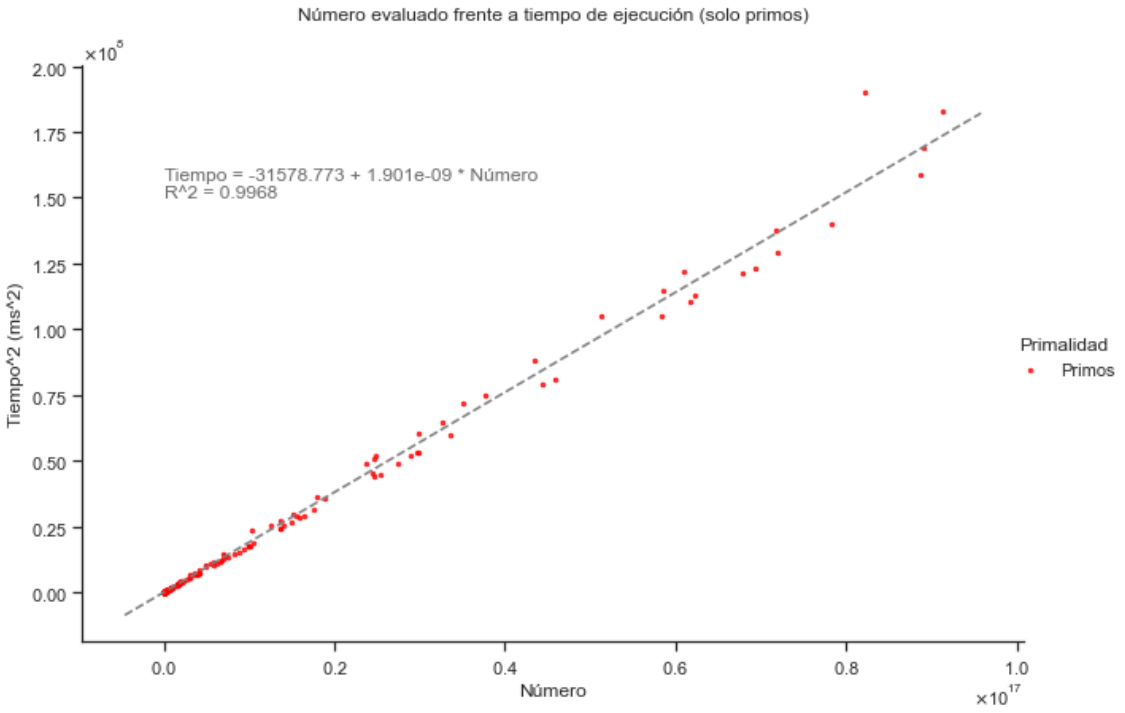


Claramente no respeta los patrones de un modelo lineal, por tanto, pasamos a verificar la hipótesis de que el modelo es potencial, o de raíz cuadrada.

Representando el número frente a los cuadrados de los tiempos se obtiene la siguiente gráfica:



Parece un modelo lineal, para ver con qué incertidumbre se puede afirmar la linealidad de este nuevo modelo se representa también la ecuación de ajuste junto con el parámetro R^2:



El ajuste es muy bueno, dado que el parámetro R^2 es muy cercano a 1 y se puede afirmar por tanto que este nuevo modelo es lineal con un bajo grado de incertidumbre, lo que implica que el modelo anterior de número frente a tiempo era una gráfica potencial de grado ½, o un modelo de raíz cuadrada.

ANÁLISIS DEL COSTE COMPUTACIONAL

Para este análisis se han tenido en cuenta únicamente los números primos que son los que implican un mayor coste computacional.

1. Llamada a la función: 1
2. Comparación: 2
3. Bucle:

$$2 + 2 * \left(\left\lfloor \frac{\sqrt{n} - 3}{2} \right\rfloor + 1\right) + condicional: 2 * \left(\left\lfloor \frac{\sqrt{n} - 3}{2} \right\rfloor + 1\right)$$

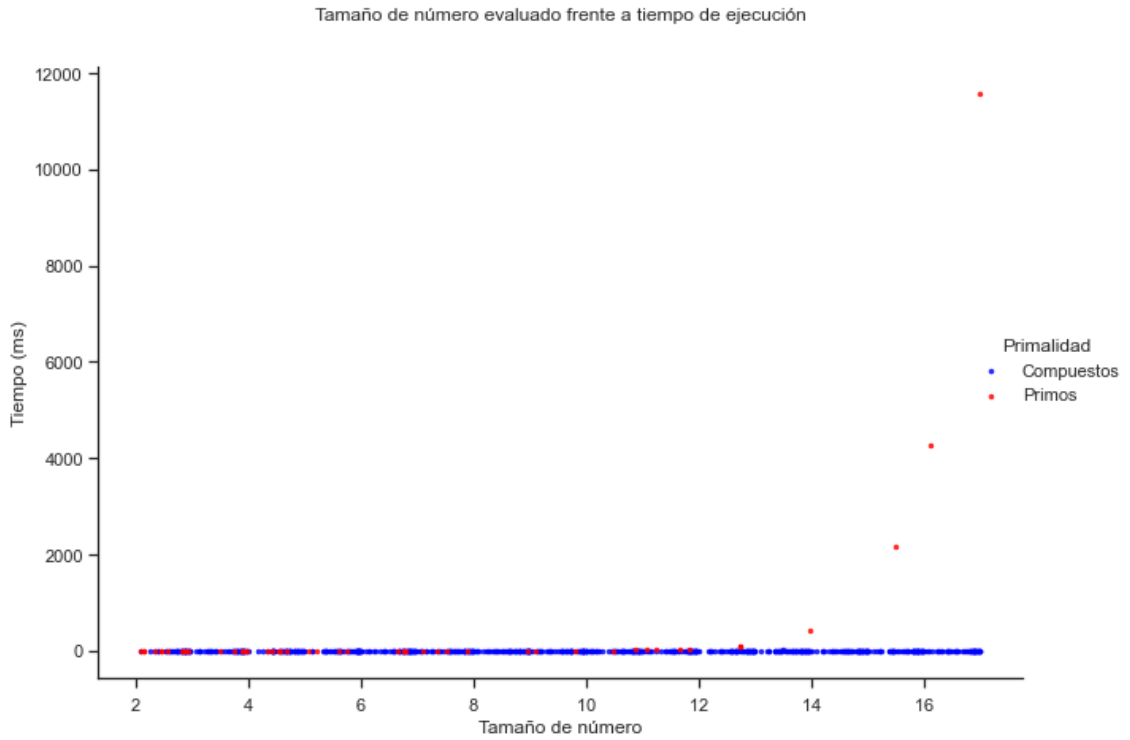
4. Retorno de la función: 1

Total = 1 + 2 + $2 + 2 * \left(\left\lfloor \frac{\sqrt{n} - 3}{2} \right\rfloor + 1\right) + condicional: 2 * \left(\left\lfloor \frac{\sqrt{n} - 3}{2} \right\rfloor + 1\right)$ +1

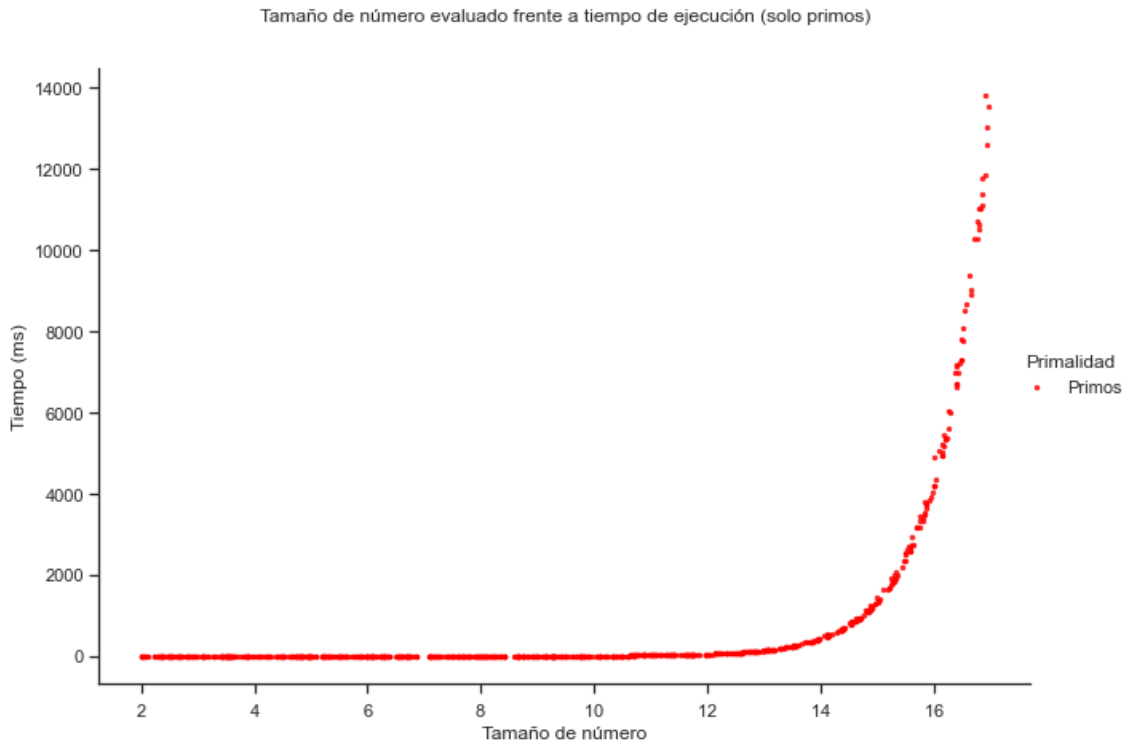
$$= 10 + 4 * \left(\left\lfloor \frac{\sqrt{n} - 3}{2} \right\rfloor\right)$$

RELACIÓN TIEMPO Y TAMAÑO DEL NÚMERO EVALUADO

De nuevo el análisis de la relación entre el tamaño del número y el tiempo de evaluación de primalidad del mismo es un estudio de interés. Para el caso de números aleatorios sin restricción en resultado es el siguiente:

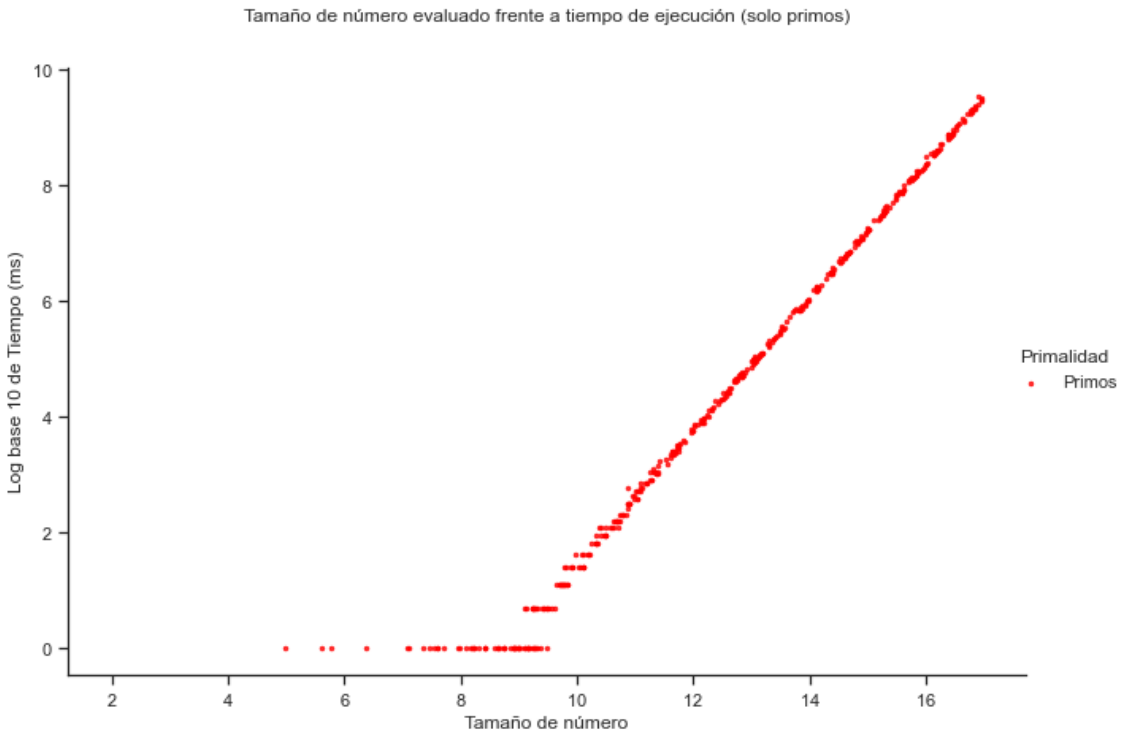


Para los números compuestos todos los tiempos son muy cercanos a 0 y para los números primos, que constituyen el peor caso en términos de complejidad, no se tienen suficientes datos como para sacar conclusiones. Por tanto, se fuerza el experimento a la generación de números primos, y realizamos de nuevo la representación, obteniendo la siguiente gráfica:



Pese a que la complejidad del algoritmo se ha reducido, la relación entre el tamaño del número y el tiempo de ejecución sigue presentando forma de modelo exponencial. Para

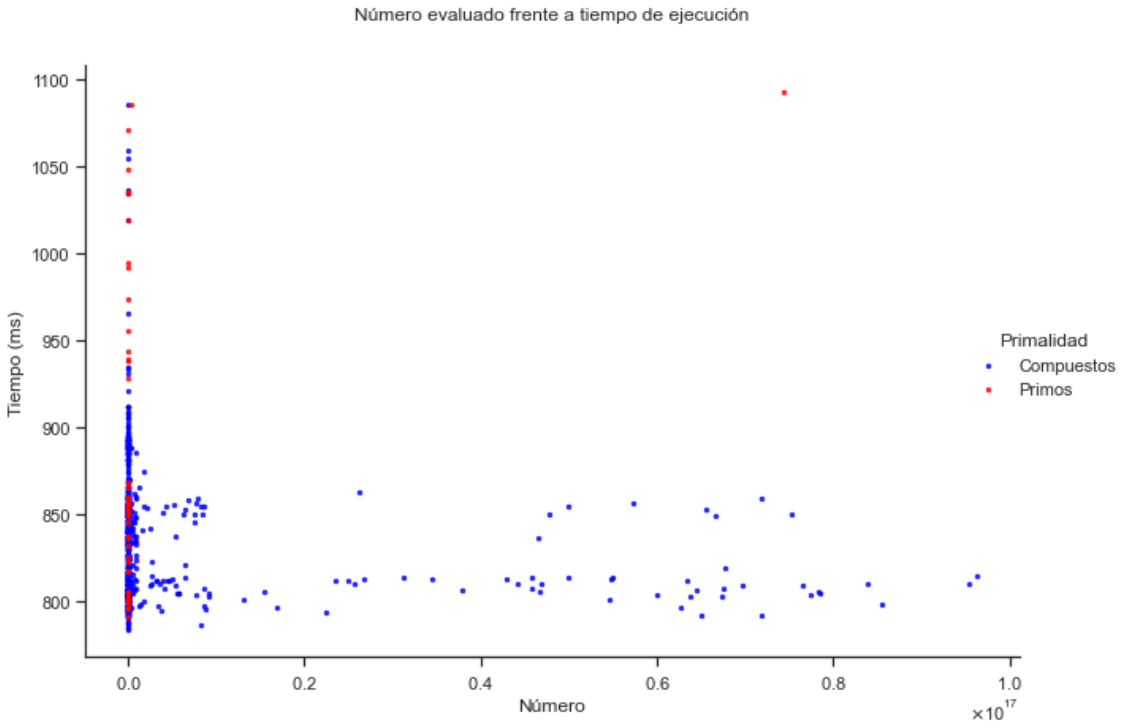
asegurar esta hipótesis, se realiza una transformación logarítmica al eje de ordenadas, obteniendo el siguiente resultado:

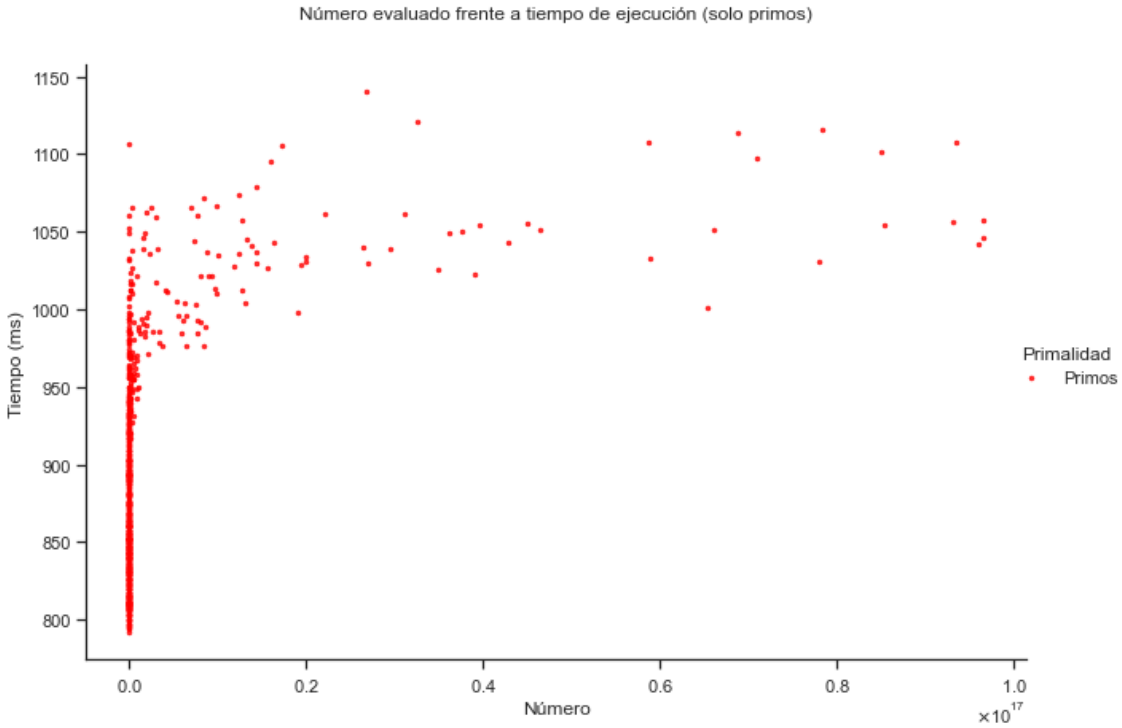


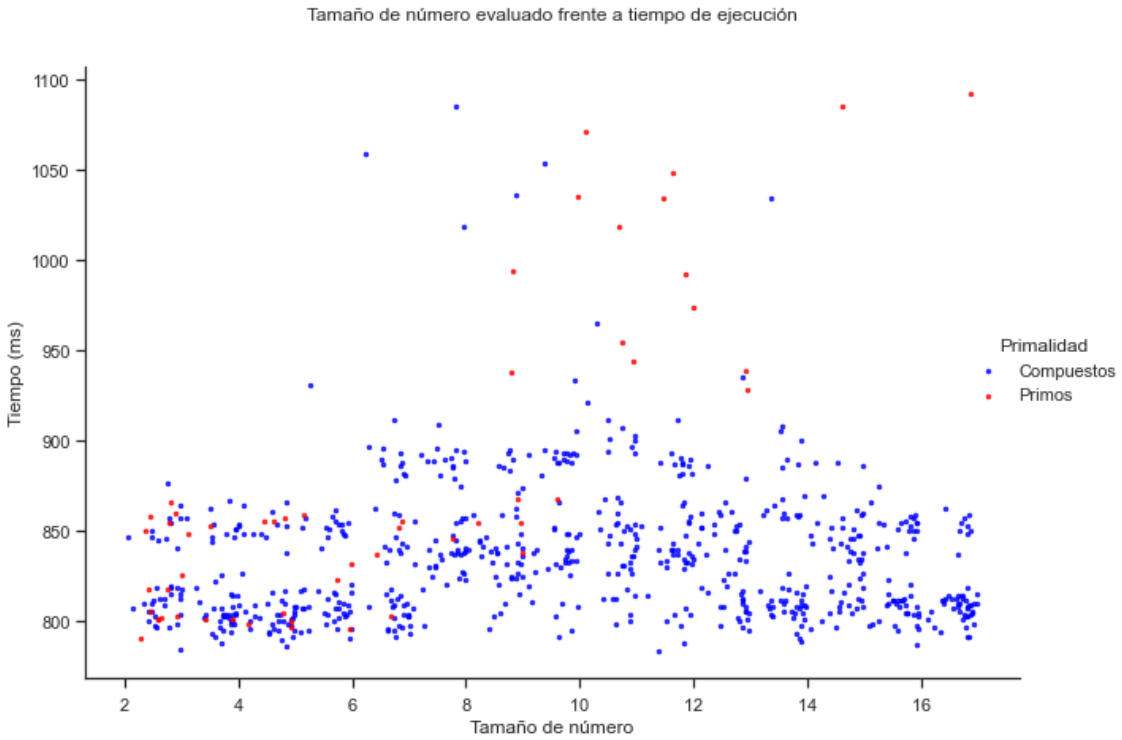
Nuevamente, salvando el ruido de los valores pequeños, el ajuste es lineal, por lo que se puede afirmar que el modelo anterior era exponencial. Se concluye así que la relación entre el tamaño del número y el tiempo de evaluación de primalidad sigue siendo exponencial para este algoritmo en el caso de los números primos.

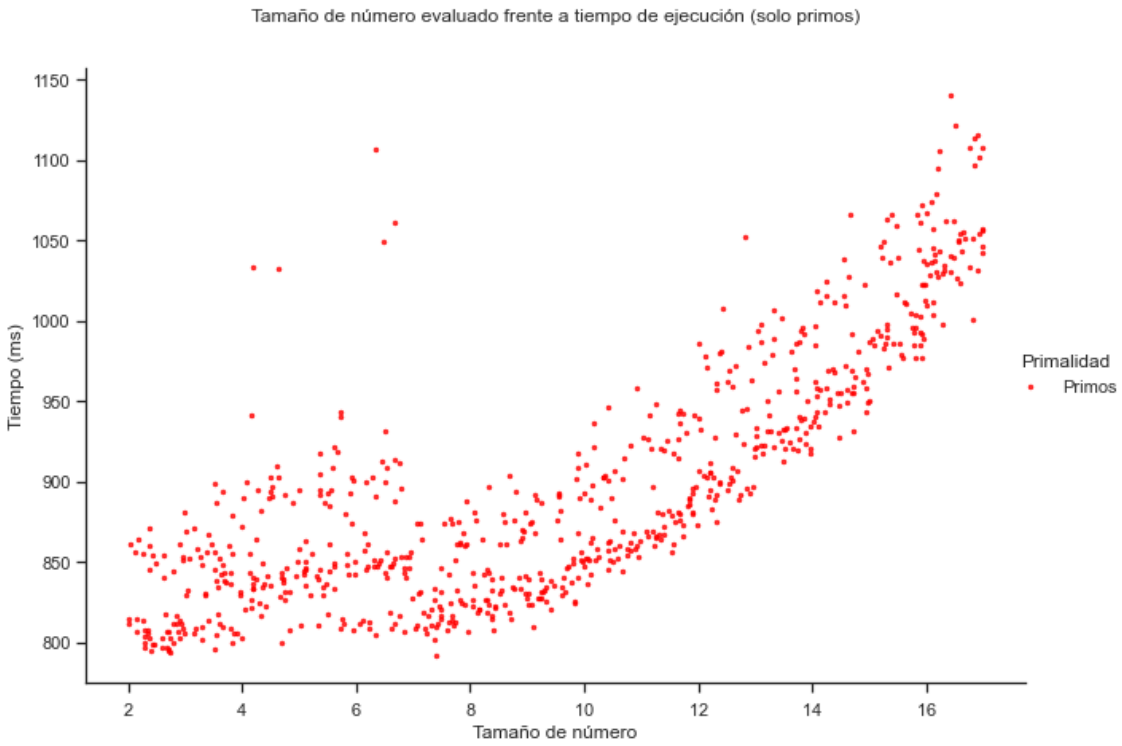
EXTRA: ALGORITMO AKS

Durante el buceo por los algoritmos de primalidad se ha estudiado el algoritmo AKS, y como curiosidad se añaden al documento, junto con sus resultados. El algoritmo es bastante complejo y no es necesario describirlo para los propósitos de este trabajo. Sin embargo, este algoritmo logra reducir la ejecución a una complejidad de $O(\log(N)^{12})$ en el peor caso, arrojando incrementos muy pequeños de tiempo de ejecución para primos de gran tamaño. A continuación, se presentan las 4 gráficas de modelos que se han estudiado para los algoritmos de autoría propia, pero no se realiza de momento el análisis.


Número evaluado frente a tiempo de ejecución
Tiempo (ms)
1100
1050
1000
950
900
850
800
0.0
0.2
0.4
0.6
0.8
1.0
Número
×10^17
Primalidad
Compuestos
Primos


Número evaluado frente a tiempo de ejecución (solo primos)
Tiempo (ms)
1150
1100
1050
1000
950
900
850
800
0.0
0.2
0.4
0.6
0.8
1.0
Número
×10^17
Primalidad
Primos


Tamaño de número evaluado frente a tiempo de ejecución
Tiempo (ms)
1100
1050
1000
950
900
850
800
2
4
6
8
10
12
14
16
Tamaño de número
Primalidad
Compuestos
Primos


Tamaño de número evaluado frente a tiempo de ejecución (solo primos)
Tiempo (ms)
1150
1100
1050
1000
950
900
850
800
2
4
6
8
10
12
14
16
Tamaño de número
Primalidad
Primos

DIFICULTADES ENCONTRADAS

La principal dificultad no ha sido el algoritmo, sino que ha residido en la utilización de la clase BigInteger, pues no se había utilizado nunca y tiene ciertas peculiaridades con respecto a los tipos de datos que se utilizan habitualmente y, en algunas ocasiones ha dificultado la realización de la práctica.

Pese a que en la documentación de BigInteger figura la función sqrt como una función dinámica perteneciente a la clase esta no se encuentra realmente en la clase. Por ello se ha tenido que implementar una función de cálculo de raíces cuadradas sobre la clase BigInteger.

En un primer lugar, se aplica un enfoque centrado solo en Java al proyecto, realizando la representación gráfica de resultados desde el mismo programa Java utilizado para generar los datos. Esto supuso una dificultad, pues nunca se había utilizado y hubo que adaptar implementaciones similares bajo la librería JFreeChart, pese a que sirvió como experiencia y se consiguió el objetivo deseado, finalmente se ha optado por exportar los resultados y realizar el análisis desde Python y Excel, dado que ofrece una mayor libertad para escoger el formato de representación de los datos.

HITO 2

ALGORITMO

La segunda propuesta de los ejercicios prácticos consiste en programar una función de reducción de un número. Esta reducción consiste en reducir a exponente 1 cada uno de los factores primos del número.

Ejemplificando la función, si pasamos como entrada el número 2520, cuya descomposición en factores es: $2^3 \times 3^2 \times 5^1 \times 7^1$. La función debe reducir los exponentes a uno, quedando $2^1 \times 3^1 \times 5^1 \times 7^1$ y devolver el valor resultante, 210.

ALGORITMO DISEÑADO

El algoritmo diseñado para determinar los factores de un número natural y calcular su producto se basará en comprobar los números comprendidos entre el 2 y el número a evaluar (los que llamaremos candidatos a factor) y ver si el número a evaluar es divisible por ese candidato y, en caso de que lo sea se añade a la lista de factores, únicamente se añadirán a la lista aquellos que no estén ya en ella, esta comprobación no se hace manualmente si no que se hace al usar la clase HashSet. A continuación, se establece un nuevo número a evaluar que es el número evaluado anterior entre el candidato a factor.

Por último, se calcula el resultado que al principio tienen valor uno multiplicando cada uno de los factores contenidos en el HashSet.

El algoritmo diseñado sin optimizaciones es el siguiente:

```
REDUCCIÓN DE FACTORES (NUMERO_A_EVALUAR: Entero positivo)
Declarar MI_SET como un hashset vacío de enteros positivos.
Para cada número CANDIDATO_A_DIVISOR entre 2 y NUMERO_A_EVALUAR + 1:
      Si CANDIDATO_A_DIVISOR es divisor de NUMERO_A_EVALUAR:
            NUMERO_A_EVALUAR es NUMERO_A_EVALUAR entre
            CANDIDATO_A_DIVISOR
            Meter CANDIDATO_A_DIVISOR en MI_SET
            Restar uno a CANDIDATO_A_DIVISOR
Fin del bucle
Establecer ACUMULADO a 1
Para cada número FACTOR del SET:
      ACUMULADO es ACUMULADO por FACTOR
Devolver el valor de ACUMULADO
```

Su implementación en código Java queda de la siguiente manera:

```
public static BigInteger factorizationReductionAlgorithm(BigInteger test) {
	Set<BigInteger> set = new HashSet<BigInteger>();

	for (BigInteger bi = BigInteger.valueOf(2); bi.compareTo(test) <= 0;
	bi = bi.add(BigInteger.ONE)) {
		if (test.mod(bi).equals((BigInteger.ZERO))) {
			test = test.divide(bi);
			set.add(bi);
			bi = bi.subtract(BigInteger.ONE);
		}
	}
	BigInteger result = BigInteger.ONE;
	for (BigInteger factor : set) {
		result = result.multiply(factor);
	}
	return result;
}
```

Este código ha sido verificado antes de realizar las comprobaciones, constatando que devuelve de forma correcta el valor de reducción esperado para todos los casos generales, extremos y particulares.

PRUEBAS EXPERIMENTALES REALIZADAS

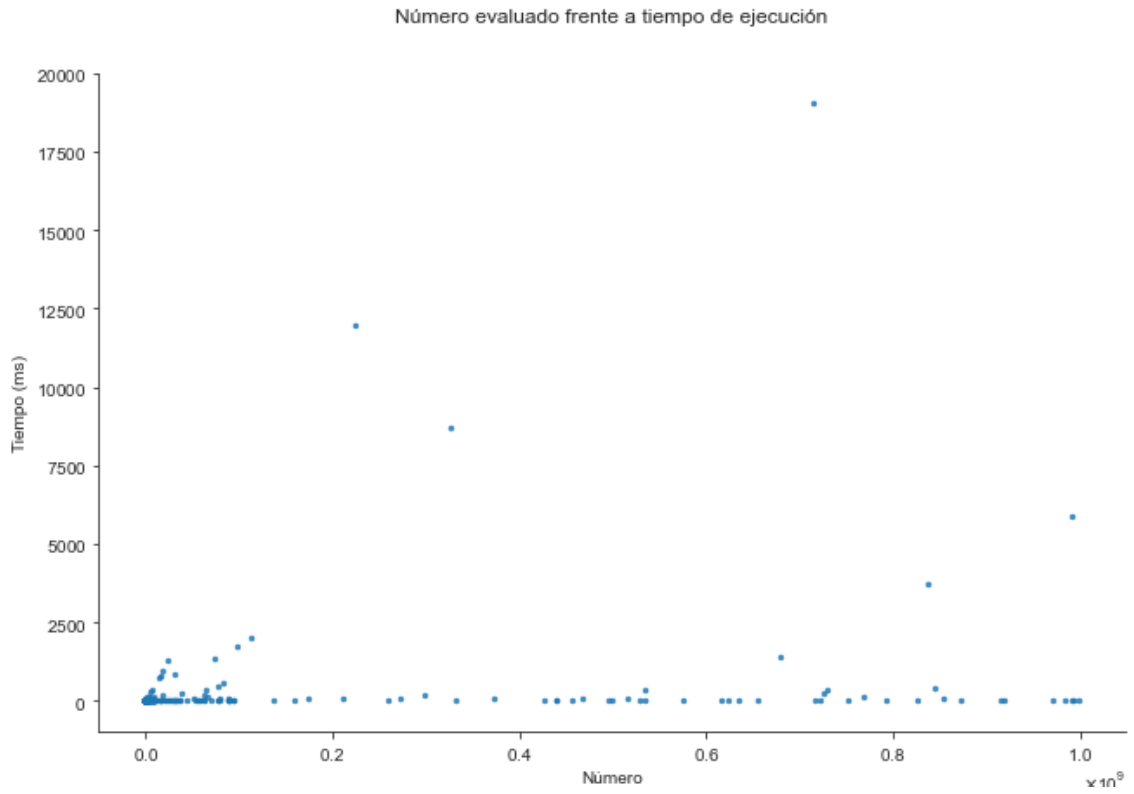
El algoritmo diseñado ha sido utilizado en diferentes números de varios tamaños con el objetivo de medir su ejecución. Tras este caso más general, se ha procedido a realizar otra vez la medición, pero forzando a que todos los números elegidos para el set de datos sean primos. Esto se ha hecho ya que en esta familia de algoritmos el peor caso se da en los números primos.

Se han generado 50 números de cada tamaño para el caso de la muestra de números aleatorios en general y también para el caso de solo primos aleatorios. Se escogerán números desde el rango de 3 dígitos hasta 9 dígitos, ambos incluidos. Se ha escogido empezar el set desde 3 dígitos porque los números más pequeños de 1 y 2 cifras se ejecutarían en tiempos cercanos 0 y esto no aporta información relevante para esta investigación. A su vez, el límite superior puesto en 9 dígitos ha sido escogido porque los números mayores que 10^{10} tienen un coste computacional que se eleva al orden de días y es imposible elevar el tiempo a esta medida.

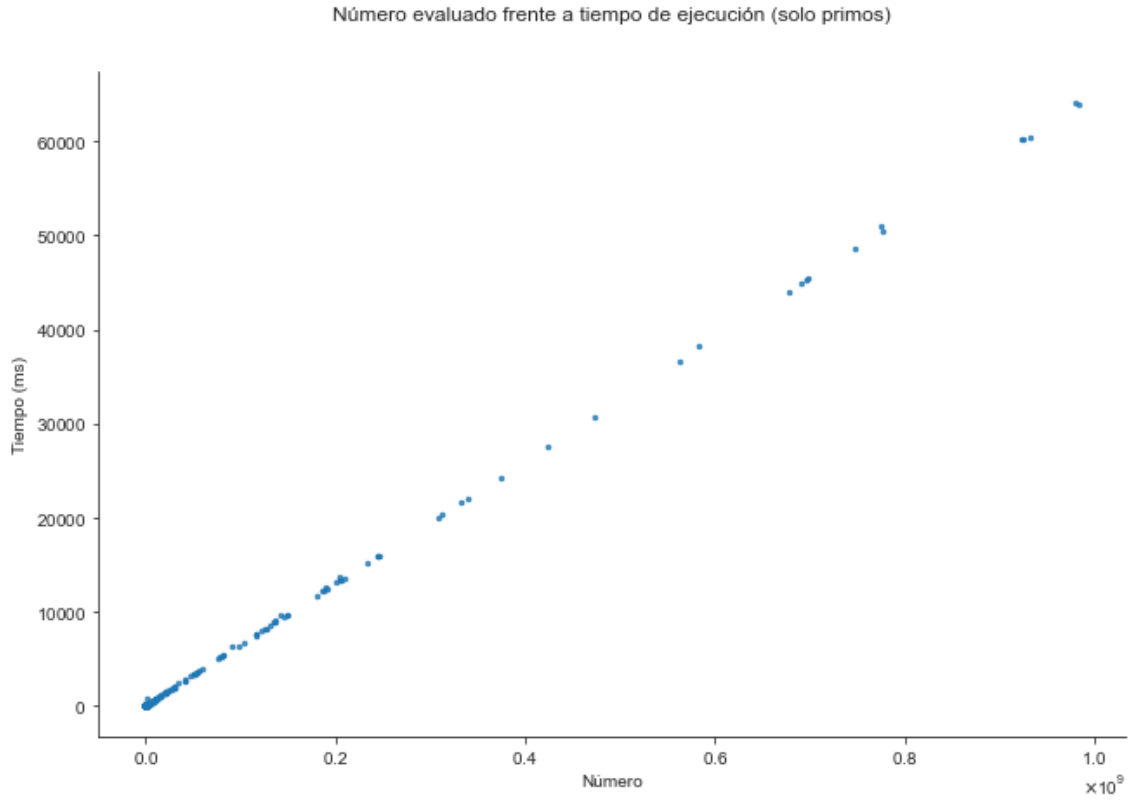
Así el total de números para cada prueba realizada alcanza los 350, estos resultados han sido medidos y posteriormente se ha hecho un estudio empírico mediante el uso de programas de autoría propia hechos en Python.

RELACIÓN TIEMPO Y NÚMERO EVALUADO

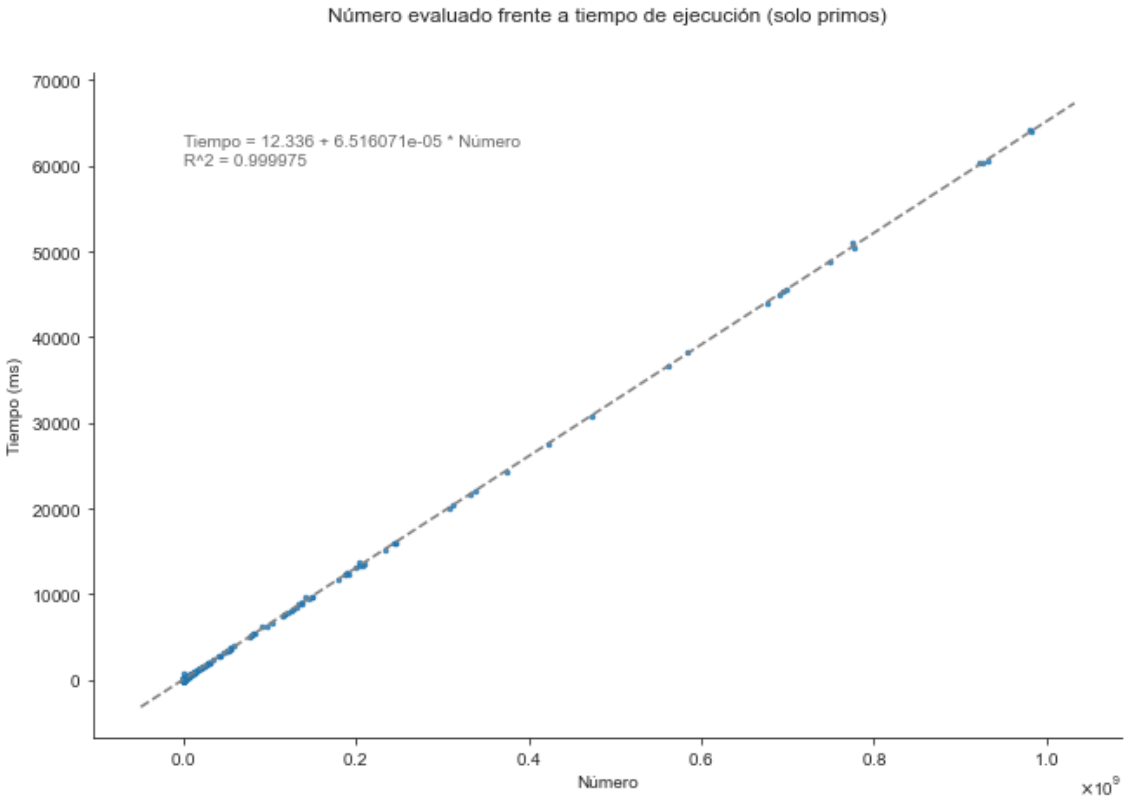
A continuación, se muestra la relación entre el número que pasamos al algoritmo y el tiempo que tarda la función en reducirlo según el procedimiento explicado anteriormente.



Los resultados son pocos concluyentes, se observa una gran cantidad de números muy cercanos al tiempo 0 de evaluación y algunos puntos muy lejanos a esta tendencia. Dado que el algoritmo reduce el tamaño del bucle cuando se encuentra un factor, el peor caso estará en la reducción de números primos. Por ello realizamos un segundo experimento en el que forzamos los números generados y pasados a la función ser primos. Los resultados son los siguientes:



Como se puede observar el ajuste parece seguir una distribución lineal, pero se verificará la fiabilidad de esta afirmación mediante la obtención de la línea de regresión y el coeficiente R^2.



Los puntos se ajustan perfectamente y el coeficiente R^2 es extremadamente cercano a 1, lo que permite afirmar sin incertidumbre que la relación representada es una relación lineal.

Con estas pruebas verificamos empíricamente que, para ambos tipos de números, primos y compuestos, el algoritmo antes empleado tiene complejidad O(n).

ANÁLISIS DEL COSTE COMPUTACIONAL

A continuación, se incluye un estudio analítico de la complejidad para los números primos, pues para el algoritmo diseñado constituyen el peor caso, además, experimentalmente se ha comprobado que son los que requieren un mayor coste computacional.

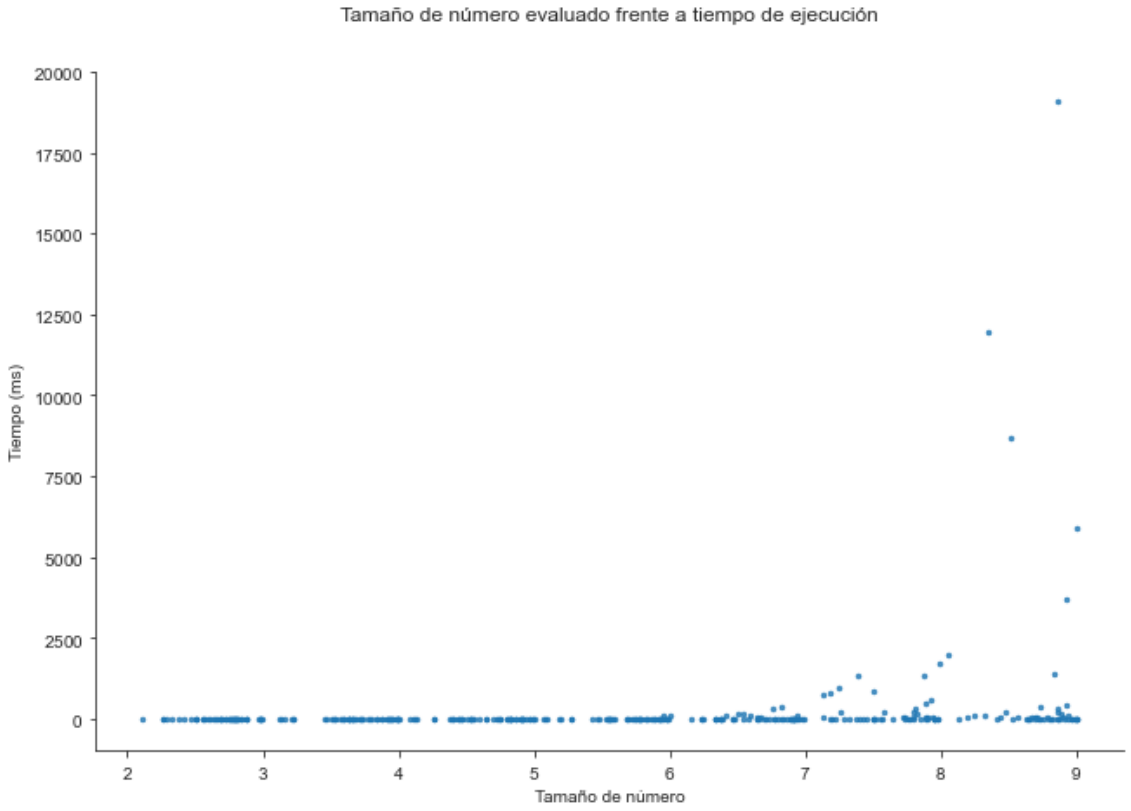
1. Llamada a función: 1
2. Asignación a una variable: 1
3. Bucle: 2 + 2n + Condicional: 2n + 5 (para los números primos)
4. Asignación a una variable: 1
5. Bucle: 4 (para los números primos)
6. Retorno de la función: 1

Total = 1+1+(2+2n) + (2n+5) + 1 + 4 + 1 = 15 + 4n

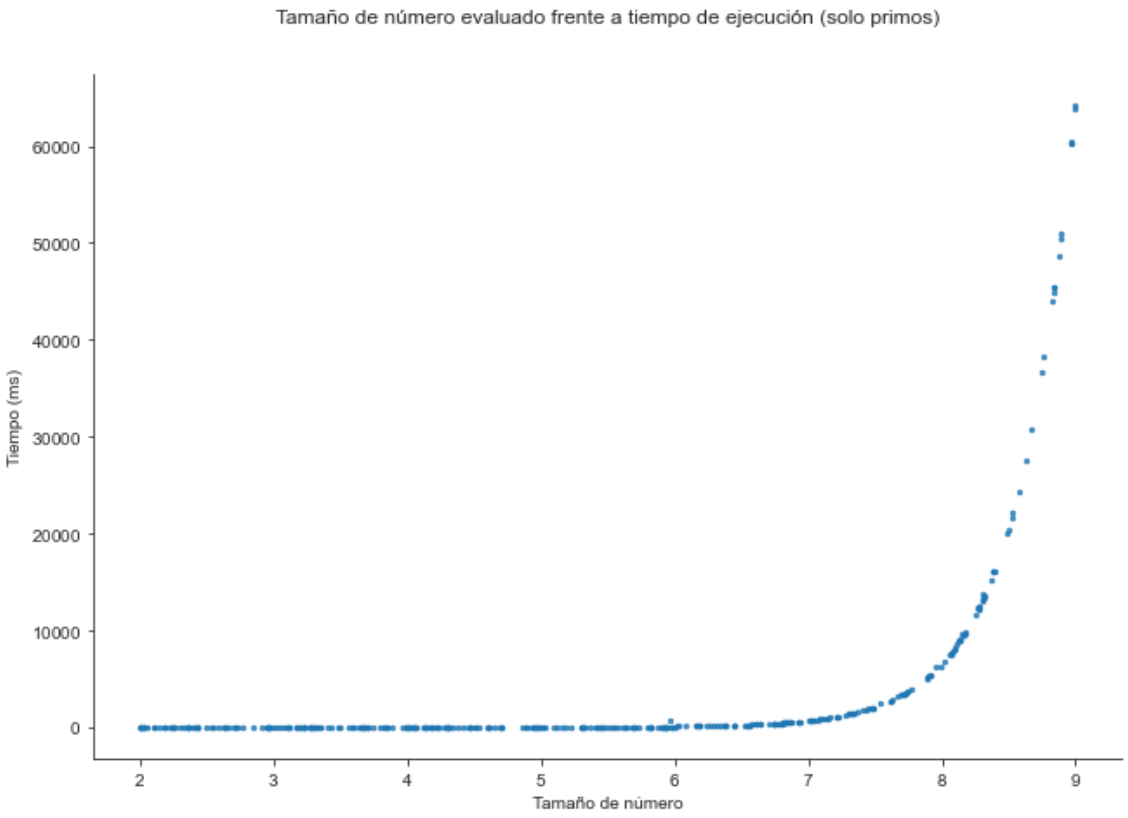
Tras el análisis del coste computacional observamos que los datos se corresponden con lo esperado, ya que el algoritmo empleado tiene una complejidad de O(n), es decir, complejidad lineal, que resulta en el ajuste lineal casi perfecto de los resultados.

RELACIÓN TIEMPO Y TAMAÑO DEL NÚMERO EVALUADO

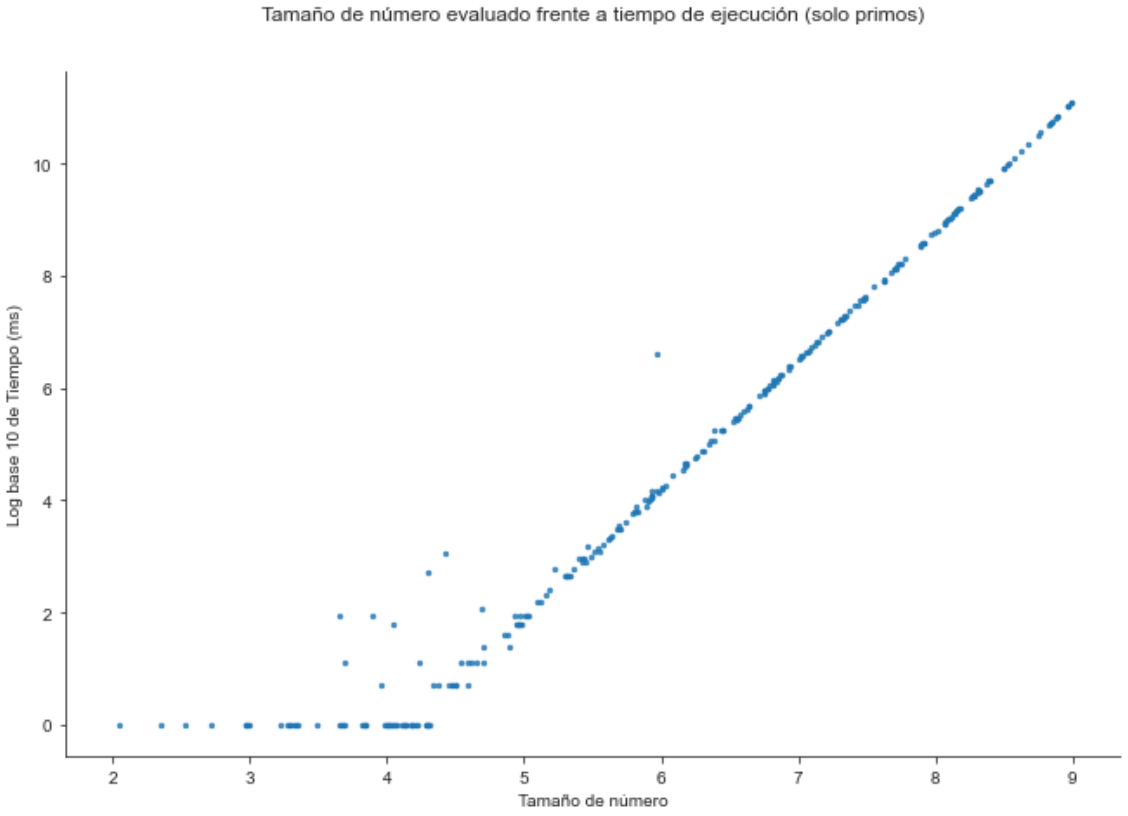
Podemos realizar un estudio que relacione el tamaño del número a reducir con el tiempo que tarda el algoritmo en reducirlo. Entendemos por tamaño del número el logaritmo en base 10 del número. Otra opción sería estudiar el tamaño en función del número de dígitos, pero el uso del logaritmo nos permitirá ser más conclusivos.



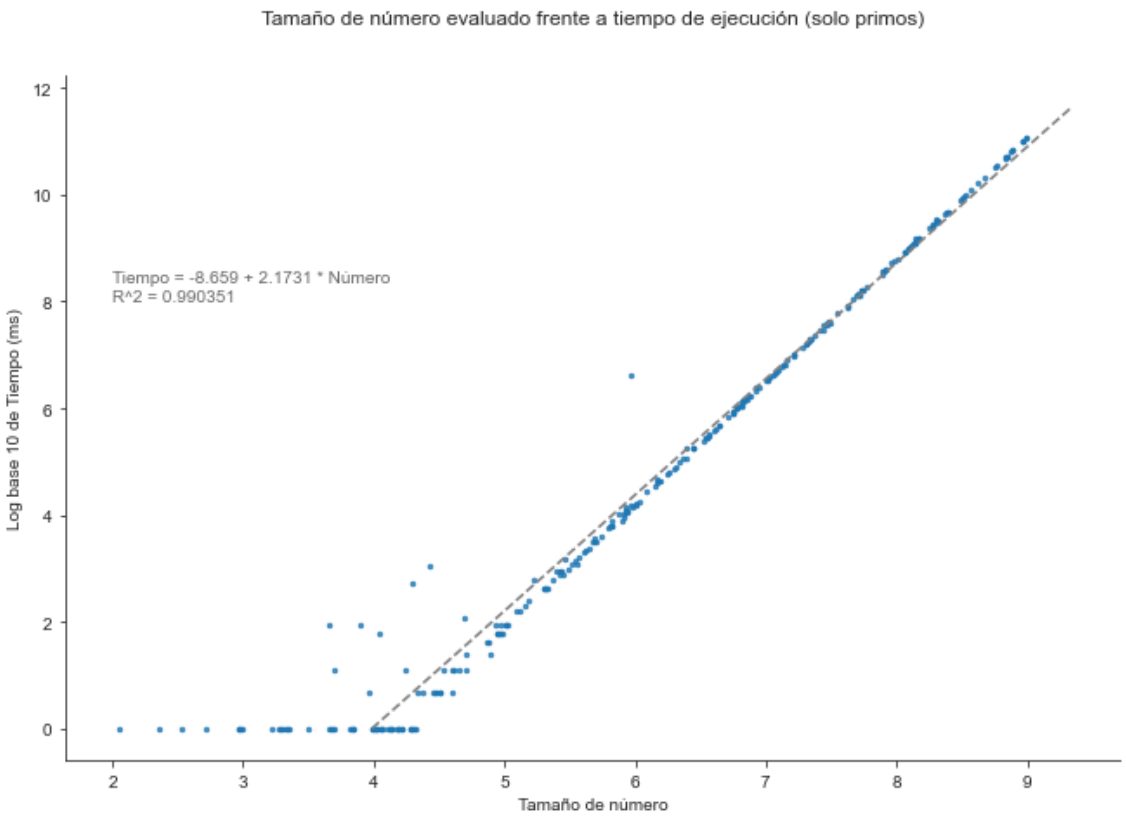
De nuevo en el caso de generación aleatoria de números sin restricciones no podemos localizar ninguna tendencia, muchos valores están cercanos a 0 y hay algunas excepciones en los valores más elevados. Recurrimos, por tanto, a la restricción de generación de números primos, para hacer el estudio de la relación en el peor caso del algoritmo. El resultado es el siguiente:



La relación parece exponencial, pero para confirmar esta hipótesis se aplica una transformación logarítmica al eje de ordenadas, si logramos inducir que entonces la relación es lineal podremos afirmar que la relación previa era exponencial.



Dado que estamos trabajando con logaritmos solo tenemos en cuenta aquellos datos cuyo valor de tiempo es mayor que 0, en este caso añadiendo el modelo lineal con el parámetro R^2 queda de la siguiente forma:



El parámetro R^2 es suficientemente alto como para concluir que la relación actual es lineal, y, con ello, que la relación entre el tamaño del número primo y el tiempo de ejecución del algoritmo de reducción es exponencial.

HERRAMIENTAS GENERALES

Durante ambos hitos se ha hecho uso de algunas herramientas que han jugado un papel muy importante a lo largo de todo el trabajo. Presentamos brevemente los algoritmos a continuación de estas funciones.

OBTENCIÓN ALEATORIA DE BIG INTEGER

La generación de números aleatorios era crucial para poder proceder al análisis empírico con una muestra lo suficientemente grande y representativa. Esta función recibe como parámetro un entero que consiste en el número de cifras que debe tener el número aleatorio generado el algoritmo es el siguiente:

```
GENERACIÓN DE NÚMERO ALEATORIO (NUMERO_DE_CIFRAS: Entero positivo)
Establecer RESULTADO como un String vacío
Hacer NUMERO_DE_CIFRAS – 1 veces:
      Añadir al final de RESULTADO una cifra aleatoria entre 1 y 9
      ambos incluidos.
Fin de bucle

Añadir al final de RESULTADO una cifra aleatoria entre 0 y 9 ambos
incluidos.
Devolver el valor de RESULTADO convertido a BigInteger.
```

El código implementado se expone a continuación:

```
public static BigInteger getRandomBigInteger(int size_of_number) {
      String str_res = "";
      for (int i = 0; i < size_of_number-1 ; i++) {
            str_res += ThreadLocalRandom.current().nextInt(1, 10);
      }
      str_res += ThreadLocalRandom.current().nextInt(0, 10);
      return new BigInteger(str_res);
}
```

OBTENCIÓN ALEATORIA DE BIG INTEGER PRIMO

Para que esta generación fuese lo suficientemente rápida ya no servía la implementación anterior de generar números aleatorios hasta que uno fuese primo y entonces devolverlo. Por ello se ha hecho uso de la función *probablePrime* de la clase BigInteger, esta función recibe el tamaño en bits del número que se quiere generar y devuelve un número que tiene una alta probabilidad de ser primo de ese número de bits, con el primer bit siempre iniciado a 1.

Es decir, si a la función se le pasa el número 4, devolverá un primo aleatorio comprendido entre 1000_2 y 1111_2, en su representación decimal y convertido a BigInteger. De todas formas, para asegurar la primalidad, en nuestra función verificamos la primalidad pasando el resultado al algoritmo AKS, el cual, debido a su rapidez, sacrifica a penas un segundo en la obtención, pero a cambio certifica la validez del resultado.

```
public static BigInteger getRandomBigPrime(int size_of_number) {
	BigInteger result;
	Random rnd = new Random();
	while(true) {
		result
	=BigInteger.probablePrime(transformSizeToBits(size_of_number), rnd);
		if(result.toString().length() == size_of_number) {
			AKS myAKS = new AKS(result);
			if(myAKS.isPrime) break;
		}
	}
	return result;
}
```

Se puede ver que esta función hace uso de la función transformSizeToBits, esta función cumple el papel de seleccionar aleatoriamente una de las posibilidades de tamaño binario que hay dado un tamaño decimal de número.

Entendiéndolo mejor con un ejemplo, si a la función se le pasa el tamaño decimal de 2 cifras, es decir, queremos obtener un primo de 2 dígitos, esta función cumple el papel de calcular que este rango de números se cubre con números de 4, 5, 6 y 7 bits y de seleccionar aleatoriamente uno de esos tamaños:

```
public static int transformSizeToBits(int size_of_number) {
	int bottom_limit = (int)(Math.log(Math.pow(10, size_of_number-
	1))/Math.log(2));
	int top_limit = (int)(Math.log(Math.pow(10,
	size_of_number))/Math.log(2)) + 1;

	return ThreadLocalRandom.current().nextInt(bottom_limit, top_limit +
	1);
}
```

Como es posible que la generación del número dado este tamaño en bits quede fuera del rango pedido en *getRandomBigPrime* se hace una comprobación extra para certificar que el número está en el rango solicitado. Es importante observar que, ante una falla de esta comprobación, se vuelve a pedir un tamaño en bits aleatorio a la función *transformSizeToBits*. Si en lugar de esto, se insistiese con el mismo tamaño de bits hasta generar un primo dentro del rango, estaríamos corrompiendo la uniformidad en la distribución de la generación de números.

DIFICULTADES ENCONTRADAS

Tras haber realizado el primer hito del trabajo ya teníamos familiaridad con la clase BigInteger, por lo que esto ya no supuso un gran reto.

Si ha resultado un tanto desafiante algunas funcionalidades implementadas a la hora de representar los datos. Todas las gráficas han sido generadas en Python por la versatilidad que ofrece. En las gráficas de demostración de linealidad, las cuales presentan la ecuación de regresión, el parámetro R^2 y la línea de regresión, la implementación no era trivial dado que la función lmplot de seaborn no tiene esta funcionalidad.

Es por esto por lo que la agregación de esos elementos la hemos hecho a través de una función propia, pintando sobre el gráfico, ajustando tamaños y colores, y salvando un gran número de excepciones que surgían con los nuevos casos. No obstante, pese a la dificultad estamos satisfechos con el resultado obtenido y creemos en la calidad de las gráficas presentadas.

HITO 3

Es este hito, únicamente se pide evaluar el tiempo de ejecución del algoritmo de primalidad mejorado, con el fin de sacar conclusiones sobre variaciones en función de CPU, sistema operativo, lenguaje, etc.

Pese a que nuestro mejor algoritmo implementado es el AKS, entendemos que no se podrían sacar conclusiones válidas si realizásemos este test con el AKS, dado que no serían comparables a los de otros compañeros. Debido a esto, el algoritmo que evaluaremos será el "Algoritmo 1" del hito 0.

RESULTADOS

La siguiente table recoge los resultados obtenidos:

DÍGITOS	NÚMERO A	TIEMPO (S)	NUMERO B	TIEMPO (S)
6	776159	0,00	776161	0,00
8	98982599	0,00	98982601	0,00
10	9984605927	0,02	9984605929	0,01
12	999498062999	0,06	999498063001	0,05
14	99996460031327	0,643	99996460031329	0,472
16	9999940600088207	4,00	9999940600088209	3,91
18	999999594000041207	40,22	999999594000041209	40,36
19	4611685283988009527	89,75	4611685283988009529	89,58
19	9223371593598182327	129,94	9223371593598182329	129,88

Todos los números de la columna *NÚMERO A*, han sido evaluados como primos, por otro lado, todos los números de la columna *NÚMERO B*, han sido evaluados como compuestos.

CARACTERÍSTICAS DE EJECUCIÓN

DATOS DE LA PRUEBA	
LENGUAJE DE PROGRAMACIÓN	Java
USO DE LIBRERÍA BIG INTEGER	Sí
CPU	Intel Core i5-7300HQ 2.5Ghz
SISTEMA OPERATIVO	Windows 10 Education, Versión 1909
OTROS	Sistema operativo de 64 bits, computadora de 2017, ejecución tras cerrar todo el resto de programas activos.

CONCLUSIONES GENERALES