EJERCICIO PRÁCTICO

ALGORITMO AKS

• FINAL •

23/03/2021 · UC3M · GRUPO 50

Hugo Romero	Irene Fernández	Maribel Medrano
100 386 322	100 386 294	100 386 149

Contenido

INTRODUCCIÓN

Uno de los problemas principales cuando se trabaja con números es el estudio de su primalidad. Numerosos algoritmos que abordan este problema se han desarrollado a lo largo de la historia, estos son deterministas o eficientes, pero no ambas cosas. Este es un gran dilema de la teoría de números y de la complejidad computacional, es por eso por lo que analizaremos el primer test de primalidad cuya complejidad es polinomial y es determinista a la vez que eficiente. Nos referimos al Algoritmo AKS, publicado en 2004 con el título *"Primes is in P"*, por Manindra Agrawal, Neeraj Kayal y Nitin Saxena [1].

Para realizar el estudio separaremos el algoritmo en las distintas fases que tiene, y abordaremos cada una de ellas mediante un estudio analítico y un estudio empírico.

A través del estudio analítico determinaremos la complejidad temporal $T(n)$ y $O(n)$, examinando por separado los distintos pasos.

A continuación, nos centraremos en realizar un análisis empírico para poder determinar experimentalmente $T(n)$, Se realizarán pruebas de ejecución en números aleatorios de diferentes tamaños, entendiendo por tamaño la cantidad de dígitos que tiene el número. Esto lo haremos para poder medir los tiempos de ejecución y generar gráficos con sus resultados.

El modus operandi, tras la experiencia lograda en otros estudios de complejidad, se ha realizado mediante el corte temporal en la generación de números. Es decir, comenzando por un número de dígitos de 3 y con un límite superior inalcanzable (normalmente 200 dígitos), se han realizado 70 repeticiones de generación de números. En algunos casos 70 primos y 70 complejos y en otros casos solo 70 primos, dependiendo de lo que tuviese más sentido en cada función. Tras ello se pasa a la siguiente generación de números (un dígito más que la anterior).

El tiempo total de ejecución se ha establecido para que sea de aproximadamente 8 horas, repartidas de forma equitativa entre las distintas subfunciones.

HEURÍSTICA 1

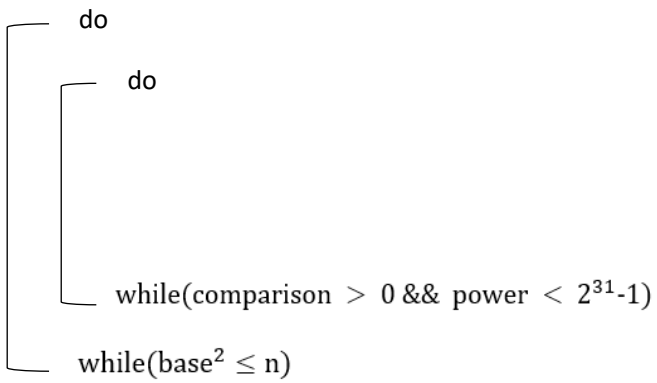
La primera parte que se va a estudiar se basa en conocer si el número evaluado es una potencia perfecta. Para esto se buscará una potencia perfecta que sea igual al entero n que estudiamos, es decir:

Veremos si existen $a, b \in \mathbb{N}$ tal que $n = a^b$

Si esto pasa, se llega a la conclusión de que el numero estudiado es compuesto. Este estudio se hace mediante dos bucles do while, uno contenido dentro del otro.

ESTUDIO ANALÍTICO

```
public boolean isPower(){
   // If ( n = a^b for a in natural numbers and b > 1), output COMPOSITE
    BigInteger base = BigInteger.valueOf(2);
    BigInteger aSquared;
    do {
        BigInteger result;
        int power = Math.max((int) (log() / log(base) - 2), 1);
        int comparison;
        do {
            power++;
            result = base.pow(power);
            comparison = n.compareTo(result); }
        while (comparison > 0 && power < Integer.MAX_VALUE);
        if (comparison == 0) {
            factor = base;
            return false; }
        base = base.add(BigInteger.ONE);
        aSquared = base.pow(2); }
    while (aSquared.compareTo(this.n) <= 0);
    return true;
}
```



En el bucle exterior, teniendo en cuenta que se permanece en él mientras que la base al cuadrado sea menor o igual que el número evaluado. La base tiene al principio el

valor 2 y se va incrementando en 1 cada vez que se ejecuta el bucle, por lo tanto, el total de veces que se ejecuta el bucle es:

$$\sqrt{n} - 1$$

En el segundo bucle, se hacen dos comprobaciones:

La primera consiste en verificar si el valor *comparison* es mayor a cero, lo que implica que $n < base^{power}$, esta tiene una complejidad de $\lceil \log_{base}(n) \rceil - power_{inicial}$. Con respecto a la segunda comprobación, para que permanezca en el bucle, tiene que ocurrir que: $power < 2^{31} - 1$.

Juntando ambas condiciones se llega a la siguiente complejidad total:

$$\max\{(\lceil \log_{base}(n) \rceil - power_{inicial}), (2^{31} - 1 - power_{inicial})\}$$

En cuanto al cuerpo del segundo bucle se han obtenido los siguientes resultados:

Operación	Complejidad
$\boldsymbol{power++}$	C_1
$\boldsymbol{resultado = base^{power}}$	C_2
$\boldsymbol{comparison = n.compareTo(resultado)}$	C_3

Teniendo en cuenta todo lo anteriormente analizado, la complejidad de la primera heurística es:

$$O(\sqrt{n}\log(n))$$

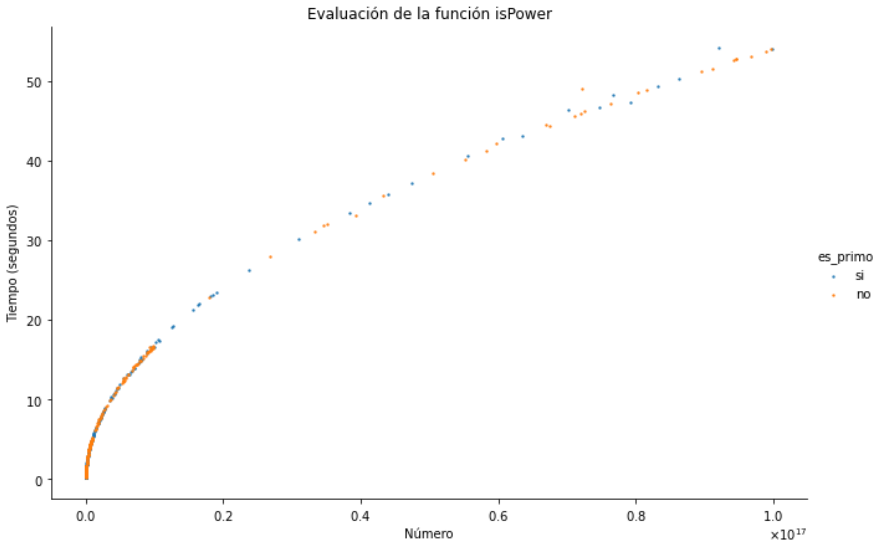
ESTUDIO EMPÍRICO

Un breve repaso univariado viene bien para entender los valores con los que se está trabajando, a continuación, se muestran las características de la base de datos generada. Cabe destacar que solo se consideran aquellos casos en los que el tiempo de ejecución mínimo, en este caso el umbral está establecido en 0,2 segundos.

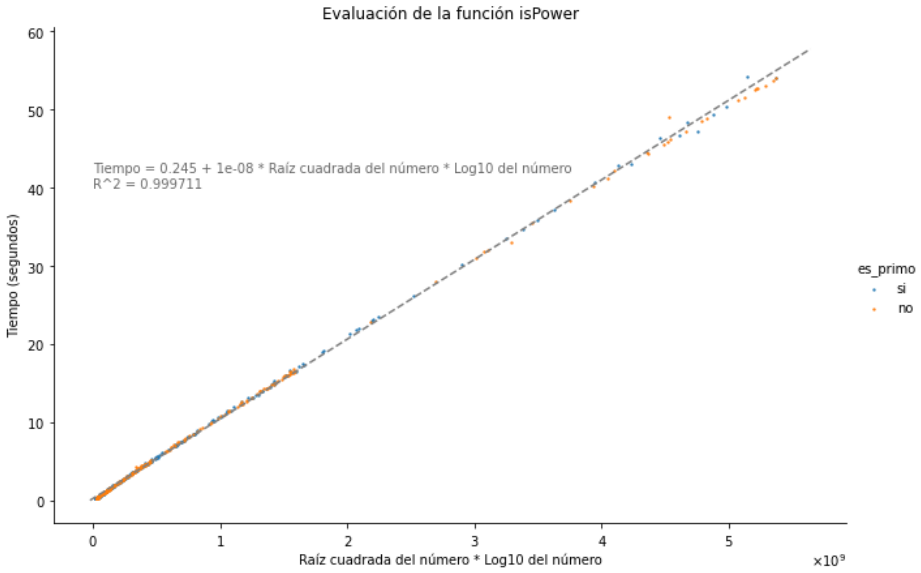
	Número	Tiempo
Casos	562	562
Media	6848256044339252 ≈ 6’85 * 10^{15}	7’758 s
Min	3049845190277 ≈ 3’05 * 10^{12}	0’202 s
Max	99894001913493871 ≈ 9’99 * 10^{16}	54’16 s

	Primalidad
Total	562
Valores únicos	2
si	273
no	289

Los siguientes datos muestran la relación entre el numero evaluado y el tiempo que ha tardado en evaluar si era una potencia perfecta o no. En azul se representan los números primos y en rojo los compuestos.



Es de esperar que el resultado sea un modelo cuya complejidad se ajuste a $O(\sqrt{n}\log(n))$, que es el resultado del análisis analítico. Para verificarlo, realizamos la transformación correspondiente a la complejidad al eje de abscisas, obteniendo lo siguiente:



Parece un modelo lineal, para ver con que incertidumbre se puede afirmar la linealidad de este modelo se analiza la ecuación de ajuste y el parámetro R^2.

Se observa que es un ajuste muy bueno, con R^2 tan cercano al 1 podemos afirmar con bajo nivel de incertidumbre que el nuevo modelo es lineal. Esto implica que el modelo visto anteriormente del tamaño del número frente al tiempo en que tarda en evaluar si es una potencia perfecta era una gráfica de grado $O(\sqrt{n}\log(n))$.

HEURÍSTICA 2

La segunda parte del algoritmo consiste en determinar si:

- $\exists a \leq r \mid 1 < mcd(n,a) < n$
- $Siendo\ r\ el\ mín\ r \mid O_r(n) > \log_2 2\,(n)$

Ya que, si lo anterior se cumple, n es compuesto.

CÁLCULO DEL MÍNIMO r

Primero nos centraremos en encontrar un r que cumpla la condición previamente planteada.

ESTUDIO ANALÍTICO

Para la determinación de r se realiza el siguiente *do while*:

```java
public BigInteger obtainR(){
    // Find the smallest r such that o_r(n) > log^2 n
    // o_r(n) is the multiplicative order of n modulo r
    // the multiplicative order of n modulo r is the
    // smallest positive integer k with n^k = 1 (mod r).
    double log = this.log();
    double logSquared = log * log;
    BigInteger k = BigInteger.ONE;
    BigInteger r = BigInteger.ONE;
    do {
        r = r.add(BigInteger.ONE);
        k = multiplicativeOrder(r); }
    while (k.doubleValue() < logSquared);
    return r;
}
```

do

$k = multiplicativeOrder(r)$

$while\ (k.doubleValue() < \log^2(n))$

Para encontrar un r que verifique $O_r(n) > \log_2 2\,(n)$ se testearán sucesivos valores de r que cumplan $n^k \neq 1(mod\, r)$ para cada valor de $k \leq \log_2 n$. Esto se traduce para un r en particular en un máximo de $O((\log(n))^2)$ multiplicaciones modulo r y una complejidad de $O\left(\left(log(n)\right)^2 \log(r)\right)$.

Sabemos que para cualquier valor de n del cual queramos saber su primalidad, existirá $r \leq máx(3, \lceil(\log n)^5\rceil)$ que $O_r(n) > (log(n))^2$ (Manindra Agrawal et al. 2004).

Esto es así ya que:

- Para $n = 2$ será trivial si $r = 3$ ya que $log\, 2 = 1$ y $O_3(2) = 2 > (\log 2)^2 = 1$.
- Mientras que para $n \geq 3$:
  Se llega a la conclusión que $\left[\left(\log(\mathrm{n})\right)^5\right] > 10$ debido a que $log\, 3 = 1{,}585$ y $\left(\log(3)\right)^5 = 10{,}00218 > 10$.

  Cada r_i divide al siguiente producto:

$$n \prod_{i=1}^{\left[\left(\log(\mathrm{n})\right)^2\right]} \left(n^i - 1\right) < n^{(\log n)^4}$$

  Por lo que aplicando propiedades de logaritmos se llega a que:

$$n^{\left(\log(\mathrm{n})\right)^4} = (2^{log\, n})^{\left(\log(\mathrm{n})\right)^4} = 2^{\left(\log(\mathrm{n})\right)^5}$$

Esto indica que solo tenemos que probar $O(log_5 n)$ valores de r y por tanto se tiene una complejidad en este bucle de:

$$O\left(\left(\log(\mathrm{n})\right)^7\right)$$

Este algoritmo *MultiplicativeOrder* que se encuentra dentro del bucle lo obtenemos ya que, en la Teoría de Números, si tenemos un entero A y un entero positivo r donde ambos sean coprimos, obteniendo $mcd(A, r) = 1$, podemos hallar que el orden multiplicativo de r será el entero positivo más pequeño k que cumpla:

$$A^k (mod\, r) = 1 \qquad \text{Teniendo } 0 < k < r$$

Realizaremos un pequeño análisis para ejemplificar esta fórmula:

Teniendo los valores de A = 191 y r = 50, tenemos como resultado k=5. En la fórmula se observa cómo se comprueba los valores k desde 1 a $r - 1$ que cumplan con la ecuación especificada anteriormente.

$$191^1 = 191 (mod\, 50) = 41$$

$$191^2 = 191 (mod\, 50) = 31$$

$$191^3 = 191 (mod\, 50) = 21$$

$$191^4 = 191 (mod\, 50) = 11$$

$$191^5 = 191 (mod\, 50) = 1$$

Y al llegar al 5 el código se detendrá ya que habrá encontrado el menor k que cumpla la fórmula.

Teniendo lo anterior llegamos a la conclusión de que este grupo de valores tendrá un φ(n) elementos (siendo φ la función de Euler).

$$\varphi(n) = \{r \in \mathbb{N} \mid r \leq A \wedge mcd(A, r) = 1\}$$

Por otro lado, se tiene una complejidad de $O_r(n)$ y los residuos que genera A se repetirán de forma cíclica hasta llegar al valor A - 1.

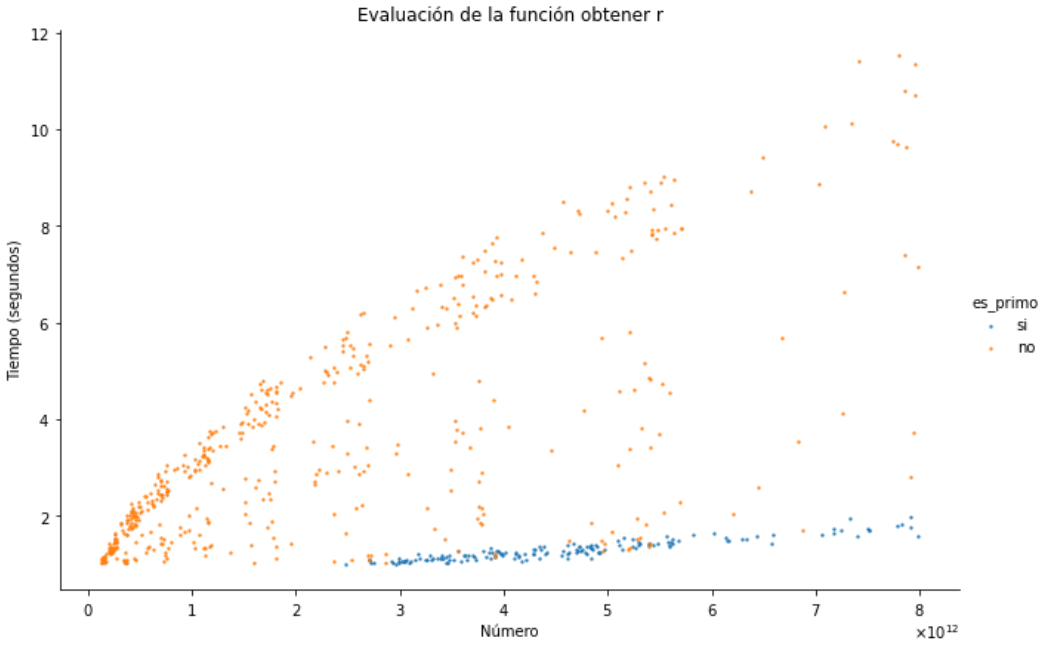
ESTUDIO EMPÍRICO

En este caso, se realizará un análisis entre el número y el tiempo de evaluación de obtener el menor valor de r que cumpla $O_r(n) > \log_2 2\,(n)$.

Debido al abundante ruido, en este caso solo se consideran los valores cuyo tiempo de ejecución supera el segundo. En azul se representan los números primos y en rojo los compuestos.

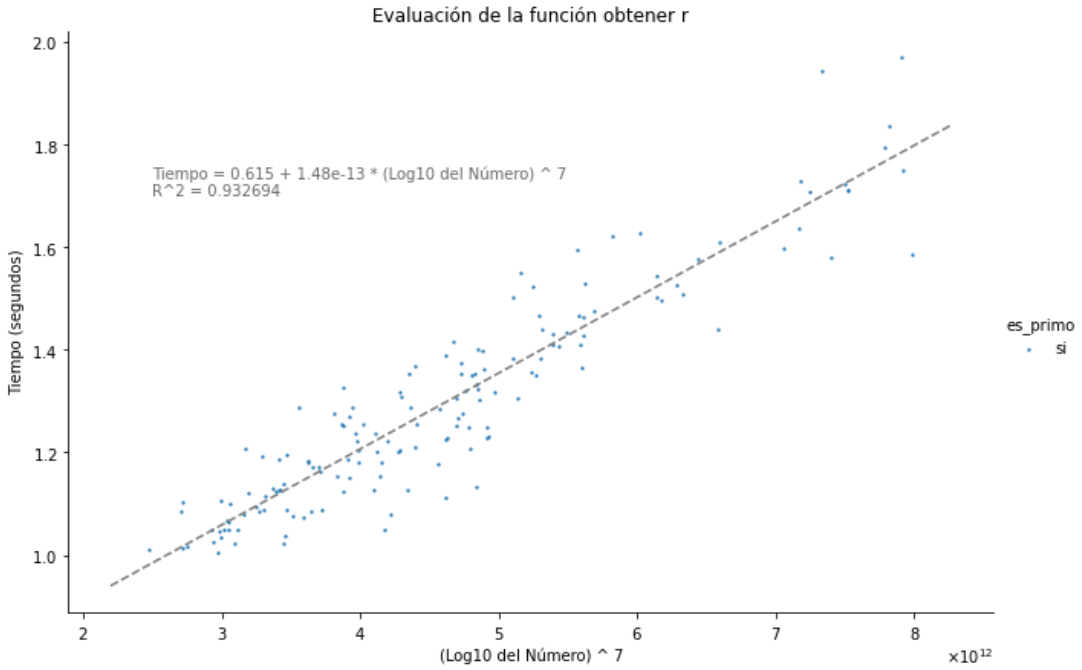
	Número	Tiempo
Total	638	638
Media	2907216238513 ≈ 2'91 * 10^{12}	3'07 s
Min	132059966735 ≈ 1'32 * 10^{11}	1'00 s
Max	7988632983827 ≈ 7'98 * 10^{12}	11'53 s

	Primalidad
Total	638
Valores únicos	2
si	154
no	484



El resultado es un modelo cuya complejidad se ajusta a un modelo logarítmico o de raíz cuadrada. Aunque como ya hemos visto en el análisis analítico el ajuste de complejidad $O\left((\log(\mathrm{n}))^7\right)$ es el que compararemos con nuestros resultados.

Debido al enorme ruido, reducimos la base de datos aquellos casos que son números primos, y, para consolidar el análisis hecho, aplicamos una transformación al eje x consistente en la complejidad predicha, lo que arroja los siguientes valores:



Obtenemos un valor de R^2 que es o suficiente cercano a 1 como para poder concluir que la relación actual es lineal. Esto implica que la relación de la gráfica anterior representa una relación de complejidad $O\left(\left(\log(\mathrm{n})\right)^7\right)$. Correspondiendo con lo obtenido en el análisis analítico.

EUCLIDES

La comprobación de $mcd(n,r)$ distinto de 1 se realizará con el algoritmo de Euclides. Este algoritmo es un método eficiente para encontrar el mayor común divisor de dos números enteros.

ESTUDIO ANALÍTICO

Para el estudio de este algoritmo analizaremos la peor instancia, siendo esta el caso de comparar dos números consecutivos en la sucesión de Fibonacci. Analizando el algoritmo vemos que se detiene cuando el segundo valor dado en este caso r llega a 1. Mediante la inducción podemos verificar esto:

Teniendo un caso base:

- Los valores de n=2 y r=1. Entonces $mcd(2,1)$ se reduce a $mcd(1,0)$ en solo un paso. Siendo N=1.

- Esto implica que n es por lo menos $f_{(N+2)}$ y b es $f_{(N+1)}$

Asumiendo que esto se cumple para hasta (N-1).

$$r \geq f_{(N-1+2)} => r \geq f_{(n+1)}$$

$$n \geq f_{(n+1)} + f_n$$

Esto se asemeja a la sucesión de Fibonacci y para determinar el número de iteraciones del bucle es necesario conocer el índice del término de esta sucesión. Para ello se utiliza la siguiente fórmula:

$$f_n = \frac{(\varphi^n - (1-\varphi)^n)}{\sqrt{5}}$$

Dado que $(1-\varphi)^n$ siempre tiene un valor absoluto menor que 1 y va cambiando de signo de tal forma que compensa la parte irracional del primer sumando, la fórmula puede escribirse de la siguiente forma:

$$f_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

Una vez hallada la fórmula anterior, para conocer la complejidad del cálculo en función de n se toman logaritmos en base φ, ya que la sucesión de Fibonacci se incrementa exponencialmente en magnitud:

$$\log_\varphi f_n = \log_\varphi \left(\frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right)$$

Eliminando los términos que no determinan la complejidad del algoritmo se obtiene como resultado:

$$\log_\varphi f_n = \log_\varphi(\varphi^n)$$

$$\log_\varphi f_n = n$$

Por lo tanto, concluimos que la complejidad del cálculo de $mcd(n, r)$ es $O(\log(n))$.

Reuniendo lo analizado en la heurística 2, que implica calcular el máximo común divisor de r números. Cada cálculo del máximo común divisor requiere un tiempo $O(\log(\mathrm{n}))$, por lo tanto, la complejidad de este paso es $O\big(\mathrm{r} * \log(\mathrm{n})\big) = O((\log(\mathrm{n}))^6)$.

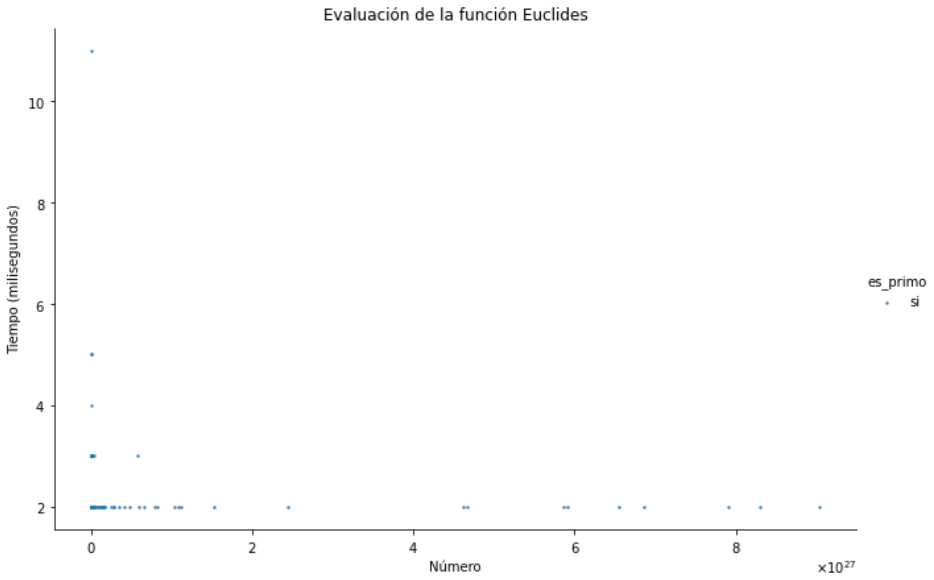
Por último, se comparará $n \leq r$, que tienen una complejidad $O(\log(\mathrm{n}))$.

ESTUDIO EMPÍRICO

El tamaño de los números se hace absurdamente grande si queremos obtener un tiempo significativo, es imposible de calcular, ya que, para estudiar la complejidad del

algoritmo euclídeo en función de n, previamente debemos realizar el cálculo del mínimo r asociado a ese número, lo que presenta una complejidad muy superior, e impide que podamos llegar a la dimensión requerida.

El siguiente gráfico muestra los resultados obtenidos cuando se ha evaluado el número con su correspondiente r mínima.



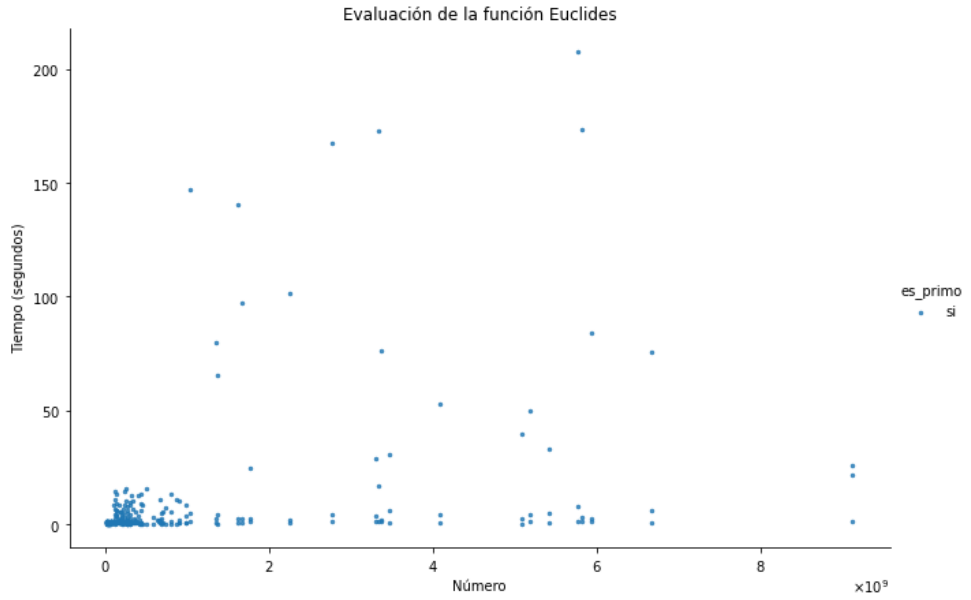
Como se puede ver, los resultados no son concluyentes y a pesar del elevado tiempo de ejecución no se logra ningún tiempo cercano a la magnitud del segundo.

Por ello hemos tratado de representar la complejidad a partir de dos números primos aleatorios, sin que uno sea el r mínimo del otro. Un breve resumen univariante de los datos con los que se ha trabajado es el siguiente:

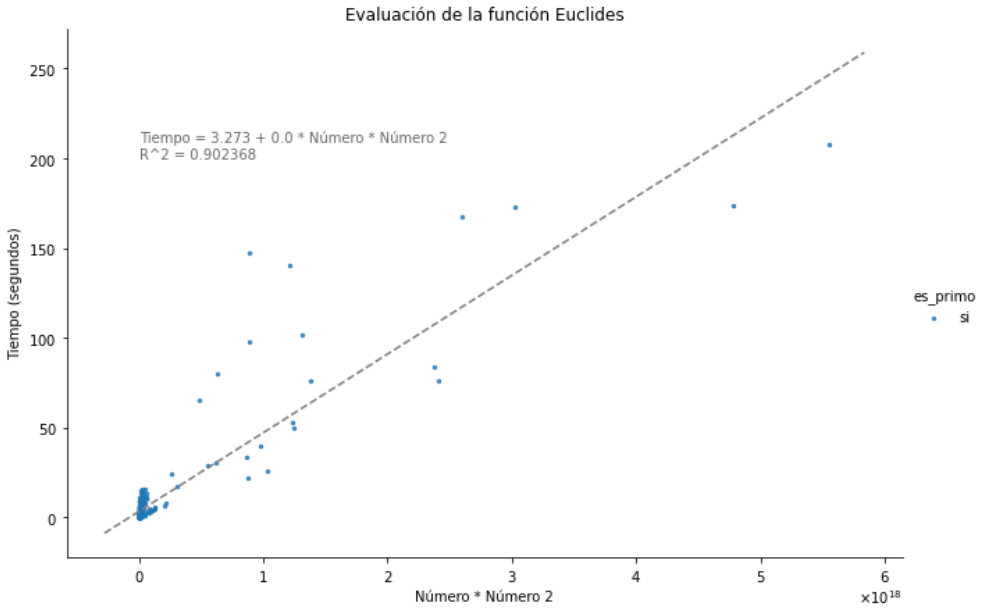
	Número	Número 2	Tiempo
Total	258	258	258
Media	1135250519 ≈ 1’13 * 10^9	53468600 ≈ 5’35 * 10^7	9’80 s
Min	10385447 ≈ 1’87 * 10^7	1342501 ≈ 1’34 * 10^6	0’21 s
Max	9125235277 ≈ 9’13 * 10^9	962610503 ≈ 9’63 * 10^8	207’66 s

	Primalidad
Total	258
Valores únicos	1
si	258

El siguiente gráfico muestra lo obtenido:



De nuevo, los resultados son muy poco concluyentes al representar el tiempo de ejecución frente al número más grande de los dos. Como prueba adicional, se ha probado a realizar la multiplicación de los dos números de entrada a la función, el resultado ha sido el siguiente:



No parece descabellado afirmar la linealidad de la función cuando se considera como variable independiente el producto de las entradas, aunque en este caso, se sale de los objetivos del estudio. Pese a ello, se ha logrado representar cierta lógica en la complejidad de la función euclídea.

CÁLCULO DE TOTIENT

Para el cálculo de Totient se ejecutan dos bucles anidados, que son los que determinan la complejidad de este cálculo y en lo que se va a centrar el siguiente estudio analítico:

ESTUDIO ANALÍTICO

```java
public static BigInteger totient(BigInteger n) {
    BigInteger result = n;
    for (BigInteger i = BigInteger.valueOf(2); n.compareTo(i.multiply(i)) > 0; i = i.add(BigInteger.ONE)) {
        if (n.mod(i).compareTo(BigInteger.ZERO) == 0)
            result = result.subtract(result.divide(i));

        while (n.mod(i).compareTo(BigInteger.ZERO) == 0)
            n = n.divide(i); }
    if (n.compareTo(BigInteger.ONE) > 0)
        result = result.subtract(result.divide(n));
    return result;
}
```

$for\ (i = 2; i < \sqrt{r}; i++)$

$while\ (r\ mod(i) = 0)$

La complejidad del primer bucle *for* es: $\lceil \sqrt{r} - 2 \rceil$ pues, se ejecutará siempre que la raíz cuadrada de r sea mayor que i, que aumenta de uno en uno y que en el momento inicial tienen un valor de 2.

En cuanto al segundo bucle, se cumple la condición para permanecer en él siempre que $r\ mod(i) = 0$, por lo tanto, para que entre en un primer momento en el bucle r debe ser múltiplo de i y, se ejecutará el bucle después si al dividir r entre i este resultado sigue siendo múltiplo de m. Por lo tanto, el peor caso se da cuando se cumple que $r = i^k$ siendo k cualquier número entero positivo.

La complejidad para el peor caso es: $\log_i r$ que para lo anteriormente definido el resultado es k.

Por lo tanto, la complejidad total de este cálculo es:

$$O\left(\sqrt{r}\log(r)\right)$$

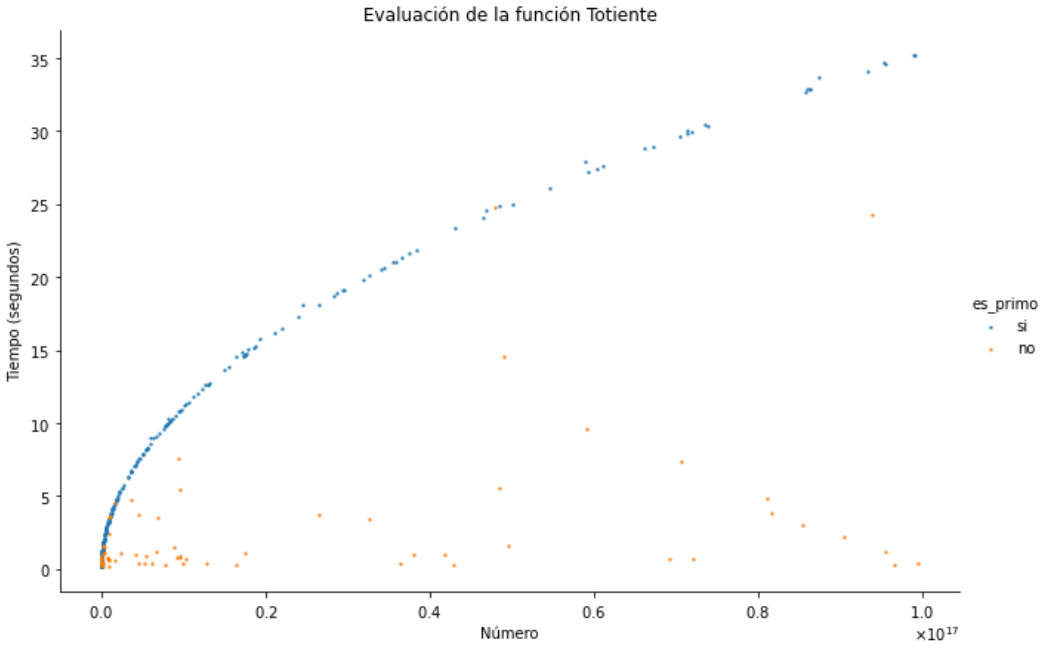
ESTUDIO EMPÍRICO

En este caso se realizará un análisis entre el tamaño del número y el tiempo de evaluación de la función Totient.

El corte a 0,2 s de mínimo reduce la base de datos de casos 1820 a 390 casos. En azul se representan los números primos y en rojo los compuestos.

	Número	Tiempo
Total	390	390
Media	12469113205935756 ≈ 1’25 * 10^{16}	6’11 s
Min	3152088310489 ≈ 3’15 * 10^{12}	0’201 s
Max	99582588993056288 ≈ 9’96 * 10^{16}	35’23 s

	Primalidad
Total	390
Valores únicos	2
si	320
no	70

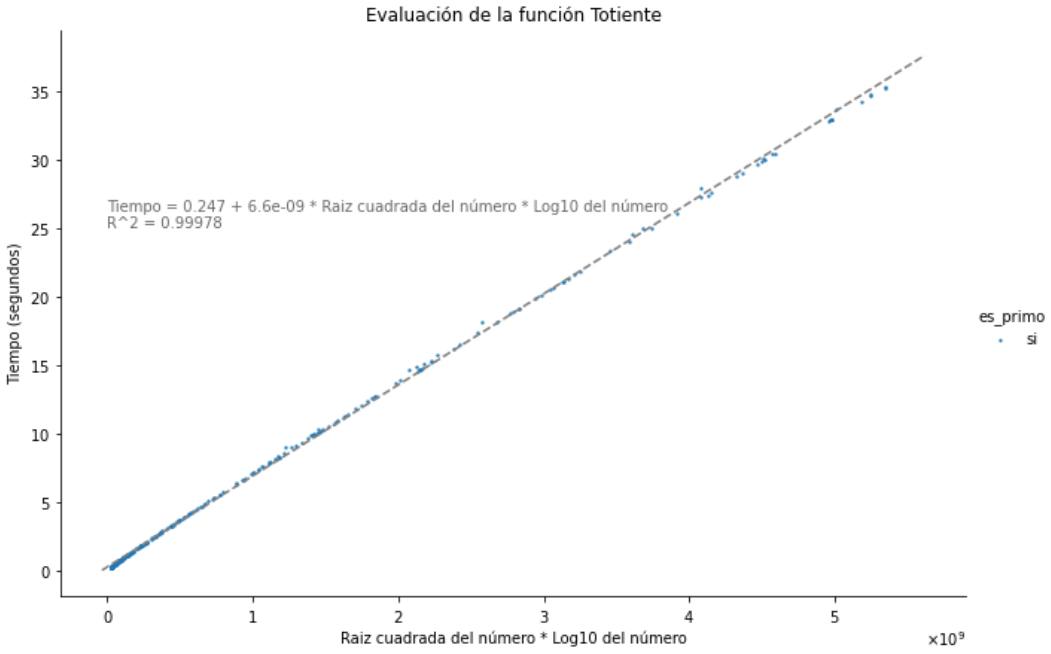


El resultado es un modelo cuya complejidad se ajusta a un modelo logarítmico o de raíz cuadrada. Aunque como ya hemos visto en el estudio analítico el ajuste de complejidad $O\left(\sqrt{r}\log(r)\right)$ es el que compararemos con nuestros resultados.

En esta gráfica se puede observar que los datos de los números compuestos son muy dispersos, visualmente no se observa ninguna relación. Lo que cuadra con la naturaleza de la función.

Dado que el caso interesante es el peor, y en este caso es el de los primos, reducimos el conjunto de valores a solo primos y, con el objetivo de consolidar el estudio analítico, aplicamos una transformación al eje de abscisas correspondiente en la multiplicación de la raíz del número por el logaritmo del número.

Además, añadiremos el modelo lineal con el parámetro R^2 queda de la siguiente forma:



Obtenemos un valor de R^2 muy cercano 1, por esto podemos afirmar con nivel de incertidumbre muy bajo que la relación actual es lineal. Esto implica que la relación de la gráfica anterior representa una relación de complejidad $O\left(\sqrt{r}\log(r)\right)$ Confirmando lo obtenido en el estudio analítico.

CONDICIÓN SUFICIENTE

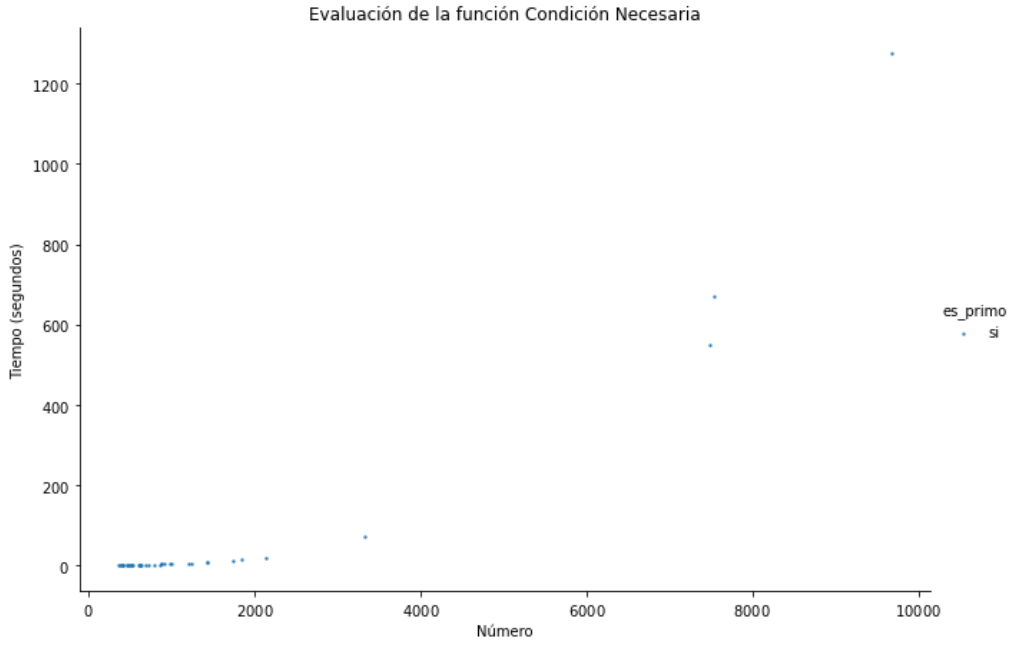
En este caso, hay que verificar $\lfloor\sqrt{totient(r)}\log(n)\rfloor$ ecuaciones. A su vez cada ecuación requiere $O(\log(n))$ multiplicaciones de polinomios de grado r con coeficientes de tamaño $O(\log(n))$. Por lo tanto, la complejidad de este paso es $O\sim(r^{\frac{3}{2}}(\log(n))^3 = O\sim(\log(n))^{\frac{21}{2}}$.

ESTUDIO EMPÍRICO

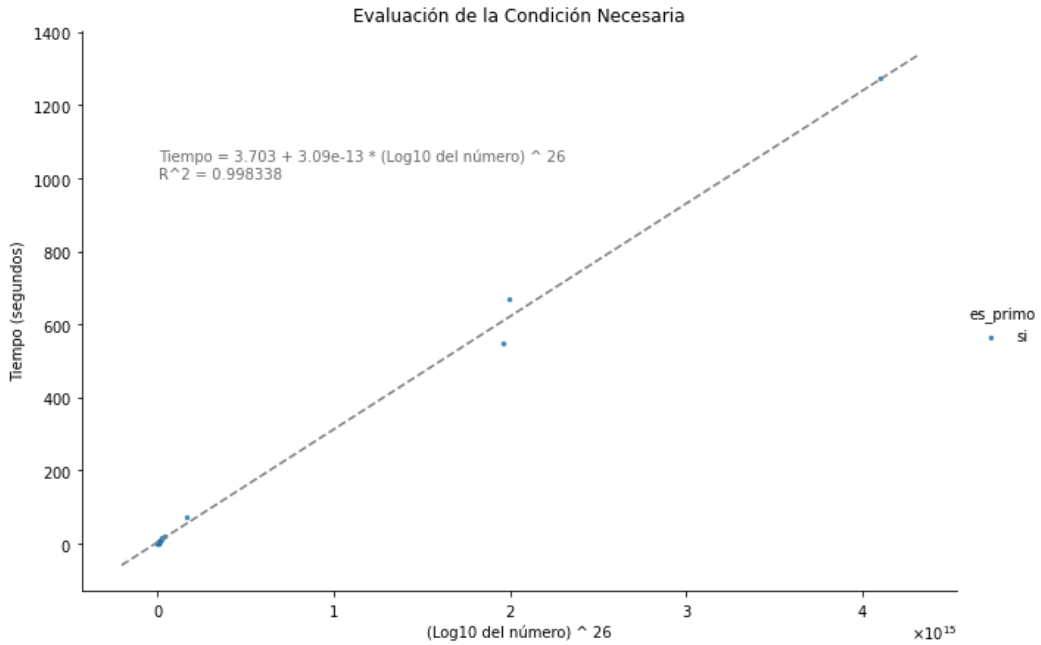
En esta función, los tiempos se disparan rápidamente, por ello el conjunto de datos es más reducido. Disponemos de 83 casos, de los cuales 80 verifican la condición de significatividad que hemos estipulado para el tiempo (superior a dos décimas de segundo). Un breve estudio univariante de los datos arroja los siguientes resultados:

	Número	Tiempo
Casos	80	80
Media	417324592411862 ≈ 4'17 * 10^{14}	6'11 s
Min	580860843655 ≈ 5'81 * 10^{11}	0'201 s
Max	4104263102518229 ≈ 4110 * 10^{15}	35'23 s

	Primalidad
Total	390
Valores únicos	1
si	80



Los valores crecen a una gran velocidad, en este caso realizamos la transformación predicha al eje de abscisas y el resultado no fue el esperado, quedando lejos de la linealidad en la gráfica. Se puede deber a la variación de versiones de AKS y/o factores no considerados. Por ello, se ha realizado un ajuste plenamente empírico hasta lograr, la transformación que arroje mejor resultado, manteniendo coherencia con la complejidad estimada. Finalmente, elevar a 26 el logaritmo del número es la solución escogida:



La regresión es capaz de explicar la relación en un 99'83%, un dato realmente bueno, por lo que se concluye de este ajuste que la complejidad de la función es $O\sim(\log(n))^{26}$. Aunque somos conscientes de que el valor del exponente puede no estar ajustado a la perfección con la realidad.

COMPLEJIDAD DEL ALGORITMO AKS

Tras el estudio de las partes que forman el algoritmo AKS, se obtiene que la complejidad total del algoritmo AKS, es la misma que la alcanzada en la parte de condición suficiente ya que esta domina a todas las demás. Por lo tanto, la complejidad total del algoritmo es:

$$O\sim(\log(n))^{\frac{21}{2}}$$

CONCLUSIONES

Tras la realización de la práctica podemos concluir que ha sido realmente instructiva para ampliar nuestros conocimientos tanto en el análisis de algoritmos como en el uso de herramientas útiles para ello. No obstante, al principio surgieron ciertos problemas con respecto al estudio analítico y más adelante también con la relación entre los resultados obtenidos en ambos estudios, en el analítico y en el empírico.

Por otro lado, nos ha sido de gran ayuda tanto la bibliografía recomendada como las recomendaciones y explicaciones recibidas en clase.

Por todo esto, pese a las adversidades encontradas en el camino, el balance ha sido positivo. Aunque hubiera sido de mucha ayuda obtener una corrección de las entregas parciales para reconsiderar nuestros cálculos o conclusiones, sin embargo, entendemos que no haya sido posible.

BIBLIOGRAFÍA

[1] M. Agrawal, N Kayal, N. Saxena, "PRIMES is in P". Annals of Mathematics, no. 160, 781-793, 2004.

DANIEL J. BERNSTEIN, 1998. Detecting perfect powers in essentially linear time. *Mathematics of computation,* **67**(223), pp. 1253-1283.

DE JESÚS, J., ANGEL, A. and MORALES-LUNA, G., *El algoritmo de Agrawal, Kayal y Saxena para decidir primalidad.*

LINKS DE APOYO

https://www.javaer101.com/en/article/1386371.html