PRÁCTICA 1

Entrega final

Teoría avanzada de la computación

Realizado por:

Alejandro Díaz García; NIA: 100383181

Daniel Romero Ureña; NIA: 100383331

Marcelino Tena Blanco; NIA: 100383266

Índice

Introducción

En este documento se documentan los análisis realizados tanto de forma empírica como de forma analítica de la implementación del algoritmo AKS, el cual se encarga de realizar un test de primalidad. Los análisis se van a desarrollar por partes que son las que componen el algoritmo, para finalmente conseguir obtener los costes computacionales globales del algoritmo. Con el fin de obtener una aproximación lo más exacta posible de la complejidad del algoritmo se realizan dos tipos de análisis, una de forma analítica analizando el código de la implementación del algoritmo para así obtener el coste computacional y por otra parte en el análisis empírico vamos analizando los tiempos que dedica cada una de las partes en ejecutar el algoritmo.

Análisis analítico

Funciones auxiliares

En este apartado se analizará la complejidad de los métodos auxiliares log y multiplicativeOrder usadas por las dos heurísticas con el fin de conocer el valor de dicha complejidad cuando alguna de estas dos funciones sea llamada por estas.

```
  T(Aux) = 5 + max(T(A),T(B)) = 5 + 9= 14
private double log(BigInteger x) {
    BigInteger b;                                                  // 1
    int temp = x.bitLength() - 1000;                               // 4
    //T(A) = 1 + 2 + 6 = 9
    if (temp > 0) {                                                // 1
        b = x.shiftRight(temp);                                    // 2
        return (Math.log(b.doubleValue()) /
                Math.log(2.0D) + temp);                            // 6
    }
     //T(B) = 5
     else
        return (Math.log(x.doubleValue()) / Math.log(2.0D));       // 5
}
T(Aux2) = 3 + T(A2) + max(T(B2),T(C2)) = 3+9p+5+4=9p+12
public BigInteger multiplicativeOrder(BigInteger r) {
    BigInteger k = BigInteger.ZERO;                                // 2
    BigInteger result;                                             // 1
    //T(A2) = 4p + 5 + 5p = 9p + 5
    do {
        k = k.add(BigInteger.ONE);                                 // 2
        result = this.n.modPow(k, r);                              // 3
    }
    while (result.compareTo(BigInteger.ONE) != 0 &&
           r.compareTo(k) > 0);                                    //4p
    //T(B2) = 4
    if (r.compareTo(k) <= 0)                                       // 2
        return BigInteger.ONE.negate();                            // 2
    //T(C2) = 1
    else {
        return k;                                                  // 1
    }
}
```

Cálculo de p

El valor de la constante p independiente del valor numérico de entrada será igual al valor de r, ya que la condición del bucle establece que la condición de parada se dará cuando k es mayor que r, ya que de las dos condiciones de parada esta es la que se da en el peor caso.

Tras la obtención del valor de la constante p, se obtiene la siguiente complejidad para los segmentos de código:

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(Aux)	Rojo	5 + max(T(A),T(B) = 5 + 9	14	Constante
T(Aux2)	Azul	3 + T(A2) + max(T(B2),T(C2))	9r + 12	Lineal

ID	Complejidad en la expresión	Complejidad final
T(Aux)	$O(1)$	$O(1)$
T(Aux2)	$O(n) + O(1)$	$O(n)$

Heurística 1: Comprobación de si el número es una potencia perfecta.

En este bloque se analizará la complejidad presente en el segmento de código que abarca la heurística 1, la cual tiene el objetivo de comprobar si un determinado número es una potencia perfecta.

```
BigInteger base = BigInteger.valueOf(2);                    // 2
BigInteger aSquared;                                        // 1
do {      //T(A)=29+29n+2n+n*(T(B)+T(C))=31n+29+8mn+6n+4n=8ml+41l+29
    BigInteger result;                                      // 1
    int power = Math.max((int) (log() /
                        log(base) - 2), 1);                 //5+2T(Aux)
    int comparison;                                         // 1
  do {    //T(B) = 6m + 2m + 6 = 8m + 6
        power++;                                            // 2
        result = base.pow(power);                           // 2
        comparison = n.compareTo(result);                   // 2
   } while (comparison > 0 && power < Integer.MAX_VALUE);   // 2m
  //T(C) =  1 + 3 = 4
  if (comparison == 0){                                     // 1
        factor = base;                                      // 1
        n_isprime = false;                                  // 1
        return n_isprime;                                   // 1
    }
    base = base.add(BigInteger.ONE);                        // 2
    aSquared = base.pow(2);                                 // 2
} while (aSquared.compareTo(this.n) <= 0);                  // 21
```

Tras realizar el análisis mediante la segmentación del código en diferentes sub-bloques podemos concluir la siguiente complejidad para cada uno de cada uno de los sub-bloques como del coste del conjunto global:

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(n)	Blanco	3 + T(A) =3 + 8mn+41n+29	8ml+41l+32	?
T(A)	Amarillo	29+29n+2n+n*(T(B)+T(C))	8ml+41l+29	?
T(B)	Naranja	6m + 2m + 6	8m+6	?
T(C)	Rojo	1 + 1 + 1 + 1	4	Constante

Cálculo de m

La condición de parada dice que no se realiza el bucle hasta que la base elevada a la potencia dada sea mayor que el número. Entonces debemos tener en cuenta el valor inicial del valor del exponente y el valor máximo que puede llegar a ser. El valor máximo que puede llegar a tener el exponente es el techo de $log_b n$ y el valor en el que empieza el exponente es $\frac{log_2 n}{log_2 b} - 2$. También hay que tener en cuenta que el bucle se sale en el caso de que el número sea mayor que 2^31 -1. Por lo tanto:

$$m = min\{log_b n - \frac{log_2 n}{log_2 b} + 2,\ 2^{31} + 1 - \frac{log_2 n}{log_2 b}\}$$

Cálculo de n

Para conocer el número de veces que se realiza el bucle debemos tener en cuenta que la salida se obtiene cuando la base elevada a 2 es superior al número introducido. Por lo tanto, el valor máximo al que puede alcanzar la base es el suelo de raíz de n. Para saber las iteraciones totales conocemos que la base empieza en 2 y que el valor de la base se realiza antes de la comparación. Lo que significa que el valor total de iteraciones es: $\sqrt{n}$

Tras obtener los valores de las constantes n y m podemos concluir las siguientes complejidades para los bloques y sub-bloques del segmento del código:

ID	Color	Complejidad total	Tipo de complejidad
T(n)	Blanco	$32 + \sum_{b=0}^{\sqrt{n}} 41 + 8 * min\{log_b n - \frac{log_2 n}{log_2 b} + 2,\ 2^{31} + 1 - \frac{lo}{lo}$	Logarítmica
T(A)	Amarillo	$29 + \sum_{b=0}^{\sqrt{n}} 41 + 8 * min\{log_b n - \frac{log_2 n}{log_2 b} + 2,\ 2^{31} + 1 - \frac{l}{l}$	Logarítmica
T(B)	Naranja	$8 * min\{log_b n - \frac{log_2 n}{log_2 b} + 2,\ 2^{31} + 1 - \frac{log_2 n}{log_2 b}\} + 6$	Logarítmica
T(C)	Rojo	4	Constante

Según los datos obtenidos en la anterior tabla podemos concluir la siguiente complejidad en el segmento de código:

Complejidad en la expresión	Complejidad final
$O(1) + \sum_{b=0}^{\sqrt{n}} O(1) + O(1) * min\{O(log\ n) - \frac{O(log\ n)}{O(log\ n)} + O(1),\ O(1) - \frac{O(log}{O(log}$	$O(log\ n)$

Heurística 2: Cálculo de r

En este bloque se analizará la complejidad del bloque de código que abarca la parte del cálculo de r en la heurística dos, posteriormente se analiza el cálculo de mcd para esta heurística.

```
double log = this.log();                    // 1 + 14
double logSquared = log * log;              // 3
BigInteger k = BigInteger.ONE;              // 2
BigInteger r = BigInteger.ONE;              // 2
//T(A) = n * (2 + 3 + 9r + 5)+(2 + 3 + 9r + 5) = 10l+9rl+9r+10
do {
    r = r.add(BigInteger.ONE);              // 2
    k = multiplicativeOrder(r);             // 1 + T(Aux2)
}while (k.doubleValue() < logSquared);      // 2n
```

Tras el análisis de complejidad se obtienen los siguientes resultados:

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(n)	Blanco	T(A) + 22 = 10n+9rn+9r+10 + 22	10l+9rl+9r+32	?
T(A)	Amarillo	n * (2 + 3 + 9r + 5)+(2 + 3 + 9r + 5)	10l+9rl+9r+10	?

Cálculo de n

Tras realizar un análisis del método log y el propio segmento de código del cálculo de r, se puede concluir que n (número de iteraciones del bucle) viene dado por logSquared, siendo este el valor al cuadrado de la variable log, cuyo valor viene dado por la función auxiliar log(), la cual recibe como parámetro de entrada el número n (número de entrada del programa).

De este modo el valor más alto que este método podría devolver sería el dado por la fórmula 2log(n) / log(2), siendo el número que se tiene como parámetro en el programa.

Por lo tanto, el coste computacional final para este segmento de código vendría dado de la siguiente forma:

Cálculo de r

Para obtener r lo que hemos realizado es buscar en la documentación y hemos obtenido que $r \leq max\{3, log^5 n\}$[1]. Como $max\{3, log^5 n\}$es el límite superior, entonces podemos utilizar la salida de este valor como resultado de r, de forma que, aunque tengamos menos precisión, obtenemos un valor de r saturado.

ID	Color	Complejidad total	Tipo de complejidad
T(n)	Blanco	98*$max\{3,\ log^{5} n\}$*(2log(n) / log(2)) + 16((2log(n) / log(2))) + 24	Logarítmica
T(A)	Amarillo	9*$max\{3,\ log^{5} n\}$*(2log(n) / log(2)) + 16((2log(n) / log(2))) + 2	Logarítmica

Tras haber obtenido la expresión general del segmento de código podemos

Complejidad en la expresión	Complejidad final
$O(1)\ *\ max\{O(1)\ +\ O(log\ n)\}\ *\ (\frac{O(log\ n)}{O(1)} + O(1)\ *\ \frac{O(log\ n)}{O(log\ n)})\ +\ O(1)$	$O(log\ n)$

Heurística 2: Cálculo del MCD

En este apartado se analiza la complejidad del segmento de código perteneciente al cálculo del mcd para la heurística 2.

```
//T(A) = n * (4 + 2 + T(B)) + 4 = 11n + 4
for (BigInteger i = BigInteger.valueOf(2); i.compareTo(r) <= 0; i =
     i.add(BigInteger.ONE))
{                                                                    // 4l + 4
    BigInteger gcd = n.gcd(i);                                       // 2
    //T(B) = 2 + 3 = 5
    if (gcd.compareTo(BigInteger.ONE) > 0 && gcd.compareTo(n) < 0) {  // 2
        factor = i;                                                  // 1
        n_isprime = false;                                           // 1
        return false;                                                // 1
    }
}
//T(C) = 1 + 2 = 3
if (n.compareTo(r) <= 0) {                                           // 1
    n_isprime = true;                                                // 1
    return true;                                                     // 1
}
```

Tras realizar el estudio de complejidad se obtienen los siguientes resultados divididos tanto de los sub-bloques como el coste general del segmento de código:

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(l)	-	T(A) + T(B) + T(C)	11l + 12	?
T(A)	Amarillo	n * (4 + 2 + T(B)) + 4	11l + 4	?
T(B)	Naranja	2 + 3	5	Constante
T(C)	Rojo	1 + 2	3	Constante

Cálculo de n

Como se puede apreciar, el bucle for hace uso de un contador que va de 1 en 1 hasta que se llegue al valor de r, empezando este contador desde el valor 2, por lo tanto se puede concluir que n es igual a (r-2). De este modo al ser el valor r lineal, podemos deducir que los segmentos de código cuya complejidad es de orden n tienen un tipo de complejidad lineal.

Una vez realizadas estas deducciones podemos concluir las siguientes complejidades:

ID	Color	Complejidad total	Tipo de complejidad
T(n)	-	11($max\{3,\ log^{5} n\}$-2) + 12	Logarítmico
T(A)	Amarillo	11($max\{3,\ log^{5} n\}$-2) + 4	Lineal
T(B)	Naranja	5	Constante
T(C)	Rojo	3	Constante

Mediante la expresión deducida en la tabla anterior podemos obtener la complejidad final del segmento de código:

Complejidad en la expresión	Complejidad final
$O(1)\ *\ (\ max\{\ O(1)\ ,\ O(log\ n)\}\ -\ O(1)\ )\ +\ 12$	$O(log\ n)$

Cálculo de Totient

```
//T(O) = 2+1+T(A)+T(D)= 3+5mp+14p+7+5 = 5mp+14p+15
BigInteger totient(BigInteger n) {
        BigInteger result = n;                                          // 2
        // T(A) = 5p+7+p*(T(B)+T(C)) = 5p+7+6p+5mp+3p = 5mp+14p+7
        for (BigInteger i = BigInteger.valueOf(2); n.compareTo(i.multiply(i))
                            > 0; i = i.add(BigInteger.ONE)) {           //5p+7
            // T(B) = 3 + 3 = 6
            if (n.mod(i).compareTo(BigInteger.ZERO) == 0)               // 3
                result = result.subtract(result.divide(i));             // 3
            // T(C) = 3m+3+2m = 5m + 3
            while (n.mod(i).compareTo(BigInteger.ZERO) == 0)            // 3m+3
                n = n.divide(i);                                        // 2
        }
        // T(D) = 5
        if (n.compareTo(BigInteger.ONE) > 0)                            // 2
            result = result.subtract(result.divide(n));                 // 3
        return result;                                                  // 1
}
```

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(O)	Blanco	2+1+T(A)+T(D)= 3+5mp+14p+7+5	5mp + 14p + 15	?
T(A)	Amarillo	5p(T(B) + T(C)) + 5 + 2	(5m + 14)p + 7	?
T(B)	Rojo	3 + 3	6	Constante
T(C)	Azul	3m + 3 +2m	5m + 3	?
T(D)	Naranja	2 + 3	5	Constante

Cálculo de m

En el caso de m el valor puede deducirse mediante la suposición de que el coste por iteración del bucle viene dado por la expresión $log_{P1}n + log_{P2}n + \ldots + log_{Pk}n$ siendo P el valor del número primo correspondiente a los divisores por interacción y n siendo el valor

del número de entrada a obtener sus divisores, por lo tanto se puede deducir la siguiente expresión: $k\ log(n)$.

En el caso de generalizar la expresión mostrada previamente, asumiendo que el valor máximo de k viene dado por la expresión $\frac{1}{log\sqrt{n}}$ se puede deducir la siguiente expresión de complejidad para la variable m: $\frac{log(n)}{log\sqrt{n}}$.

Cálculo de p

El valor de p es igual a $\sqrt{n}$ debido a la condición establecida en el bucle for mediante la comprobación de n.compareTo(i.multiply(i)) > 0, siendo el valor del número introducido por parámetro de entrada.

Por lo tanto, **tras obtener los valores de las variables m y p** podemos deducir la siguiente complejidad final para el sub-bloque de código:

ID	Color	Complejidad del segmento de código	Complejidad total	Tipo de complejidad
T(O)	Blanco	2+1+T(A)+T(D)= 3+5mp+14p+7+5	$5\frac{log\ n}{log\sqrt{n}}\sqrt{n}\ +\ 14\sqrt{n}\ +\ 15$	Lineal
T(A)	Amarillo	5p(T(B) + T(C)) + 5 + 2	$5\frac{log\ n}{log\sqrt{n}}\sqrt{n} + 14\sqrt{n}\ +\ 7$	Lineal
T(B)	Rojo	3 + 3	6	Constante
T(C)	Azul	3m + 3 +2m	$5\frac{log\ n}{log\sqrt{n}}\ +\ 3$	Logarítmica
T(D)	Naranja	2 + 3	5	Constante

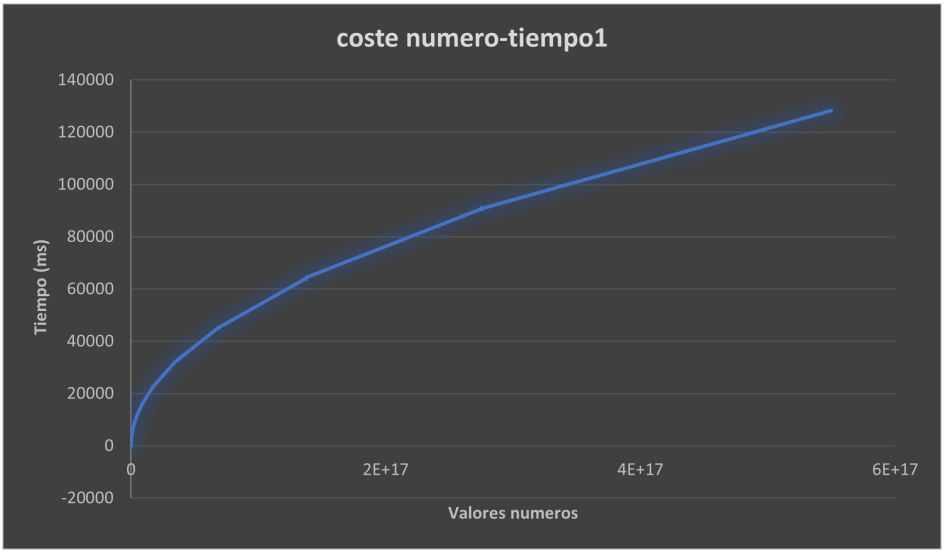
Complejidad en la expresión	Complejidad final
$O(1)\ *\ \frac{O(log\ n)}{O(log\sqrt{n})} + O(\sqrt{n})\ +\ O(1)\ *\ O(\sqrt{n})\ +\ O(1)$	$O(\sqrt{n})$

Análisis empírico

A continuación se va a realizar el análisis empírico de las dos heurísticas vistas, en concreto se van a tomar los tiempos de ejecución de diferentes números y para ver su correspondiente complejidad computacional ante el crecimiento de números.

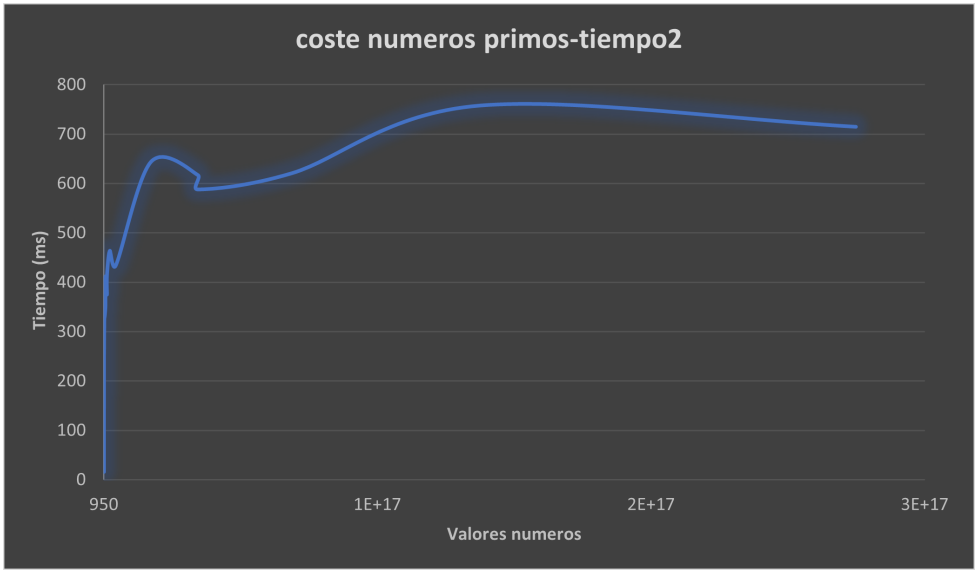
Heurística 1: Comprobación de si el número es una potencia perfecta.

Para comprobar los tiempos de ejecución de forma empírica en dicha heurística hemos colocado funciones que nos devuelven el tiempo en el momento en el que se ejecutan, para luego calcular la diferencia y obtener lo que ha tardado en ejecutarse. Como se puede apreciar en la siguiente gráfica la tendencia es alcista, en concreto se puede decir que la complejidad asociada es logarítmica dado el crecimiento que sigue O(log n), de forma aproximada. Cada vez que los números van creciendo de orden se va aumentando el coste de tiempo en la comprobación de si es una potencia perfecta.

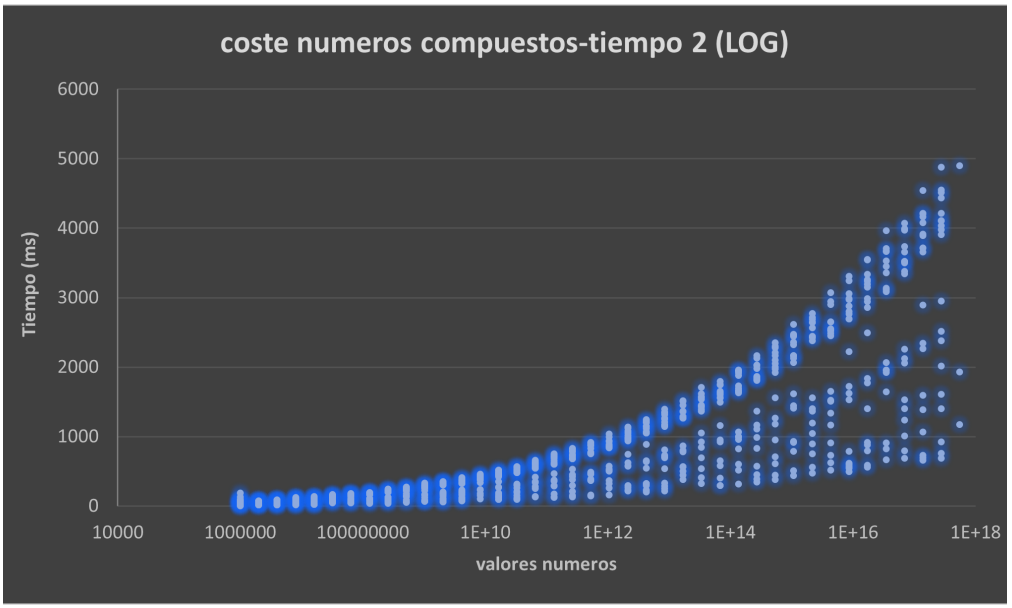


Heurística 2: Cálculo de r

Para comprobar el cálculo del parámetro r se ha realizado de la misma manera que en la heurística 1, tomando los tiempos en los momentos determinados para luego lograr sacar la diferencia. Con los datos obtenidos hemos podido sacar las siguientes gráficas tras realizar un análisis de las mismas. Al principio con todo el conjunto de números y sus correspondientes tiempos al calcular r nos salía una gráfica en la cual no se podía apreciar ningún tipo de tendencia y ni siquiera una aproximación a algún tipo de función de complejidad. Por ello decidimos dividir las gráficas en dos, una con los números primos para ver si en dicho caso existe alguna tendencia y otra con los números compuestos restantes.
Al obtener la gráfica correspondiente a los números primos nos encontramos con una tendencia logarítmica, pero hay que tener en cuenta que esto solamente sería a la hora de calcular r con números primos.

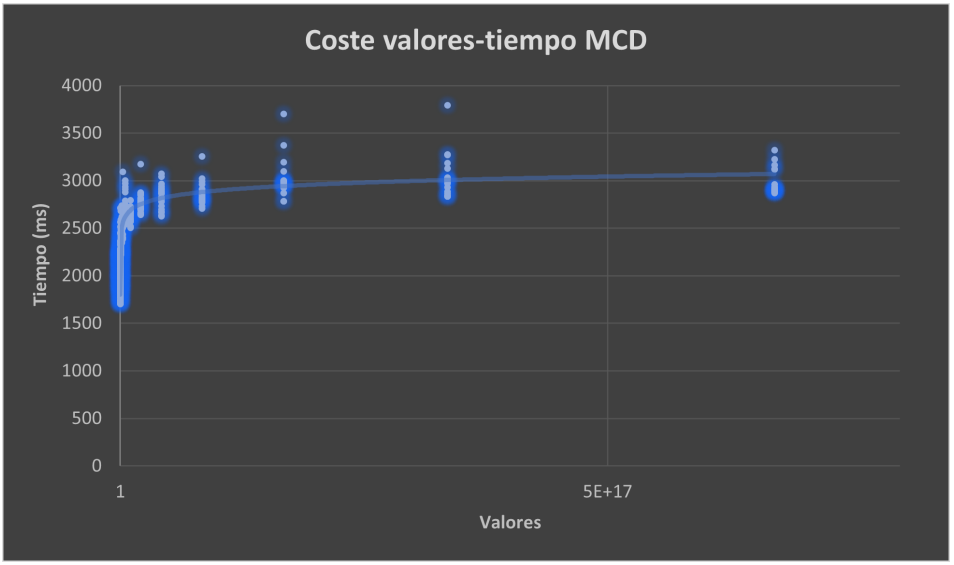


Por otro lado, la gráfica de los números compuestos no nos aportaba ningún tipo de información ni indicios de tener alguna tendencia en su complejidad, al igual que la primera gráfica que obtuvimos con todos los números que se computaron con el algoritmo. Por lo tanto decidimos ponerla en escala logarítmica con el fin de ver si existía algún tipo de tendencia en cuanto al orden de los números compuestos y para nuestra sorpresa pudimos ver la siguiente gráfica. Con esta gráfica podemos apreciar que dependiendo del valor del número compuesto independientemente de su orden va a tardar más o menos tiempo. En lo general, tras computar una gran magnitud de números superior a la que se muestra en esta gráfica podemos afirmar que la tendencia es exponencial al margen de aquellos valores que tardan menos, y en dichos casos las complejidades serían diferentes llegando a ser lineales en algunas ocasiones. En conclusión dependiendo de los números a evaluar se puede llegar a obtener una complejidad u otra dependiendo de lo que le cueste calcular r.



Heurística 2: Cálculo del MCD

En el cálculo del MCD nos encontramos que la complejidad es constante a partir del orden de elevado a 14, mientras que en valores que son de un orden inferior a este nos encontramos una complejidad que varía en función del número al que es necesario computar el MCD.

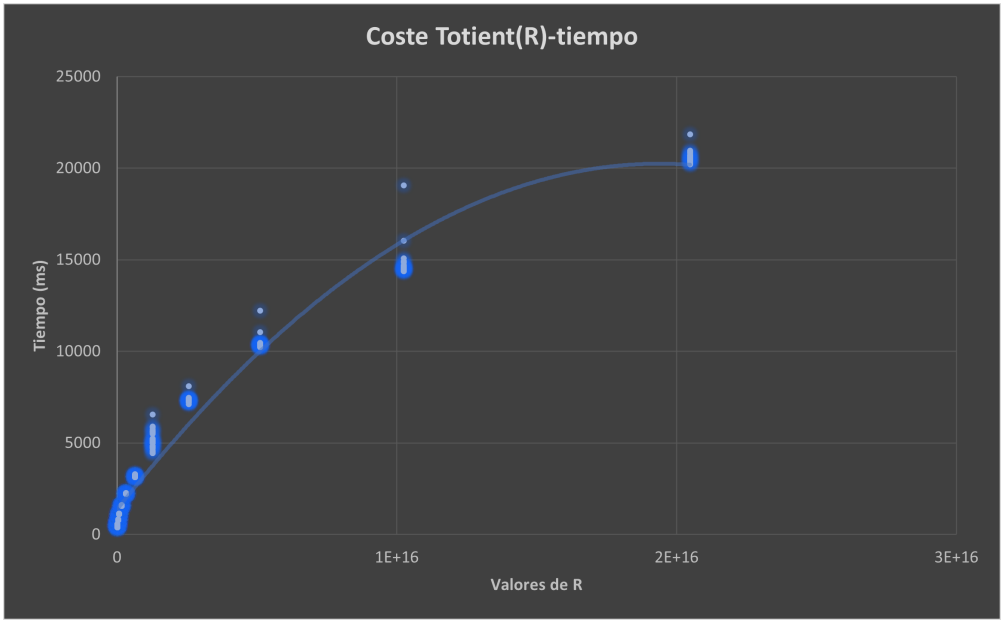


En las siguientes gráficas podemos apreciar que en valores de orden menor a 14 no es constante, esto puede darse por ruido producido por otros procesos del sistema operativo que estén afectando a la ejecución, pero asumimos que es constante en torno a 0.

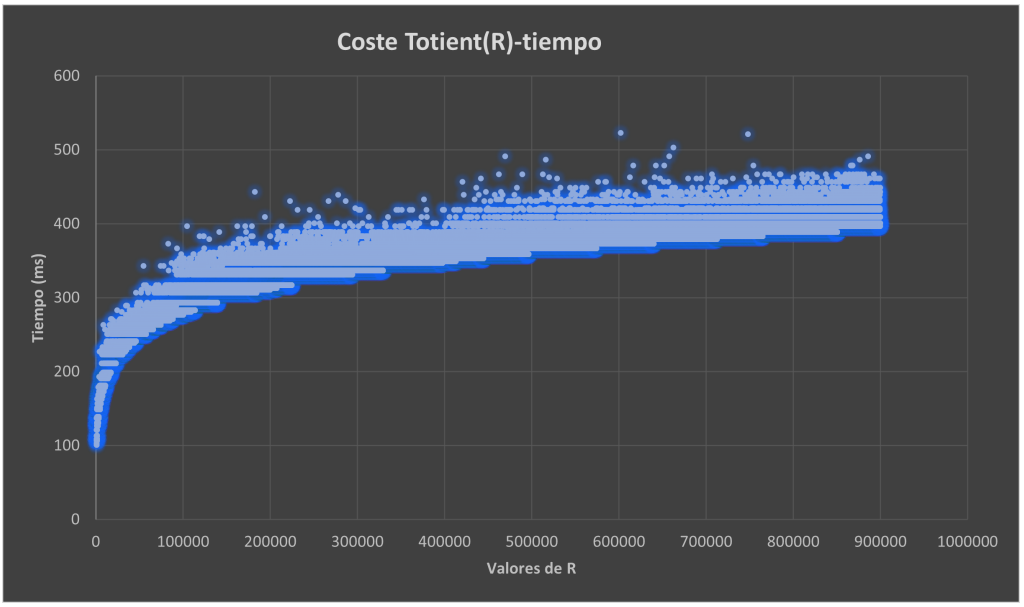
Cálculo de Totient

A continuación se va a analizar la complejidad algorítmica en cuanto al tiempo de ejecución en la función del cálculo de Totient. Sabiendo que la función Totient se basa en calcular el número de primos de 1 a n, esperamos que con los números que son primos los valores sean bastante superiores a la de los valores compuestos. Por lo tanto, al no tener demasiado interés en ver los valores compuestos de Totient, solo vamos a evaluar los números que son primos dentro de n.

Gráfica con todos los valores obtenidos de n, siendo n=r.



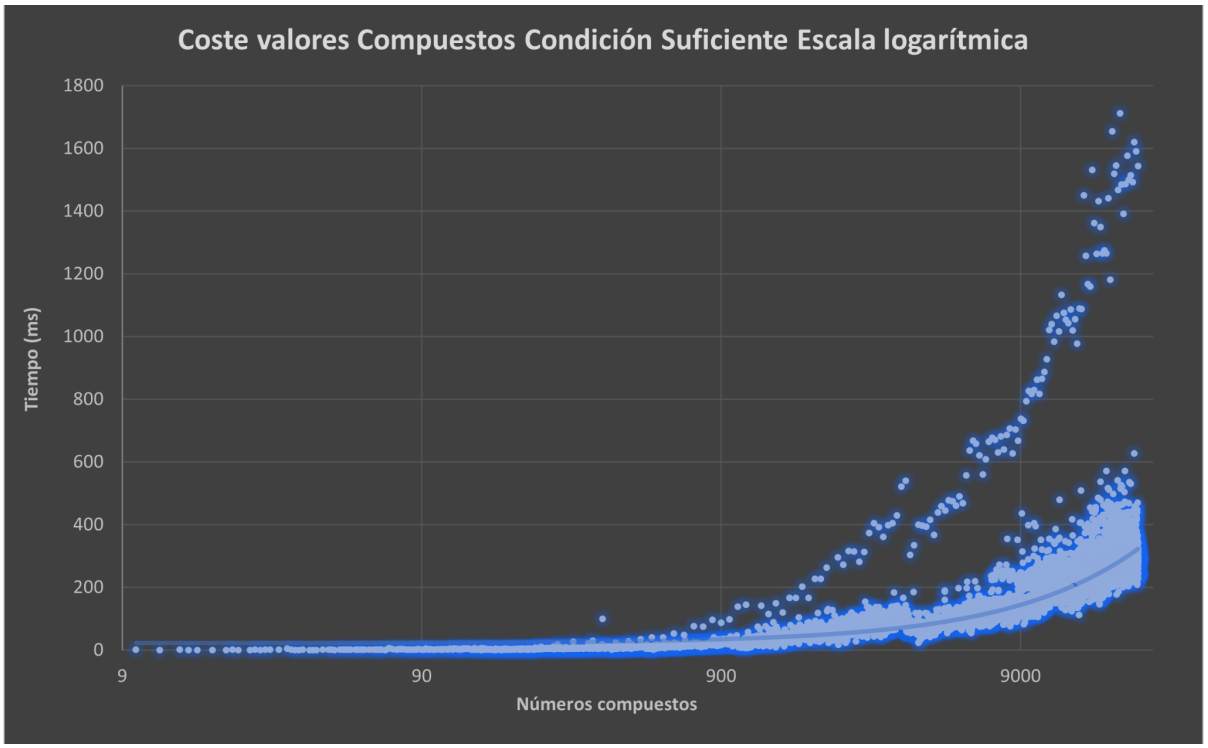
En esta otra gráfica se puede observar para valores más pequeños.



Como se puede observar la tendencia de las gráficas concuerda con la de una función logarítmica. Por lo tanto la complejidad de la función del cálculo de Totient mediante el método empírico, concluye en que es de complejidad logarítmica.

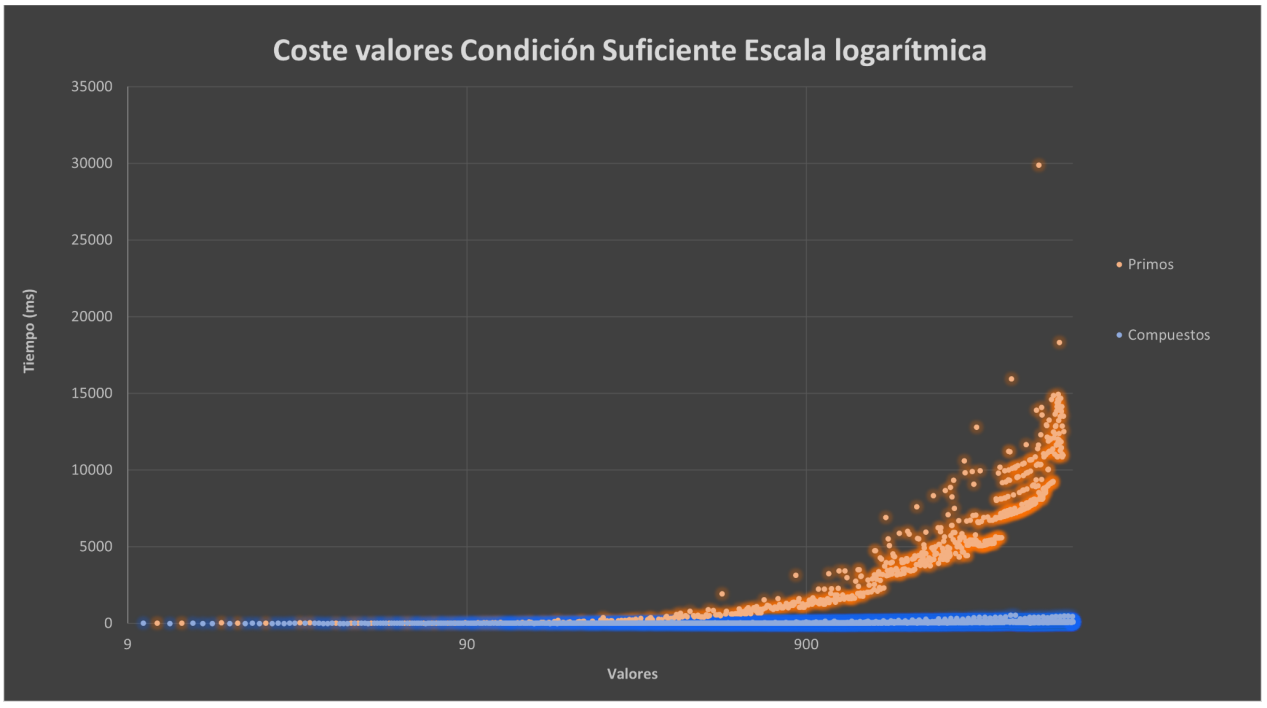
Determinación de la primalidad mediante el análisis de la condición suficiente

Este bloque procederá a realizar un análisis empírico del segmento de código del algoritmo AKS. Para ello se han ejecutado valores compuestos durante una hora con el fin de ver la tendencia que sigue en cuanto a costes temporales cuando se trata de calcular esta serie de valores. Por otro lado también se ha realizado una ejecución de una hora con números Primos, con los valores obtenidos de las ejecuciones podemos sacar las siguientes gráficas y por lo tanto las siguientes conclusiones.



En la gráfica de valores compuestos podemos apreciar que la tendencia a seguir es logarítmica salvo una serie de valores atípicos que nos podemos encontrar por encima de dónde estaría dicha línea de tendencia. Tras la hora de ejecución podemos ver que se han obtenido números compuestos de magnitudes como 22.000, y unos tiempos que en promedio tardan unos 200 ms en obtenerse el resultado.

En la gráfica correspondiente a los valores primos, nos podemos dar cuenta de que las magnitudes de tiempo son mayores y los valores alcanzan una menor magnitud que los obtenidos en los números compuestos durante la ejecución del algoritmo. Esto es debido principalmente a que es mucho más rápido para el algoritmo de condición suficiente determinar que un número no es primo, qué el caso de afirmar que sí lo es. Analizando los valores obtenidos en los números primos a diferencia de los compuestos ha llegado a la magnitud de 5.000 en lugar de los 22.000 obtenidos en los compuestos. En cuanto al tiempo obtenido de promedio rondan los 5.000 ms, tiempos mucho mayores en comparación a los vistos en los números compuestos.



Como se puede observar en esta gráfica comparativa, los valores primos disparan el coste en tiempo en contraste con los compuestos, los cuales mantienen unos valores muy bajos. Este hecho tiene sentido debido a que los valores compuestos deben pasar por un menor número de filtros durante la ejecución del algoritmo con respecto a los primos, hecho que dispara el coste en tiempo de ejecución.

En cuanto a la tendencia de ambos algoritmos, esta aparenta ser una complejidad lineal en ambos casos, teniendo una tendencia creciente de coste conforme aumenta el valor del número.

Conclusiones y problemas encontrados

Tras la realización de este primer hito hemos podido comprobar en su mayoría el funcionamiento del algoritmo AKS para la obtención de números primos, así como el principal objetivo de la práctica que trata de analizar la complejidad computacional por partes de un algoritmo complejo para obtener la complejidad computacional final del algoritmo.

En cuanto a los problemas y dificultades que hemos encontrado durante la realización de este hito, nos encontramos al principio la dificultad de analizar el código fuente con las implementaciones del algoritmo AKS, para posteriormente realizar sus correspondientes análisis empírico y analítico. Dentro del análisis analítico nos hemos encontrado varias dificultades relacionadas con la evaluación de los costes dentro del código, así como a la hora de computar la complejidad total de cada una de las partes. En el apartado de la complejidad empírica el mayor reto ha sido obtener una tendencia a partir de los valores que hemos obtenido tras realizar las ejecuciones correspondientes, dado que dependiendo de los casos que se den dentro de la ejecución del algoritmo se van a poder apreciar algunas tendencias o por el contrario no va a ser posible sacar nada en claro.

Con este segundo hito nos hemos encontrado el estudio del cálculo de Totient que a priori tras realizar el primer hito la dificultad de este no ha sido tanta en comparación al anterior. Como punto a destacar se encuentra en aprender cómo funciona el algoritmo de cálculo de Totient, el cual nos ha resultado bastante interesante.

En el tercer hito nos dimos cuenta del funcionamiento total del algoritmo. Hemos podido notar que la posición de los algoritmos es importante. El algoritmo lo que realiza es primero realizar comprobaciones más sencillas donde si el resultado sale un número primo, devuelve el resultado. En caso contrario, es comprobado de otra forma hasta que llega a la condición suficiente. Es importante la organización entre comprobaciones porque la comprobación si es primo o no se hace más costosa cuanto más avanza el algoritmo. De forma que la heurística 1 puede comprobar de forma muy rápida si es primo o no, pero puede dar como no primo algunos que sí lo son. Así hasta llegar a la condición suficiente que comprueba de forma fehaciente si es primo o no, pero es el que más tarda en dar una conclusión. En el caso de que se analice un número compuesto, no hay ningún problema porque todos los algoritmos verifican que no es primo en muy poco tiempo.

Por lo general comparando el estudio analítico realizado se puede apreciar que en su mayoría coinciden los resultados expuestos del analítico con las gráficas obtenidas de las ejecuciones del empírico. Con ello se puede concluir que se ha realizado aparentemente de la forma correcta el estudio analítico de los algoritmos que componen AKS.

Referencias

[1] M. Agrawal, N. Kayal y N. Saxena, “PRIMES is in P”, Annals of Mathematics, vol. 160, pp. 781-793, 2004.