TEORÍA AVANZADA DE LA COMPUTACIÓN

Grado en Ingeniería Informática y Administración de Empresas

MEMORIA

Práctica 1 - AKS - Entrega final

Jaime Martínez de Miguel. 100386281
Ricardo Prieto Álvarez. 100386267
Miriam Valdizán Arce. 100386356

ÍNDICE

Introducción:

En este documento, realizaremos el análisis empírico y el estudio analítico de la implementación del algoritmo AKS, así como una conclusión comparando los resultados anteriores.

Estudio empírico:

Para cerciorarnos de que los resultados obtenidos tras el estudio analítico coincidan con las gráficas conseguidas del estudio empírico hemos utilizado la herramienta *GeoGebra.*

En este estudio empírico solamente incluiremos las pruebas realizadas con números primos, debido a la falta de claridad en los resultados obtenidos con números compuestos. Esto se debe a que tienen tiempos que difieren independientemente del tamaño del número (no siguen una tendencia) y por ello arrojan gráficas muy confusas e imposibles de analizar.

En las 3 primeras heurísticas y en el cálculo de *Totient* ejecutamos pruebas para números hasta 50 bits utilizando el método *probablePrime()*, con 100 iteraciones por número (es decir, 100 números creados para cada número de bits). En cambio, en el último estudio (el análisis de la condición suficiente) solamente lo hicimos hasta 19 bits con 50 iteraciones por bit, debido al gran tiempo que tardaba en ejecutarse cada número a partir de los 17 bits (en total, tardamos 12 horas en ejecutar estas pruebas).

En un principio, el análisis empírico arrojaba gráficas con huecos muy grandes entre medias (como se puede apreciar en la Fig 1.). Finalmente se intuyó que esto podría deberse al crecimiento del contador de bits del bucle proporcionado en el programa *AksTiming*, el cual se incrementaba de 2 en 2 bits. Por ello, cambiamos este crecimiento y pasó a aumentar de 1 en 1 bit, ofreciendo gráficas mucho más precisas y claras (como se puede apreciar en la Fig 2. , ambas midiendo el tiempo de la potencia perfecta).

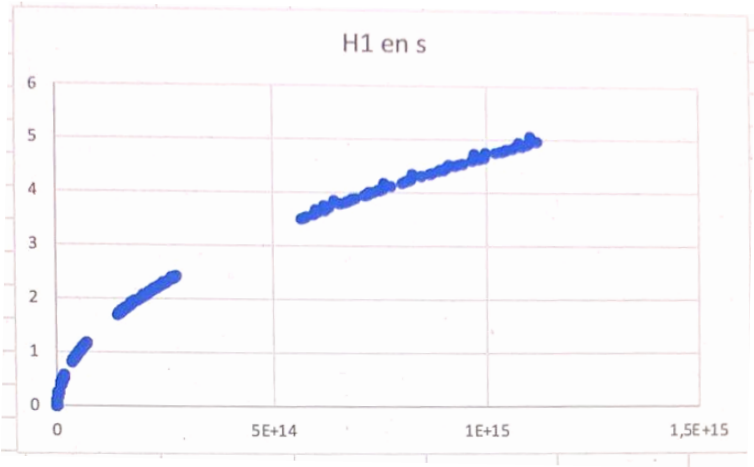

Fig 1. (comprobación de si el número es una potencia perfecta) con números primos

A continuación, insertamos la Fig 2. , que representa el cálculo de la heurística 1 (comprobación de si el número es una potencia perfecta) con números primos:

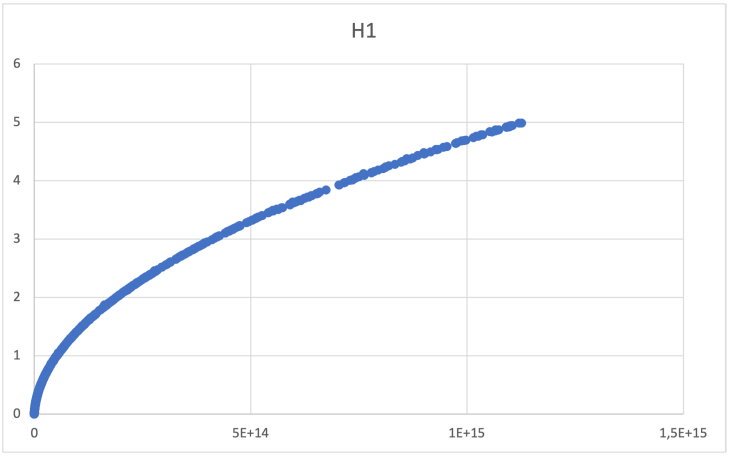

Fig 2. Heurística 1 (comprobación de si el número es una potencia perfecta) con números primos

Como podemos observar, se aproxima de una manera muy precisa a la función $\sqrt{n} \times log(n)$, que es la función que resulta del estudio analítico.

A continuación, insertamos la Fig 3. ,que representa el cálculo de la heurística 2 (cálculo de r) con números primos:

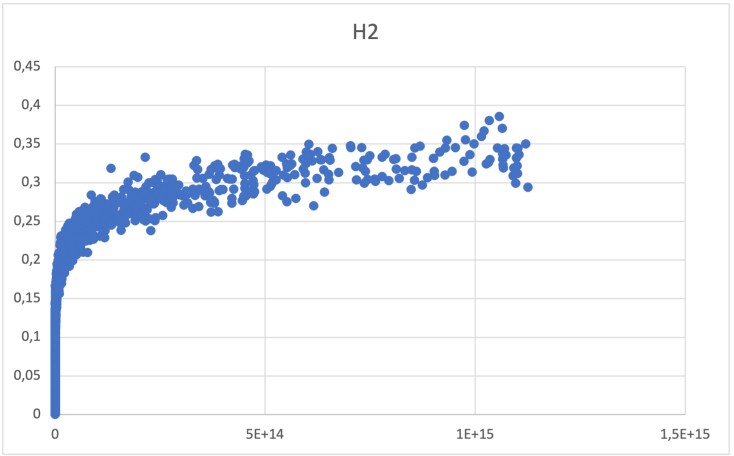

Fig 3. Heurística 2 (cálculo de r) con números primos

Como podemos observar, se aproxima a una función logarítmica con un crecimiento algo mayor que el de la función $log(n)$, debido a que la complejidad, demostrada en el estudio analítico, es $\log(n)^5$, con lo que concuerdan ambos estudios.

A continuación, insertamos la gráfica Fig 4. que representa la comprobación de que mcd(n,r) distinto de 1 con números primos:

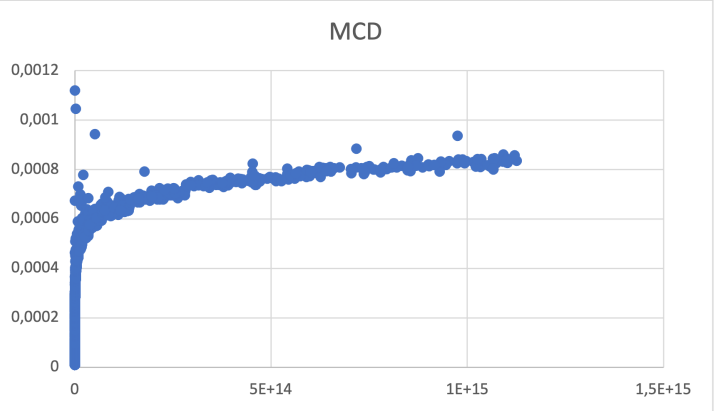

Fig 4. Comprobación de que el mcd(n, r) es distinto de 1 con números primos

Como podemos observar, se aproxima a una función logarítmica, que es la función que resulta del estudio analítico.

A continuación, insertamos la gráfica Fig 5. que representa el cálculo de *Totient* con números primos:

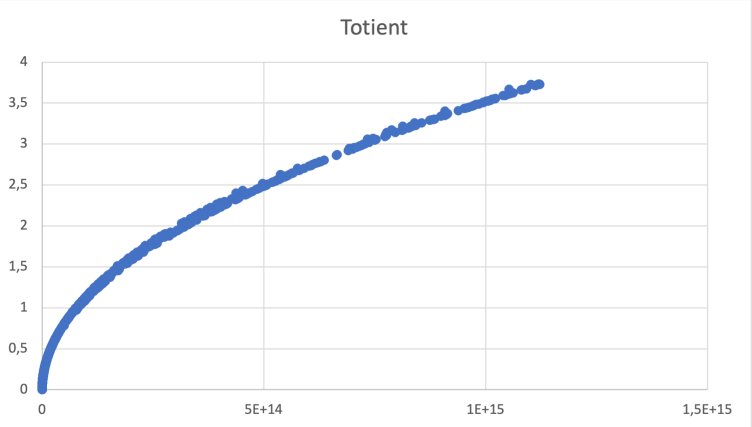

Fig 5. Cálculo de totient con números primos

Como podemos observar, se aproxima bastante a la función $\sqrt{n}$ xlog (n), que es la función resultante tras el estudio de la complejidad analítica.

A continuación, insertamos la gráfica Fig 5. que representa el análisis de la condición suficiente con números primos:

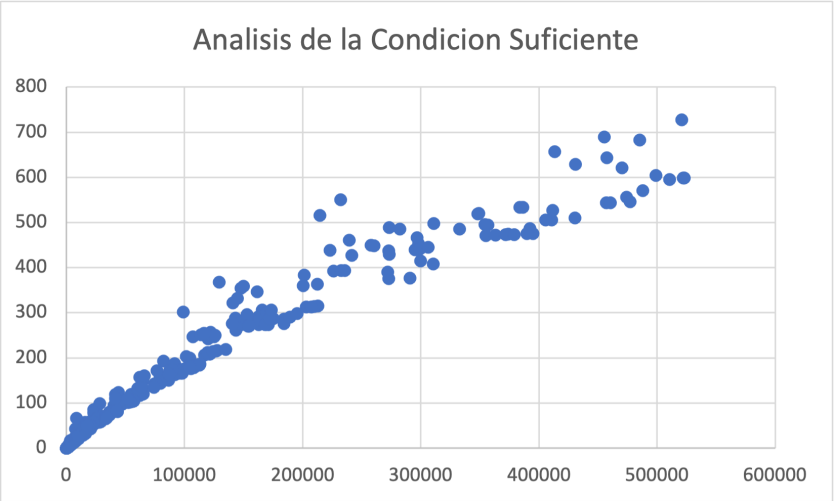

Fig 6. Condición suficiente con números primos

Como podemos observar, se aproxima ligeramente a la función $\sqrt{\sqrt{n} \times log(n)} \times log(n)$, que es la función resultante tras realizar parte del estudio de la complejidad analítica, donde faltaría analizar todos los métodos de la clase *Poly*.

Estudio analítico:

- Hito 1:

  - Heurística 1 - Cálculo Potencia Perfecta

El do-while externo, al compararse $base^{2} \leq n$, se ejecutará un máximo de $\sqrt{n}$ veces. En la Fig 7. se demuestra la complejidad del bucle do-while interno así como la complejidad total del cálculo de la potencia perfecta.

`BigInteger result;`	
`int power = Math.max((int) (log() / log(base) - 2), 1);`	O(1)
`int comparison;`	O(1)
`do`	
`    power++;`	O(1)

`result = base.pow(power);`	O(1)
`comparison = n.compareTo(result);`	O(1)
`while (comparison > 0 && power < Integer.MAX_VALUE);`	O(log(n))
`if (comparison == 0)`	O(1)
`factor = base;`	O(1)
`n_isprime = false;`	O(1)
`return n_isprime;`	O(1)
`base = base.add(BigInteger.ONE);`	O(1)
`aSquared = base.pow(2);`	O(1)
`}`	
`while (aSquared.compareTo(this.n) <= 0);`	O(√n)
O($\sqrt{n}$) x O(log (n))	

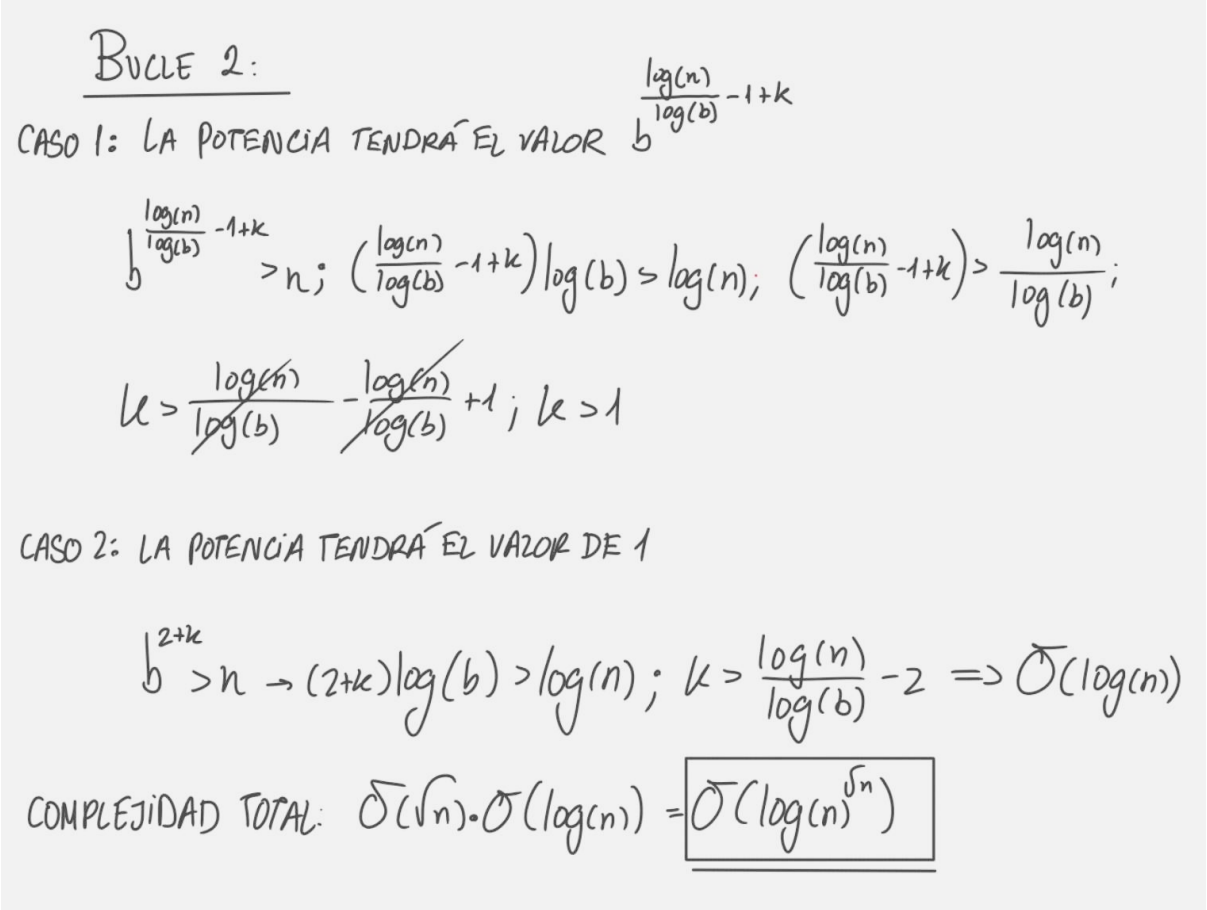

Fig 7. Demostración complejidad bucle do-while interno Heurística 1.

- Heurística 2 - Cálculo de r

```
Do
    k = multiplicativeOrder(r)
while( k.doubleValue() < logSquared)
```

Lemma 4.3. *There exist an* $r \leq \max\{3, \lceil \log^5 n \rceil\}$ *such that* $o_r(n) > \log^2 n$.

El cálculo de r se ejecuta a través de un bucle do-while, en el cual la condición depende de k (el orden multiplicativo de *n módulo r*, $O_r(n)$) y logSquared (representa un simple $log^2 n$). Como hemos podido comprobar en la literatura [Lemma 4.3], el máximo r que cumple esta condición es $log^5(n)$.

Por tanto, la complejidad del cálculo de r es:

$$O(log^5(n))$$

Dentro del método *multiplicativeOrder*, hemos podido demostrar empíricamente (a través de la realización de pequeños estudios) que el máximo número de iteraciones del bucle do-while es k = r - 1.

- Máximo Común Divisor

En la literatura proporcionada hemos podido consultar que la peor instancia en el cálculo del máximo común divisor se corresponde con el caso en que dos números sean elementos sucesivos de la sucesión de Fibonacci. Con esto sabemos que el número de iteraciones del bucle será equivalente al subíndice del término de la sucesión (f_3 serían 3 iteraciones, por ejemplo).

A través de la fórmula de Lucas, en la que desarrolla cómo obtener este término,

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

y teniendo en cuenta que f_n es asintótico a $\frac{\phi^n}{\sqrt{5}}$ se puede determinar, tomando logaritmos en base ϕ (el número áureo), que la complejidad del cálculo del mcd es $O(log\ f_n)$.

$$n \simeq log_\phi(f_n\sqrt{5}) \simeq log_\phi(f_n)$$

- Relación entre n y r:

En este aspecto, hemos de mencionar que la relación que han mantenido n y r ha sido siempre la misma para todos nuestros cálculos y pruebas. En concreto esa relación ha sido $n \leq r$.

❖ Hito 2

- Cálculo de Totient

Debido a que la condición de salida del bucle for es $n > i^2$, el bucle se ejecutará $\sqrt{n}$ veces. Además el while interno, al ir subdividiendo el problema (dividiendo por i, básicamente) sucesivamente, tiene una complejidad logarítmica.

`BigInteger totient(BigInteger n)`	
`BigInteger result = n;`	O(1)

`for (BigInteger i = BigInteger.valueOf(2); n.compareTo(i.multiply(i)) > 0; i = i.add(BigInteger.ONE)) {`	$O(\sqrt{n})$
`if (n.mod(i).compareTo(BigInteger.ZERO) == 0)`	O(1)
`result = result.subtract(result.divide(i));`	O(1)
`while (n.mod(i).compareTo(BigInteger.ZERO) == 0)`	$O(log_i(n))$
`n = n.divide(i);`	O(1)
`if (n.compareTo(BigInteger.ONE) > 0)`	O(1)
`result = result.subtract(result.divide(n));`	O(1)
`return result;`	O(1)
$O(\sqrt{n})$ x $O(log$ (n))	

❖ Hito 3

Debido a la alta complejidad matemática que hay detrás del cálculo de:

$$Si\ (\ (X+a)^{num} \neq X^{num} + a\ (mod\ X^r - 1,\ num)\ ),$$ entonces COMPUESTO;

nos impide analizar con total exactitud y precisión la complejidad analítica de esta parte del programa (en la clase denominada Poly). Lo que sí podemos analizar es la cantidad de iteraciones que tendrá el bucle que realiza este cálculo anterior.

El bucle for tiene como condición de salida $i \leq limit$, donde limit es igual a $\sqrt{\phi(n)} \times log(n)$ y por ello parte de la complejidad analítica será:

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

Conclusiones:

En esta práctica valoramos bastante el hecho de haber hecho un seguimiento a través de los hitos, revisando y rectificando los errores cometidos cuando ha sido necesario. En concreto hemos tenido más dificultades a la hora de realizar las pruebas empíricas debido a que no sabíamos muy bien cuántas pruebas eran necesarias para observar con claridad el comportamiento de los distintos fragmentos del programa.

En relación al estudio analítico hemos obtenido una gran ayuda a través del material de soporte que se nos proporcionó en Aula Global, tal como fragmentos de libros y apuntes relacionados con el algoritmo AKS así como vídeos explicativos de estos apuntes. Además de esto, hemos encontrado información de gran utilidad en internet que nos ha servido para demostrar cálculos relativos a la complejidad de algunas partes.

En general, consideramos que ha sido una práctica donde hemos podido aplicar en gran parte los conocimientos adquiridos en las clases de teoría.

Referencias:

Para buscar la tendencia de las gráficas: Calculadora gráfica - GeoGebra:
https://www.geogebra.org/graphing?lang=es

Maximo r posible en el *multiplicativeOrder* (lemma 4.3):
[1] M. Agrawal, N. Kayal y N. Saxena, “PRIMES is in P”, Annals of Mathematics, vol. 160, pp. 781-793, 2004.

Ayuda para demostrar la complejidad del máximo común divisor:
https://math.stackexchange.com/questions/2477328/why-are-fibonacci-numbers-bad-for-euclids-algorithm-and-how-to-derive-this-uppe