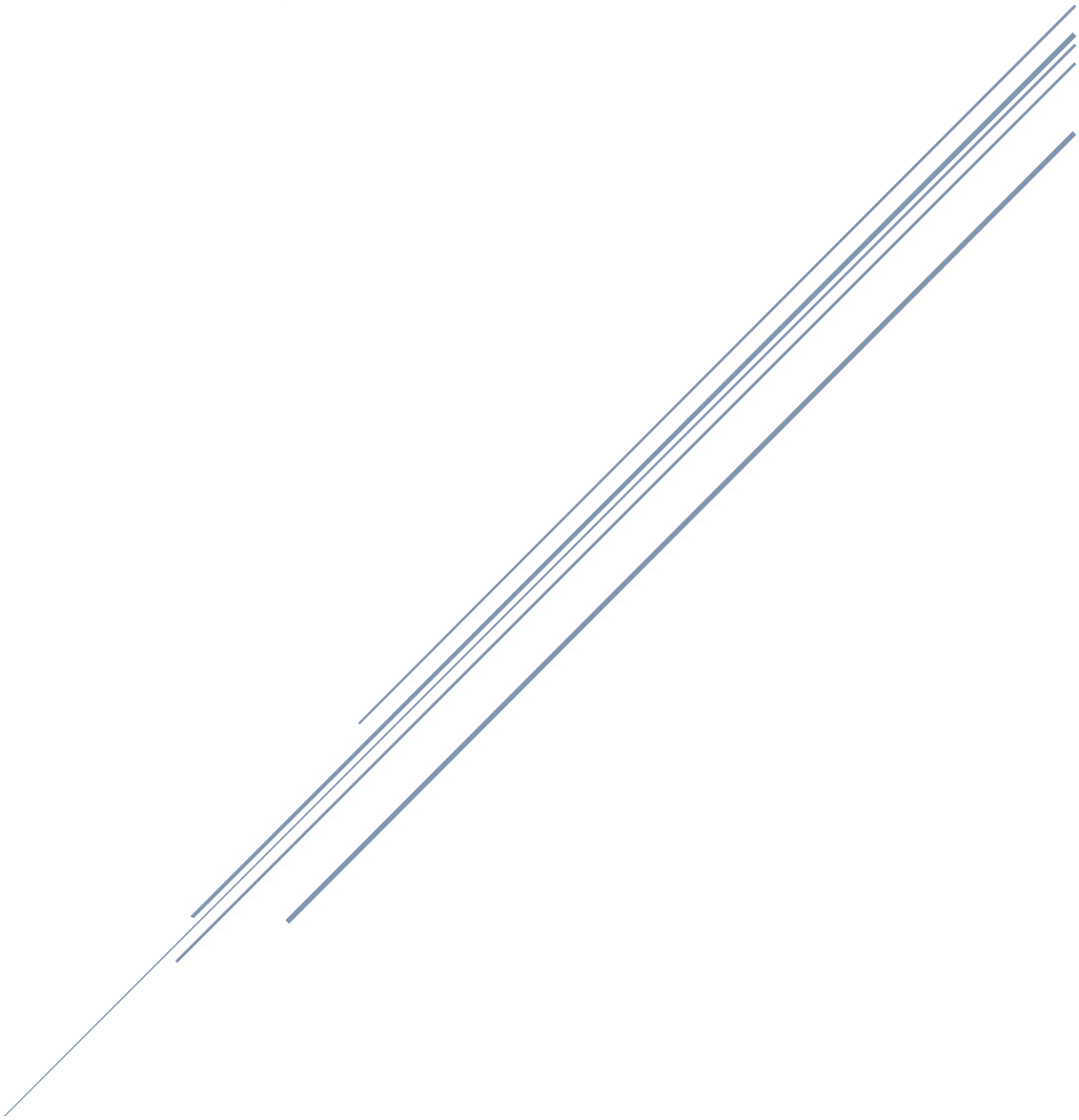
TEORÍA AVANZADA DE LA COMPUTACIÓN

PRÁCTICA 1, 4º curso

Universidad Carlos III Madrid, Ingeniería Informática, 2021



grupo 80 - Colmenarejo - IVAN AGUADO PERULERO – 100405871
grupo 80 - Colmenarejo – JAVIER CRUZ DEL VALLE – 100383156
grupo 80 - Colmenarejo - JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN

El objetivo de esta práctica es realizar un análisis de la complejidad del algoritmo AKS. Este es un algoritmo determinista que determina, en tiempo polinómico, si un número natural es primo o compuesto. Se basa en el Teorema Pequeño de Fermat (TPF) utilizando el cálculo de potencias perfectas, del número *r* tal que $Or(n) > 4\ log2\ n$ y del máximo común divisor (*mcd*).

En cuanto al presente documento, en primer lugar, encontramos una introducción al algoritmo y a la práctica. A continuación, un estudio empírico en el que se determina experimentalmente la complejidad temporal *T(n)* realizando distintas pruebas de ejecución y midiendo los tiempos correspondientes para obtener *O(n)*. Lo siguiente es un estudio analítico en el que se examinan los distintos pasos del algoritmo y se deducen los costes computacionales. Por último, se detallan unas conclusiones extraídas durante la realización de la práctica y el estudio realizado, así como los problemas encontrados.

2.- ESTUDIO EMPÍRICO

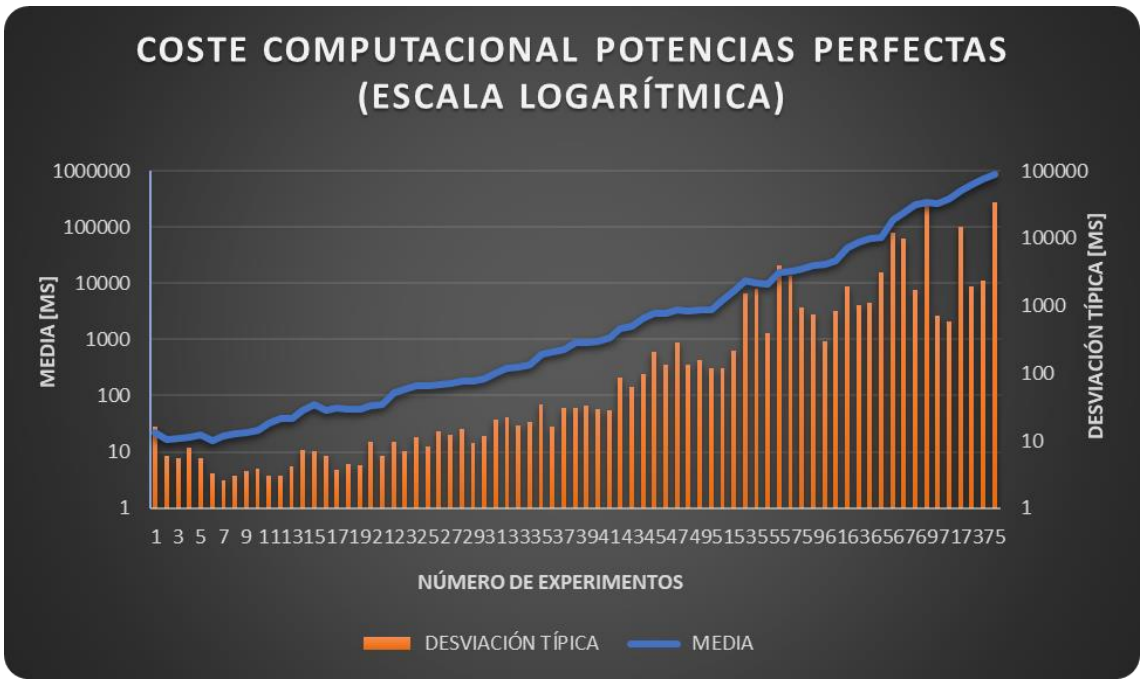
En esta primera sección el objetivo es determinar experimentalmente el coste computacional *O(n)*, considerándose el tamaño de entrada como el número de dígitos. Se han realizado un total de 75 pruebas distintas que comprenden números primos de orden 10 hasta orden 19, con 10 representantes por orden desde el orden 10 al 14, y 5 representantes del orden 15 al 19. Estos últimos son menos, pues los equipos con los que contamos no tienen suficiente potencia para ejecutar el algoritmo en un tiempo razonable.

Cada número primo se ha ejecutado 10 veces para evitar sesgos y poder mostrar su valor medio, así como la desviación típica. La elección de este tipo de números es debido a que muestran el peor caso posible para los algoritmos estudiados.

Para su obtención hemos usado una web que, dado un número cualquiera, te decía su número primo posterior o anterior. Esta se encuentra referenciada en la bibliografía.

Cabe destacar que para la realización del estudio hemos extraído los algoritmos del código principal para, de esta manera, poder testearlos de manera individual. Con el fin de automatizarlo y no tener que realizar numerosas ejecuciones manualmente, hemos generado un array ordenado con los números que íbamos a probar, de tal manera que el propio código los fuera cogiendo de ahí y ejecutando iterativamente.

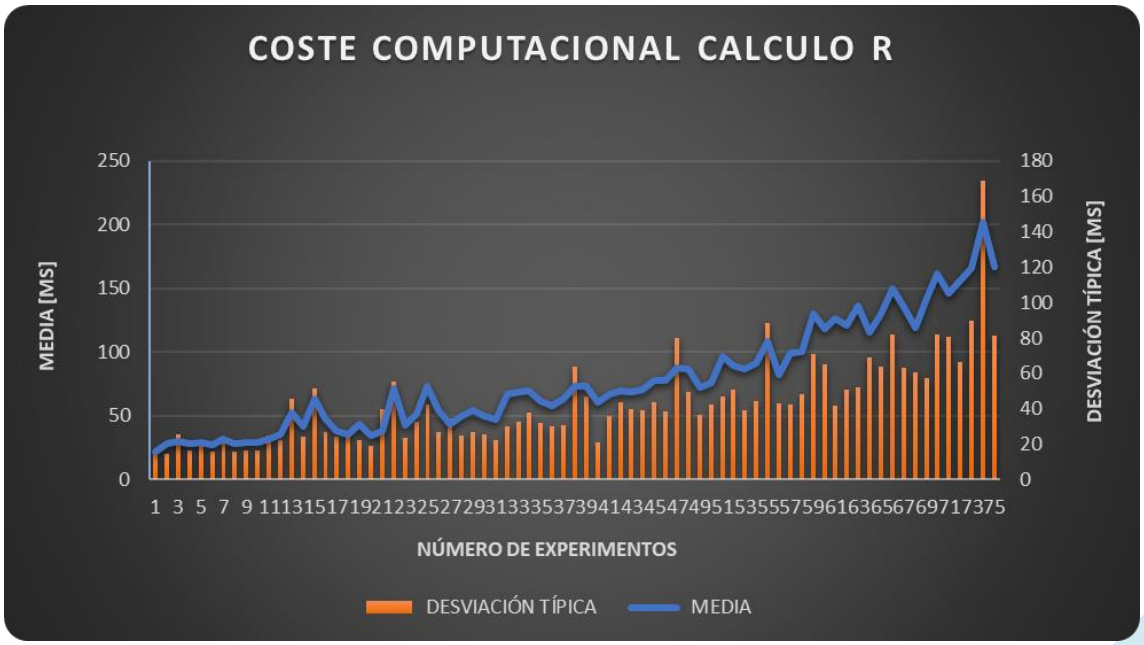
2.1.- Verificación de potencia perfecta



Como podemos ver, la gráfica se encuentra en escala logarítmica debido a que la visualización de la misma sin esta modificación carece de sentido, perdiendo precisión de cada uno de los valores tomados. El comportamiento es claramente exponencial para ambas variables, media y desviación típica, razón del uso de la escala.

Para el primer valor, correspondiente al primo 1731121291, se obtiene un tiempo total de ejecución de 221 milisegundos; mientras que, para el último valor, correspondiente al primo 7305843009213694067, el tiempo total de ejecución ha sido de 2 horas y 21 minutos.

2.2.- Cálculo de r

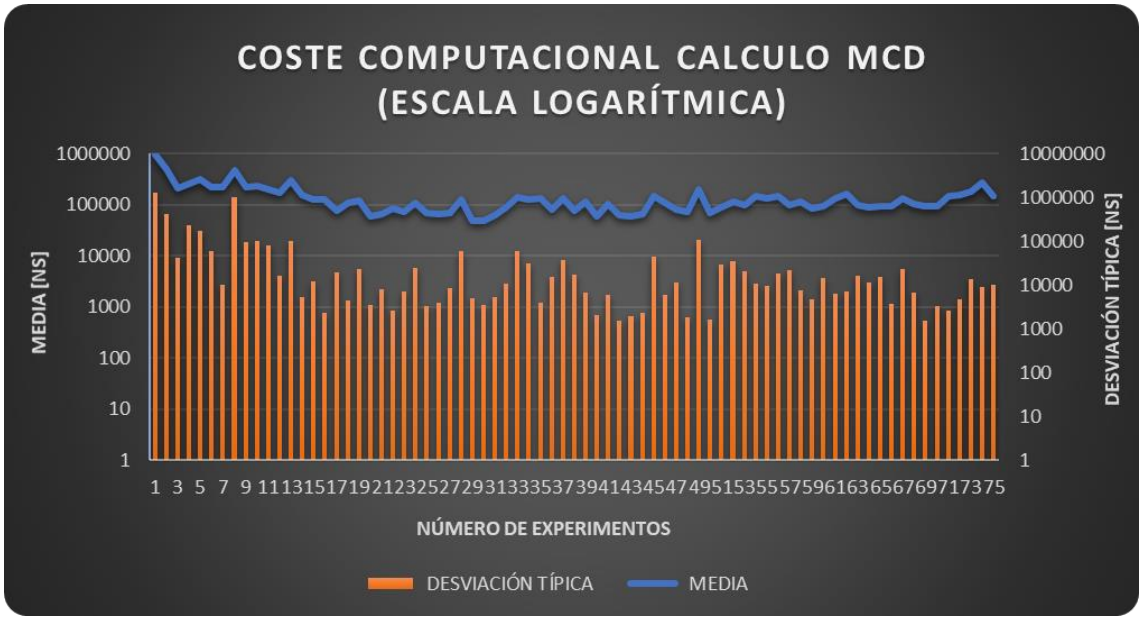


Para esta gráfica ya no ha sido necesaria la escala logarítmica, ya que los valores obtenidos no son tan grandes como en el caso anterior. A su vez, en este caso la automatización del código no ha sido posible pues los resultados obtenidos no eran correctos, esto se comentó con nuestra profesora y procedimos a hacerlo manualmente.

Como se puede observar, para los números primos elegidos, el tiempo de ejecución crece exponencialmente a medida que se aumenta el número de cifras del número.

Para el primer valor, correspondiente al primo 1731121291, se obtiene un tiempo total de ejecución de 22 milisegundos; mientras que, para el último valor, correspondiente al primo 7305843009213694067, el tiempo total de ejecución ha sido de 167 ms. Por lo que se puede ver, aunque aumente para los números elegidos, no se obtiene una gran diferencia en cuanto a tiempos entre los números más grandes y los más pequeños.

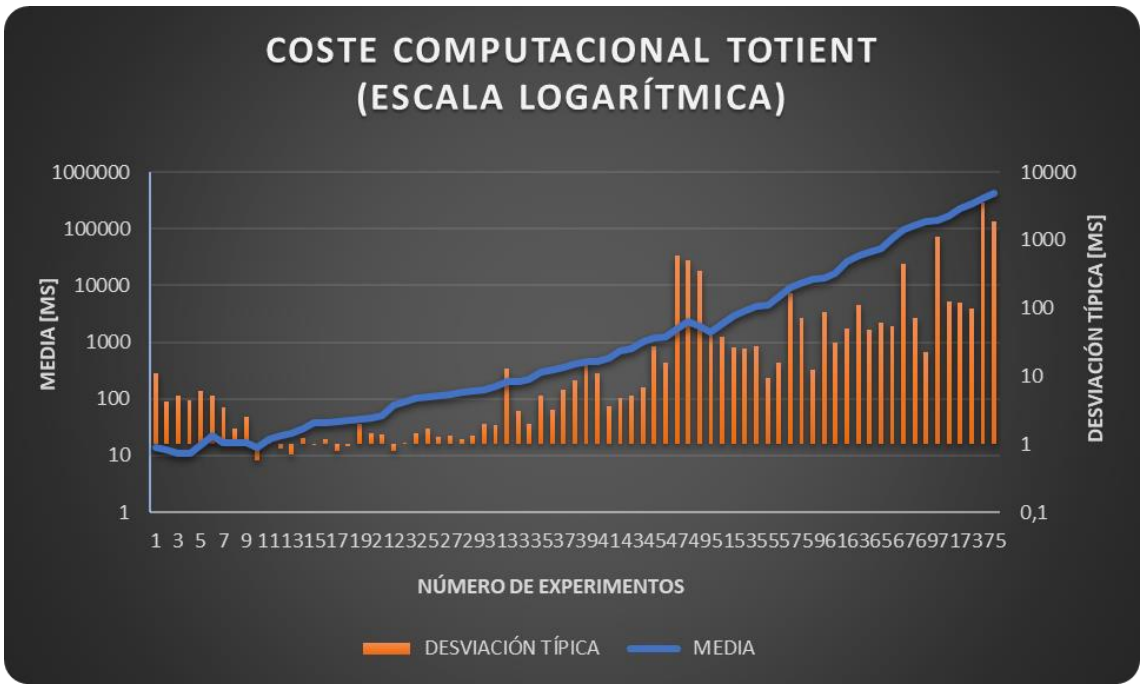
2.3.- Cálculo del mcd



Por último, en la siguiente gráfica vemos los resultados obtenidos para realizar el cálculo del máximo común divisor (*mcd*) entre dos números. En este caso, de nuevo no es necesaria la escala logarítmica, ya que los valores obtenidos son muy pequeños. De hecho, en este caso hemos tenido que modificar el código para obtener los tiempos en nanosegundos en vez de milisegundos, porque si no la gráfica no aportaba ninguna conclusión.

La media más alta se encuentra en 0,97 ms, y la desviación típica en aproximadamente 1,27 ms. Esto se debe a que el algoritmo de Euclides empleado para calcular el *mcd* es extremadamente óptimo y por eso tiene un coste computacional más bajo y constante que el resto. Por otro lado, también observamos que la desviación típica es demasiado alta (mayor que 1), lo que quizá implique la presencia de valores extremos. Esto es debido a la primera medición en el bucle, y por eso decidimos realizar 10 experimentos para cada número, como se explicará más adelante.

2.4.- Cálculo del *totient* y determinación de primalidad

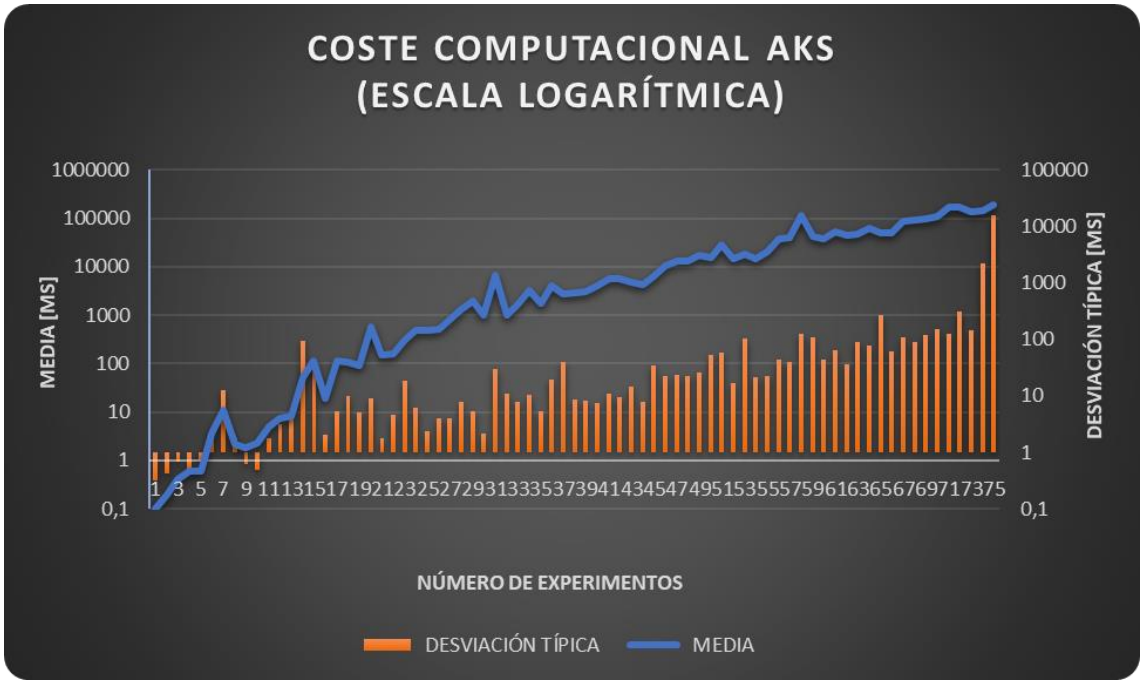


En este caso volvemos a utilizar una escala logarítmica, ya que los datos crecen exponencialmente según van aumentando el número de cifras del número primo evaluado, y de esta manera podremos visualizarlos mejor. En este caso el análisis se ha realizado de la función que calcula el *totient* y determinación de primalidad, sin tener en cuenta el paso 5 completo del algoritmo AKS, ya que así se nos ha indicado en clase.

Para el primer valor, correspondiente al primo 1731121291, se obtiene un tiempo total de ejecución de 137 milisegundos; mientras que, para el último valor, correspondiente al primo 7305843009213694067, el tiempo total de ejecución ha sido de 1 hora y 17 minutos.

2.5.- Algoritmo AKS completo

Para esta sección en concreto se han realizado un total de 75 pruebas distintas que comprenden números primos de orden 1 hasta orden 9, con 5 representantes por orden desde el orden 1 al 4 y con 11 representantes del orden 5 al 9. Nuevamente, cada número primo se ha ejecutado 10 veces para evitar sesgos y poder mostrar su valor medio, así como la desviación típica. Hemos cambiado los números por unos más pequeños que en el caso anterior porque si no los tiempos serían excesivamente altos y nuestros dispositivos no rendían lo suficiente.



Observamos que la gráfica se encuentra en escala logarítmica debido a que la visualización de la misma sin dicha modificación carecería de sentido, al igual que nos ocurría anteriormente. También podemos decir que el comportamiento es claramente logarítmico para ambas variables, media y desviación típica, aunque se perciben ciertas anomalías tal vez causadas por el propio dispositivo de medición. Estas anomalías se suelen producir cuando el número primo introducido pasa a tener una cifra más, por lo que podría tener sentido que el coste computacional fuera mayor.

Para el primer valor, correspondiente al primo 1, se obtiene un tiempo total de ejecución de 1 milisegundo; mientras que, para el último valor, correspondiente al primo 999999929, el tiempo total de ejecución ha sido de 32 minutos.

Cabe destacar que en el resto de gráficas si se utilizara un rango más amplio de números de prueba se obtendrían curvas más diferenciadas que se asemejan más a las complejidades obtenidas durante el estudio analítico. Esto ha sido comprobado y es así, pero optamos por obtener una precisión más alta en nuestras mediciones con la ejecución de cada experimento 10 veces, pues pudimos ver que las primeras ejecuciones siempre tardaban más hasta que el ordenador “se acostumbraba”. Si solo trabajáramos con este primer tiempo las gráficas estarían muy sesgadas. De esta manera en el estudio empírico se puede ver la curva creciente del coste de los algoritmos, la cual definiremos y entenderemos mejor con el estudio analítico que viene a continuación.

3.- ESTUDIO ANALÍTICO

A continuación, analizaremos la complejidad temporal de manera analítica resaltando en cada una de las líneas de código el número de operaciones elementales que se ejecutarán. Cada una de las estructuras iterativas, así como condicionales, tendrán un cálculo intermedio y finalmente se expondrá el coste total de dicho paso.

Antes de comenzar, queremos destacar una serie de aspectos fundamentales que hemos tenido en cuenta gracias a la ayuda del docente. Las llamadas a funciones propias de AKS, así como las llamadas a funciones propias de *Java*, tendrán un coste base de 1 y toda instanciación o devolución de variables tendrán un coste 0. Aunque también se realizará el estudio posterior de las mismas pues forman parte del algoritmo.

Para el análisis de los bloques de código hemos utilizados tres fórmulas vistas en clase:

- La primera de ella, relacionada con estructuras condicionales, en donde se determina que el coste máximo de dicho bloque será el coste de la condición más el máximo de cada una de los sub-bloques:

$$T(n) = T(C) + max\ \{T(IF), T(ELSE)\}$$

- En segundo lugar, utilizaremos las fórmulas para las estructuras de bucle *while*, que determinan que el coste total será la suma del coste de la condición con la suma de dicha condición con el bloque interior multiplicado tantas veces como iteraciones haya. Dicha fórmula la aplicaremos en un *do_while* de tal forma:

$$T(n) = T(DO) + T(WHILE) \rightarrow T(DO) = T(C) + T(S)\ ;\ T(WHILE) = T(C) + n * (T(C) + T(S))$$

$$T(n) = 2T(C) + T(S) + n * (T(S) + T(C))$$

- En último lugar, aplicaremos la fórmula que nos permite cambiar una estructura en bucle de tipo *for* por su equivalente en while de la siguiente manera:

$$For\ (I, C, S')\ \{S\} == I; while(C)\ \{S + S'\} \rightarrow T(n) = T(C) + n * (T(S) + T(S') + T(C))$$

La nomenclatura utilizada en las capturas de pantalla ha sido la siguiente:

- Complejidad temporal (*OE*, operaciones elementales) de cada línea de código.
- Suma de complejidades temporales debajo de cada estructura de código (secuencial, condicional, iterativa…) Entre corchetes se especifica con una L de qué línea a qué línea pertenece la complejidad mostrada.
- Al final del todo se suman las complejidades, se obtiene la total y su complejidad computacional asociada.

3.1.- Verificación de potencia perfecta

```
public static boolean isPP() {
    do {
        BigInteger result; // 0 OE
        int power = Math.max((int) (log() / log(base) - 2), 1); // 7 OE
        int comparison; // 0 OE
        do {
            power++; // 2 OE
            result = base.pow(power); // 2 OE
            comparison = n.compareTo(result); // 2 OE
        } while (comparison > 0 && power < Integer.MAX_VALUE); // 3 OE
        // 9 + 9n OE [L6-L10]
        if (comparison == 0) { // 1 OE
            if (verbose) // 1 OE
                System.out.println(n + " is a perfect power of " + base); // 1 OE
            // 2 OE [L13-L14]
            factor = base; // 1OE
            n_isprime = false; // 1OE
            return n_isprime; // 0OE
        }
        // 5 OE [L12-L19]
        if (verbose) // 1OE
            System.out.println(n + " is not a perfect power of " + base); // 1OE
        // 2 OE [L21-L22]
        base = base.add(BigInteger.ONE); // 2OE
        aSquared = base.pow(2); // 2OE
    } while (aSquared.compareTo(this.n) <= 0); // 3OE
    // 9n^2 + 39n + 30 OE [L2-L26]
    if (verbose) // 1 OE
        System.out.println(n + " is not a perfect power of any integer less than its square root"); // 1 OE
    // 2 OE [L28-L29]
}
// SOLUCIÓN: 9n^2 + 39n + 32 OE --> T(n) = O2
```

Comenzamos con el algoritmo del cálculo de la potencia perfecta. Observamos que se trata de un bucle anidado a otro, los cuales, como mínimo, serán ejecutados una vez debido a su estructura. Internamente cuentan con estructuras condicionales de orden 1. Es decir, podemos concluir que el orden total de dicho algoritmo es cuadrático, en base a la estructura externa que presenta y sin analizar cada una de las funciones propias de Java o llamadas a otras funciones de la clase desarrollada.

Para contrastar el análisis de la potencia perfecta verificaremos la condición de parada tanto del bucle interno y externo.

El bucle interno se realizará hasta que la base "*b*" elevada a la potencia "*n*" dada sea mayor que el número introducido. Por tanto, se deberá tener en cuenta el valor inicial del exponente y el valor máximo que puede llegar a ser. Este máximo es el techo de la función logaritmo en base "*b*" de "*n*" y el valor en el que empieza el exponente será el cociente de los logaritmos en base 2 de "*n*" y "*b*" menos 2.

Sabemos que el bucle externo se ejecutará un máximo de raíz de "*n*" veces ya que como confirman las cuestiones básicas aportadas nos mantendremos en el bucle siempre que la base al cuadrado sea menor que "*n*".

Por tanto, obtenemos una complejidad logarítmica multiplicada por una complejidad racional por lo que despejando el resultado obtenemos una complejidad total del logaritmo de "*n*" elevado a su raíz cuadrada.

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

3.2.- Cálculo de r

```
public static BigInteger calculateR() {
    double log = log(); // 2 OE
    double logSquared = log * log; // 2 OE
    BigInteger k = BigInteger.ONE; // 2 OE
    BigInteger r = BigInteger.ONE; // 2 OE
    do {
        r = r.add(BigInteger.ONE); // 3 OE
        if (verbose) // 1 OE
            System.out.println("trying r = " + r); // 1 OE
        // 2 OE [L8-L9]
        k = multiplicativeOrder(r); // 2 OE
    } while (k.doubleValue() < logSquared); // 2 OE
    // 9n + 9 OE [L6-L12]
    if (verbose) // 1 OE
        System.out.println("r is " + r); // 1 OE
    // 2 OE [L14-L15]
    return r; // 0 OE
}
// SOLUCIÓN: 9n + 19 --> T(n) = O1
```

Seguimos con el algoritmo del cálculo de *r*, el cual se trata de un bucle que cuenta con estructuras condicionales dentro y fuera del orden 1. Es decir, se puede afirmar que el orden total de dicho algoritmo es lineal.

Esto es si no analizamos las llamadas a las funciones *multiplicativeOrder* y *logSquared*, si lo hacemos:

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

Según el lema 4.3 se puede afirmar que en el peor de los casos $r = log(n)^5$

Debido a esto se puede ver que $T(n) = log(n)^2$ multiplicado por *r* en su peor caso $T(n) = log(n)^7$.

Esto cambiaría la complejidad del cálculo de *r* a $O(log\ (n))$.

MultiplicativeOrder no cambiaría nada en cuanto a complejidad pues, es el menor *k* tal que $\frac{(n^k - 1)}{r}$ sea exacto, este obligatoriamente debe ser menor que *logSquared* para seguir el bucle.

3.3.- Cálculo del mcd

```
public static void calculateMCD() {
    for (BigInteger i = BigInteger.valueOf(2); i.compareTo(r) <= 0; i = i.add(BigInteger.ONE)) { // 6 OE
        BigInteger gcd = n.gcd(i); // 2 OE
        if (verbose) // 1 OE
            System.out.println("gcd(" + n + "," + i + ") = " + gcd); // 1 OE
        // 2 OE [L4-L5]
        if (gcd.compareTo(BigInteger.ONE) > 0 && gcd.compareTo(n) < 0) { // 5 OE
            factor = i; // 1 OE
            n_isprime = false; // 1 OE
            return false; // 0 OE
        }
        // 7 OE [L7-L11]
    }
}
// SOLUCIÓN: 16n + 4 --> T(n) = O1
```

Finalizamos con el algoritmo del cálculo de *mcd*, el cual se trata de un bucle que cuenta con dos estructuras condicionales dentro del orden 1. Claramente, se tratará de una complejidad de orden lineal.

En la literatura que se nos ha proporcionado, podemos observar que se nos menciona que el peor caso del cálculo del máximo común divisor es el que se produce cuando se utilizan dos números sucesivos de la sucesión de Fibonacci. Por lo tanto, sabemos que el número de iteraciones del bucle será igual al índice del término de la sucesión.

Por otro lado, la fórmula de Lucas nos dice que un término de la sucesión de Fibonacci es:

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

Utilizando el número áureo ($\Phi = \frac{1+\sqrt{5}}{2}$) como base para tomar logaritmos y despejar *n*, podemos transformar la fórmula de Lucas f_n y concluir que la complejidad el logarítmica.

$$O(log\ f_n).$$

2.4.- Cálculo del totient y determinación de primalidad

En primer lugar, analizaremos el código del método *totient* en concreto, dando un valor de uno a las llamadas a funciones externas.

```
public static BigInteger totient(BigInteger n) {
    BigInteger result = n; // 1 OE
    for (BigInteger i = BigInteger.valueOf(2); n.compareTo(i.multiply(i)) > 0; i = i.add(BigInteger.ONE)) { // 8 OE
        if (n.mod(i).compareTo(BigInteger.ZERO) == 0) // 4 OE
            result = result.subtract(result.divide(i)); // 3 OE
        // 7 OE [L4 - L5]
        while (n.mod(i).compareTo(BigInteger.ZERO) == 0) // 4 OE
            n = n.divide(i); // 2 OE
        // 6n + 4  OE [L7 - L8]
    }
    // 6n^2 + 17n + 3 OE [L3 - L10]
    if (n.compareTo(BigInteger.ONE) > 0) // 2 OE
        result = result.subtract(result.divide(n)); // 2 OE
    // 4 OE [L12 - L13]
    return result; // 0 OE
}
//SOLUCIÓN: 6n^2 + 17n + 8 OE --> T(n) = O2
```

Observamos que se trata de una estructura condicional anexada a un *while* dentro de un bucle *for*. Internamente cuenta orden *1+n*, por lo que podemos concluir que el orden total de dicho algoritmo será de *n*(1+n)* sin analizar cada una de las funciones internas llamadas ni otras de la clase previamente desarrollada.

Sin embargo, la condición de salida del bucle *for* es $n > i^2$, por lo que este tramo de código se ejecutará $\sqrt{n}$ veces, es decir tendrá una complejidad O($\sqrt{n}$).

Por otro lado, en el *while* interno se subdivide iterativamente el problema, dotando así a esta parte de una complejidad logarítmica.

Dicho todo esto, si se tuviese en cuenta, la complejidad del cálculo del *totient* pasaría a ser:

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

2.5.- Análisis de la condición suficiente

Todo el algoritmo AKS se basa en la idea del siguiente lema:

Lemma 2.1. Let a ∈ Z, n ∈ N, n ≥ 2, and (a, n) = 1. Then n is prime if and only if $(X + a)^n = X^n + a \ (mod\ n)$

Para 0 < i < n, el coeficiente de x^i en la función anterior es $\binom{n}{i} a^{n-1}$. Por lo tanto, si *n* es primo el resultado es 0, y si es compuesto el coeficiente no es 0, lo que sugiere pasar a la siguiente ecuación:

$$(X + a)^n = X^n + a \ (mod\ X^r - 1, n)$$

Por el lema anterior, todos los primos lo cumplen para todos los valores de *a* y *r*. El problema es que puede haber algunos números compuestos que también lo hagan para algunos de esos valores. Sin embargo, para un valor de *r* correctamente elegido, si la ecuación anterior se ha satisfecho para diversos valores de *a*, entonces *n* será una potencia perfecta. El número de *aes* y de *r* apropiados están designados por un polinomio en de complejidad logarítmica, por lo que obtenemos un tiempo determinista.

De hecho, esto viene respaldado por el teorema:

Theorem 5.1. The asymptotic time complexity of the algorithm is $O \sim (log^{21/2} / n)$, que podemos encontrar en la literatura proporcionada.

4.- CONCLUSIONES

En primer lugar, consideremos que la práctica ha sido una muy buena introducción para poder aplicar los conocimientos teóricos adquiridos. Cabe mencionar el seguimiento semanal de los hitos a través de las revisiones con el docente, lo cual queremos destacar, ya que ha sido de gran ayuda y es algo que valoramos y agradecemos enormemente.

Respecto a los problemas y dificultades que hemos tenido en la realización de esta práctica, comenzamos con el análisis encapsulado de cada parte del código fuente proporcionado, así como la generación de números primos y su correspondiente automatización para poder generar las gráficas. En cuanto al análisis analítico, se nos han complicado las evaluaciones de los costes dentro del código debido a las funciones propias de Java y otras implementadas, así como a la hora de computar la complejidad total de cada una de las partes.

Comparando ambos estudios se puede apreciar que en su mayoría los resultados expuestos toman las mismas tendencias de forma empírica y analítica por lo que se puede concluir con que se ha realizado correctamente el estudio completo de los distintos algoritmos que componen AKS.

5.- REFERENCIAS

[1] «Number Empire Prime Numbers,» 2021. [En línea]. Available:

https://es.numberempire.com/primenumbers.php

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

[3] Algoritmo AKS - Cuestiones Básicas, Aula Global.

[4] Algoritmo AKS, Aula Global.

[5] Cálculo de r y mcd, Aula Global.

[6] AKS Condición Suficiente, Aula Global.

[7] Análisis de algoritmos (I), Aula Global. (Teoría de clase)

[8] Ejercicios prácticos introductorios, Aula Global.