Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 1, 19/10/20

Práctica 1: Determinación Experimental de la Complejidad Temporal de un Algoritmo

Valle Martínez Luis Eduardo, Rivero Ronquillo Omar Imanol

lvalle212@gmail.com, imanol.rivero7@gmail.com

Resumen: En esta práctica fueron desarrollados dos algoritmos, para calcular la complejidad temporal de los mismos utilizando los ejemplos vistos en clase.
Palabras Clave: Complejidad, Algoritmos, Java.

Introducción

En matemáticas, ciencias de la computación y disciplinas relacionadas, un algoritmo es un conjunto de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos. Un algoritmo se compone de tres partes: Entrada, salida y proceso, siendo el proceso la serie de los pasos ejecutados. La importancia de los algoritmos en nuestra vida moderna es invaluable, pues las grandes comodidades de las que gozamos todos los dias tienen involucrados algoritmos simples o complejos.

Por lo general existen una variedad de algoritmos para resolver el mismo problema, por lo que es necesario determinar cual es el mejor entre todos, esto solamente es posible de saber cuando de realiza un análisis obteniendo la complejidad del algoritmo.

La finalidad de esta práctica es la de determinar la complejidad temporal de los algoritmos propuestos, generando gráficas que reflejen los resultados obtenidos.

Conceptos Básicos

Para comprender el trabajo realizado en esta práctica debemos revisar algunos conceptos.

- **Notación Big θ:** Si $f(n) \in \theta(g(n))$, entonces $f(n) \geq 0\ \forall n \in \mathbb{N}$, tales funciones son llamadas asintóticamente no negativas o asintóticamente positivas.Cuando usamos la notación big θ, estamos diciendo que una cota asintóticamente ajustada sobre el tiempo de ejecución. Es .ªsintóticamente"porque importa únicamente para los valores grandes de n.
- **Notación Big O**: Dado una función $g(n), \mathbb{O}(g(n))$ denota el conjunto de funciones como: $\mathbb{O}(g(n) = \{f(n) : \exists n\ C > 0$ y $n_0 > 0$ tal que $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$. Usamos la notación O grande para las cotas superiores asintóticas, ya que acota el crecimiento del tiempo de ejecución por arriba para entradas suficientemente grandes.

- **Notación Big Ω:** Dadas dos funciones $f(n)$ y $g(n)$, $f(n) \in O(g(n))$ si $f(n) \in \Omega(g(n))$. Usamos la notación Ω grande para límites asintóticos inferiores, ya que acota el crecimiento del tiempo de ejecución por abajo para entradas de tamaños suficientemente grandes.

Suma binaria

La implementación del algoritmo utilizado para sumar arreglos de 2 números binarios, se basó en un *medio sumador*, que es un circuito común y sencillo en el ámbito de la electrónica digital.

Se considera:
r: Número potencia de 2
A: Arreglo de tamaño **r**, cada índice contiene 1 bit(*1*,*0*) representando un número binario donde: $0 \leq$ **Binario** $\leq 2^r - 1$
B: Arreglo de tamaño **r**, cada índice contiene 1 bit(*1*,*0*) representando un número binario donde: $0 \leq$ **Binario** $\leq 2^r - 1$
C: Arreglo de tamaño **r+1**, almacena el resultado de la suma binaria en **A** y **B**.

Tomándolo como un *medio sumador* con 2 números de 1 bit, se guarda en un arreglo *C*, el valor de la operación **XOR** de A[i], B[i] y el acarreo. Se guarda el acarreo resultado de esos 2 bits y mediante un *for*, se avanza de forma descendente en los arreglos A y B.

Pseudocódigo:

```
* Notación
    ^ = XOR
    | = OR
    & = AND

    SumaBinaria(A,B,r)
        acarreo = 0
        for i=r-1  to i>=0 do
            C[i+1] = acarreo ^ A[i] ^ B[i];
            acarreo = (acarreo & A[i]) | (acarreo & B[i]) |
            (A[i] & B[i])
        C[0] = (acarreo > 0) ? 1 : 0
        return C
```

Algoritmo de Euclides(MCD)

El algoritmo de Euclides es una técnica para encontrar rápidamente el Máximo Común Divisor(MCD) entre 2 números enteros.

El procedimiento encontrará el MCD cuando el residuo de la división es igual a 0. Los pasos a realizar son los siguientes:

1. Se divide el número mayor entre el menor
2. Si el residuo no es igual a 0 el divisor será el MCD. En caso contrario se sigue el siguiente paso
3. Se divide ahora el divisor entre el residuo. Se siguen estos pasos hasta encontrar un residuo igual a 0

Se considera:
m: Número mayor
n: Número menor
r: Residuo de la división de **m/n**

El pseudocódigo de la implementación sería:

```
Euclides(m, n)
    r = 0
    while n != 0
        r = m % n
        m = n
        n = r
    return m
```

Experimentación y Resultados

Suma binaria

Gráficas de tamaño vs tiempo

Las siguientes gráficas muestra el desempeño del algoritmo con diferentes tamaños **r** de arreglos con respecto al tiempo.

En el eje de las abscisas se colocarán los valores de **r** y en el de las ordenadas los valores del tiempo **t** en μs.

Gráfica e impresión en consola de los valores de $k(2^k)$ desde 0 hasta 8

```
 --- Pares Ordenados(tamaño vs tiempo) ---
1. (1,1)
2. (2,0)
3. (4,0)
4. (8,1)
5. (16,1)
6. (32,1)
7. (64,3)
8. (128,5)
9. (256,12)
```

Figura 1: Resultado en consola de los pares ordenados de (r,t)

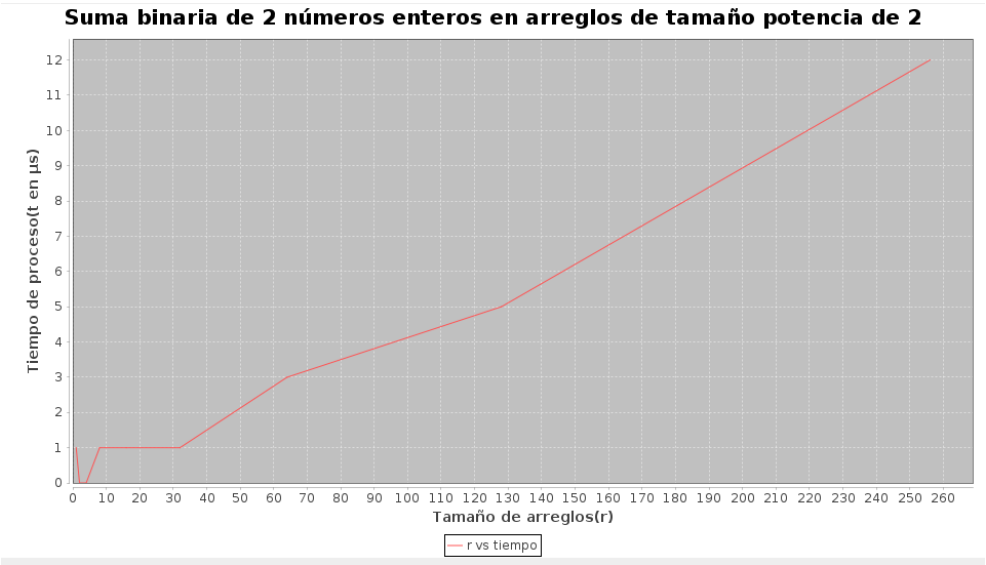

Figura 2: Gráfica con tendencía a complejidad lineal θ**(n)**

Se puede observar en la gráficación de los pares ordenados que los primeros valores de k tienden a variar de forma caótica por lo que no nos aporta información relevante sobre una aproximación real de la complejidad del algoritmo. Sin embargo se alcanza a divisar con los últimos valores un patrón que describiría la complejidad del algoritmo como lineal.

Gráfica e impresión en consola de los valores de $k(2^k)$ desde 0 hasta 15

```
 --- Pares Ordenados(tamaño vs tiempo) ---
1. (1,1)
2. (2,0)
3. (4,0)
4. (8,15)
5. (16,1)
6. (32,3)
7. (64,3)
8. (128,5)
9. (256,10)
10. (512,19)
11. (1024,39)
12. (2048,80)
13. (4096,157)
14. (8192,319)
15. (16384,592)
16. (32768,1224)
```

Figura 3: Resultado en consola de los pares ordenados de (r,t)

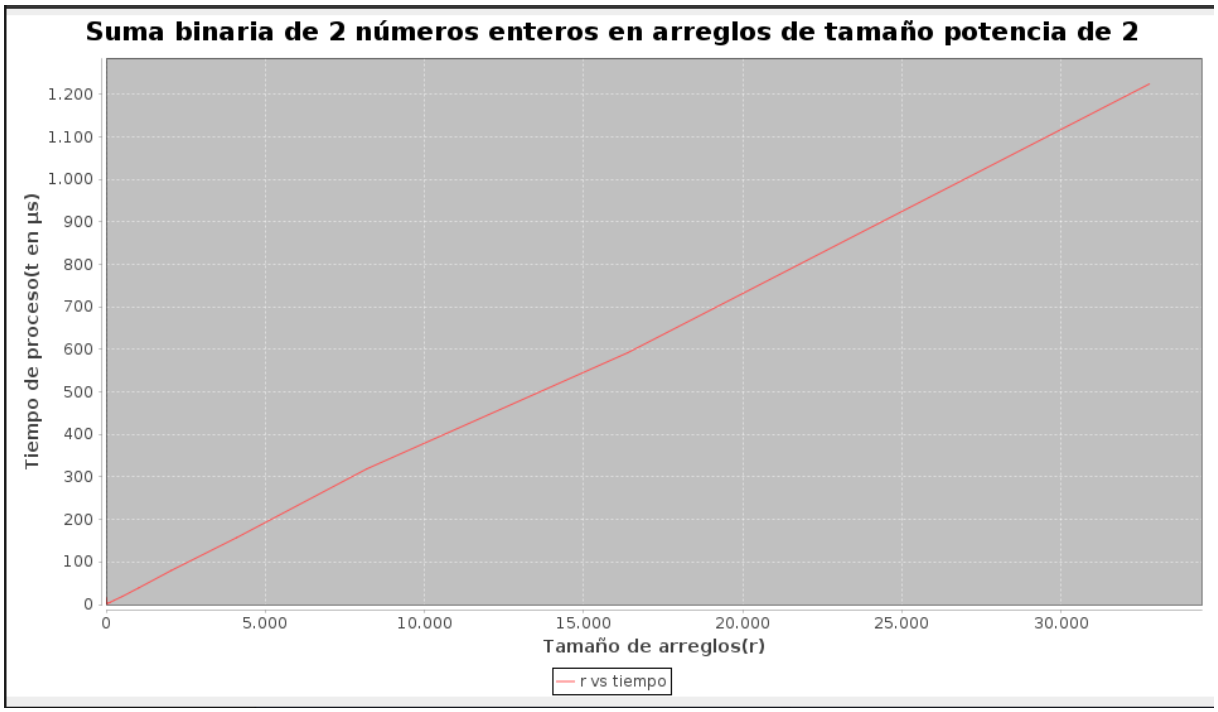

Figura 4: Gráfica con clara tendencia a complejidad lineal $\theta(\mathbf{n})$

Es claramente observable en esta graficación, una tendencia lineal con muy pocas anormalidades en el trazo de la recta.

Gráfica e impresión en consola de los valores de $k(2^k)$ desde 0 hasta 20

```
--- Pares Ordenados(tamaño vs tiempo) ---
1. (1,1)
2. (2,0)
3. (4,0)
4. (8,1)
5. (16,1)
6. (32,1)
7. (64,3)
8. (128,7)
9. (256,10)
10. (512,21)
11. (1024,49)
12. (2048,79)
13. (4096,165)
14. (8192,293)
15. (16384,609)
16. (32768,1227)
17. (65536,483)
18. (131072,576)
19. (262144,786)
20. (524288,1301)
21. (1048576,2660)
```

Figura 5: Resultado en consola de los pares ordenados de (r,t)

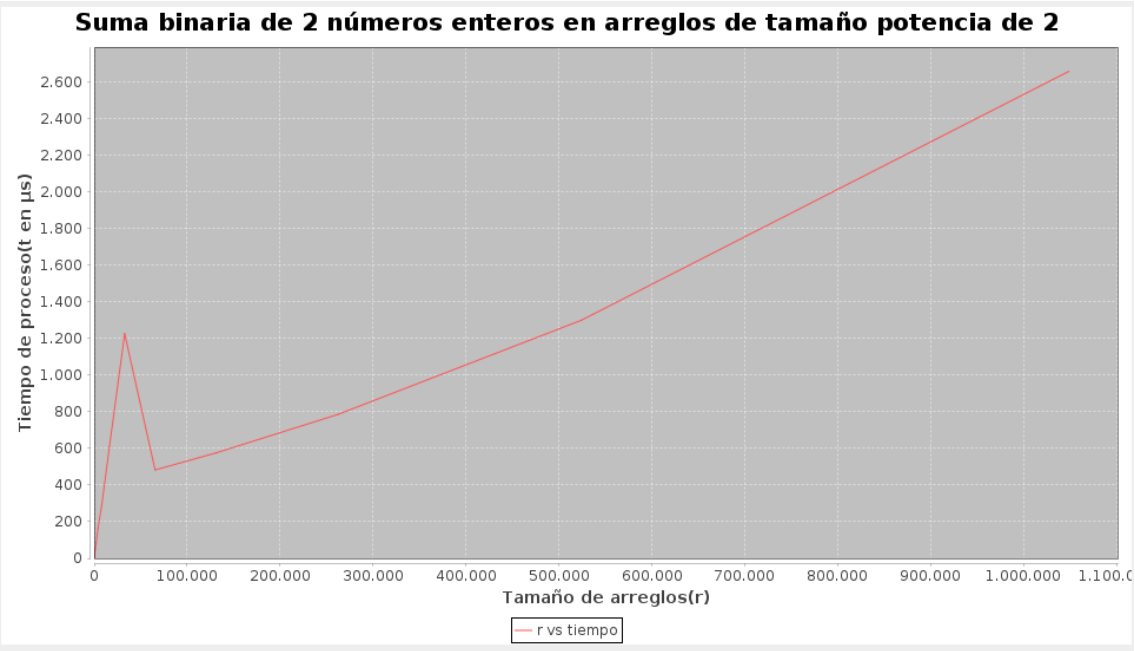

Figura 6: Gráfica con tendencia a complejidad lineal $\theta(\mathbf{n})$ con anormalidad

Para el último caso expuesto en este documento, se tiene una gráfica que toma valores de hasta 2^{20} en el eje de las abscisas. Es evidente la anormalidad surgida en el par ordenado *16*, que sin embargo no marca un punto de inflexión o cambio de la tendencia en la que crecen los demás pares, de los que no es necesario decir que crecen linealmente.

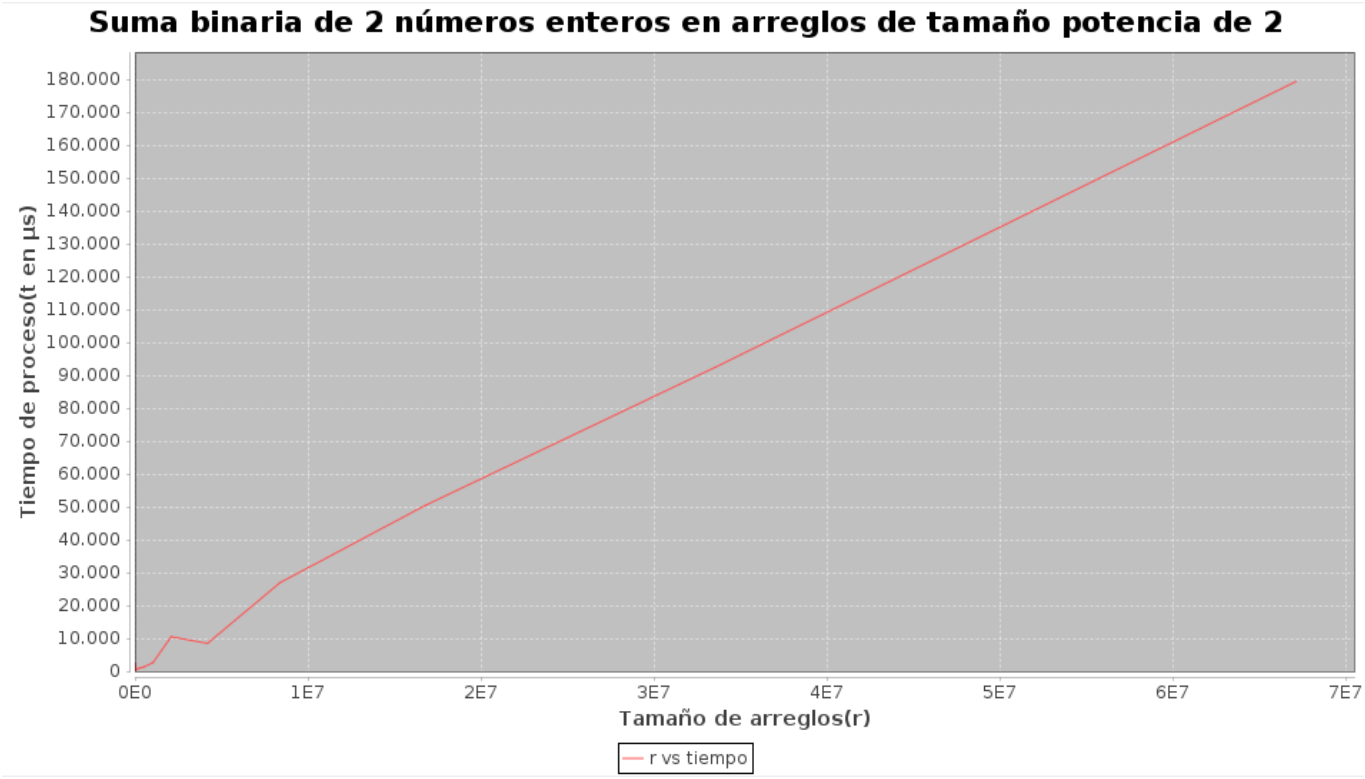

Figura 7: Gráfica resultante de valor $k = 27$

Análisis Asintótico

Como pudo verse en las gráficas de la sección anterior, el crecimiento descrito por el tiempo de ejecución para diferentes entradas al algoritmo de la suma binaria, muestran una clara similitud con una ecuación que la describe con un crecimiento lineal.

Derivado de la misma estructura del algoritmo, identificamos que no existe un mejor o peor caso, por lo que las representaciones mostradas pertenecen a un caso promedio de ejecución del algoritmo.

Basándonos en los puntos generados de la monitorización del desempeño con el mayor número de valores, se propone una ecuación:

$$f(n) = \frac{306}{8191}n$$

Obtenida al usar la ecucación de *recta-pendiente* $y = mx + c$, y la pendiente se calculo con el primer par ordenado $(1, 1)$ y el último $(32765, 1225)$. Aplicando la ecuación $m = \frac{y_2 - y_1}{x_2 - x_1}$, el valor es $m = \frac{306}{8191}$ Bajo la premisa de proponer una función *g(n)* tal que $Suma \in O(g(n)$ y la *g(n)* sea mínima, se propone una de las rectas familias de f(n), a la cual se le suma una constante y que mantenga todos los puntos que exceden a la recta *f(n)* por debajo de esta.

El cálculo exacto de la constante c, se encontró al calcular la distancia del punto externo más lejano a la *f(n)*, y se identificó el par ordenado *(4096,165)*. Se utiliza la ecuación para medir la distancia desde la recta a un punto que no este sobre esta:

$$D = |\frac{Ax_1 + By_1 + c}{\sqrt{A^2 + B^2}}| = |\frac{(306)(4096) - (8191)(165) + 0}{\sqrt{(306)^2 + (-8191)^2}}|12$$

De esta forma podemos concluir que c(Curva color rojo) puede tomar los valores de:$c \geq 12$. Con la $g(n) = \frac{306n}{8191} + 12$ cumple su función como cota superior asintótica para toda n mayor que 0.

Sin embargo, debido a la naturaleza de su cálculo donde se toma un punto exacto de una corrida, y puede haber otro punto en otra corrida que exceda esta *g(n)*. Se propone otra cota que contempla los mas de los puntos y no es demasiado superior a la *f(n)*. Esta *g(n)*(Curva color azul) es $g(n) = \frac{x}{10}$, cumplirá para todos los casos con *n¿8*, obteniendo esta limitación no de la curva *f(n)* propuesta, pero de los pares ordenados que muestran puntos con n¡8, exterior a la *g(n)*.

Por lo tanto:

$$g_1(n) = \frac{306n}{8191} + 12$$

$$g_2(n)\frac{x}{10}$$

$$f(n) = \frac{306n}{8191} \in O(g_1(n)) \in O(g_2(n))$$

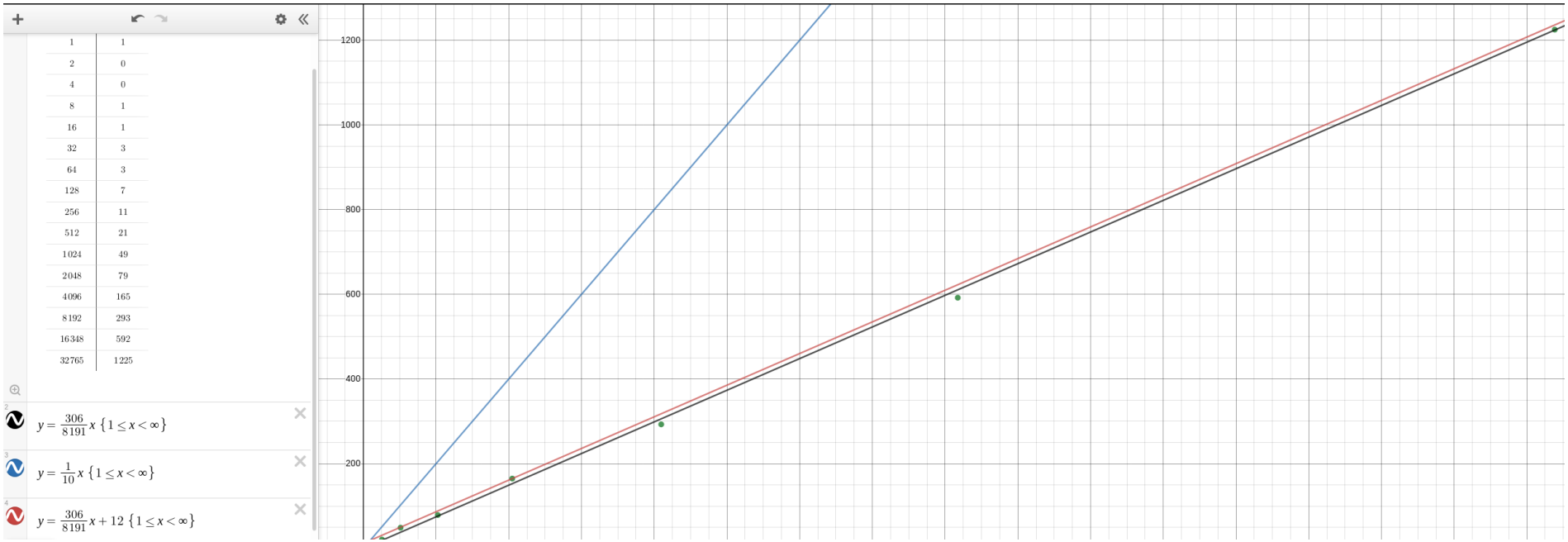

Figura 8: Dos gráficas propuestas para la acotación de la función propuesta del comportamiento de los pares ordenados

Algoritmos Euclidiano

Gráficas de tamaño vs tiempo

Las siguientes gráficas muestra el desempeño del algoritmo con diferentes valores de **m,n** con respecto al tiempo.

En el eje de las abscisas se colocarán los valores de **m y n**, en el de las ordenadas los valores del tiempo **t** en μs.

Con la finalidad de mostrar el peor caso de desempeño en este algoritmo se utilizan los valores consecutivos de la sucesión de Fibonacci.

Gráfica e impresión en consola de los pares ordenados obtenidos de la evaluación de los números en pareja que van desde la primera posición hasta la posición 20 en la sucesión de Fibonacci

```
 --- Pares Ordenados(m/n vs tiempo) ---
1. (0/1,312)
2. (1/2,423)
3. (3/5,459)
4. (8/13,405)
5. (21/34,479)
6. (55/89,573)
7. (144/233,743)
8. (377/610,893)
9. (987/1597,947)
10. (2584/4183,868)
```

Figura 9: Resultado en consola de los pares ordenados de (m/n,t)

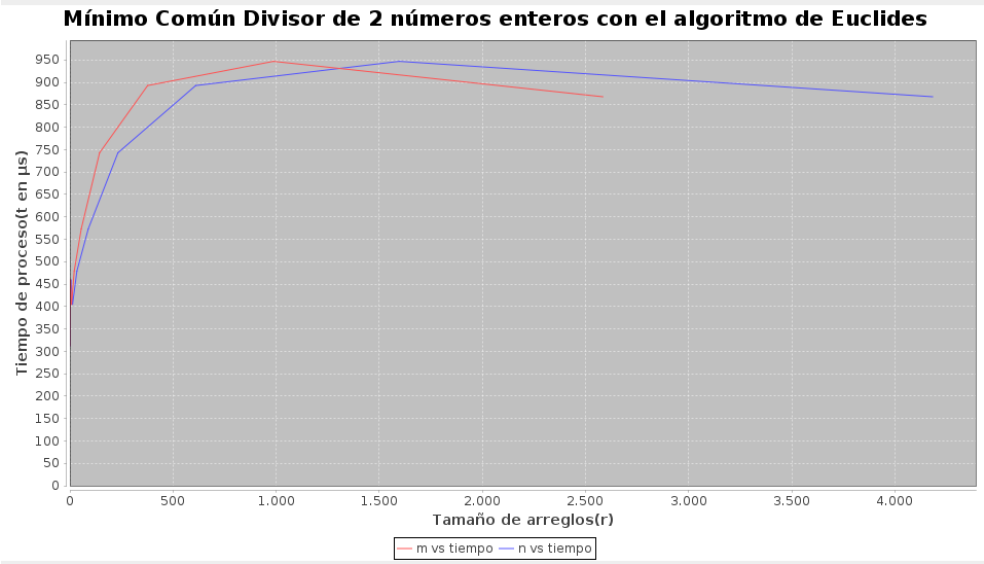

Figura 10: Gráfica con tendencía a complejidad logarítmica **log(n)**

Se aprecia en la gráfica una curva característica parecida a la de un logaritmo con un desplazamiento en el eje del tiempo.

Gráfica e impresión en consola de los pares ordenados obtenidos de la evaluación de los números en pareja que van desde la primera posición hasta la posición 30 en la sucesión de Fibonacci

En perspectiva con la gráfica obtenida anteriormente con un número menor de corridas, se puede apreciar una curvatura menos delicada con una tendencia a un crecimiento muy lento con respecto al eje de las ordenadas.

```
 --- Pares Ordenados(m/n vs tiempo) ---
1. (0/1,269)
2. (1/2,505)
3. (3/5,429)
4. (8/13,472)
5. (21/34,449)
6. (55/89,425)
7. (144/233,542)
8. (377/610,535)
9. (987/1597,530)
10. (2584/4183,529)
11. (6767/10950,495)
12. (17717/28667,557)
13. (46384/75051,526)
14. (121435/196486,621)
15. (317921/514407,657)
```

Figura 11: Resultado en consola de los pares ordenados de (m/n,t)

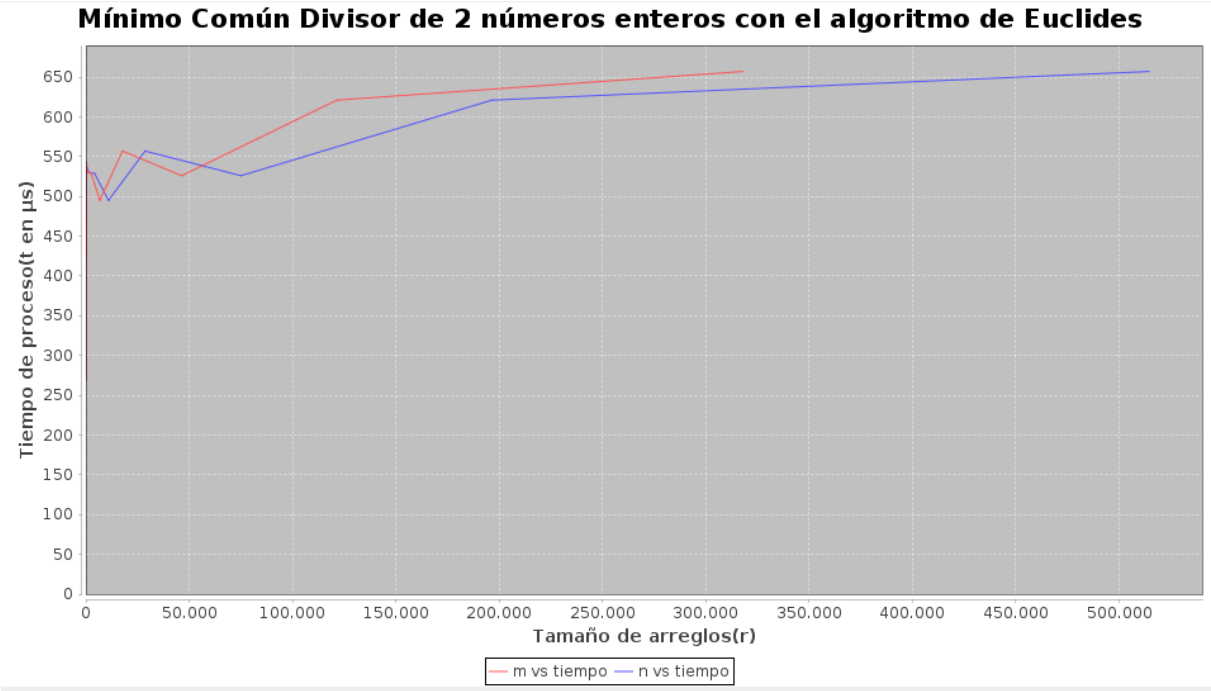

Figura 12: Gráfica con tendencia a complejidad logarítmica **log(n)**

Gráfica e impresión en consola de los pares ordenados obtenidos de la evaluación de los números en pareja que van desde la primera posición hasta la posición 40 en la sucesión de Fibonacci

Para el último caso expuesto en este documento, se tiene una gráfica que toma los primeros 40 valores generados de la sucesión de Fibonacci. En esta gráfica es ahora completamente claro un comportamiento de crecimiento rápido en el eje de las abscisas y en su contraparte, un crecimiento sumamente lento en el de las ordenadas. Esta evidencia nos permite afirmar que el algoritmo tendrá una complejidad en su peor caso de entrada de forma logarítmica **O(n)**.

Como pudo verse en las gráficas de la sección anterior, el crecimiento descrito por el tiempo de ejecución para diferentes entradas al algoritmo de la suma binaria, muestran una clara similitud con una ecuación que la describe con un crecimiento lineal.

```
 --- Pares Ordenados(m/n vs tiempo) ---
1. (0/1,413)
2. (1/2,711)
3. (3/5,696)
4. (8/13,605)
5. (21/34,647)
6. (55/89,726)
7. (144/233,933)
8. (377/610,799)
9. (987/1597,574)
10. (2584/4183,566)
11. (6767/10950,539)
12. (17717/28667,625)
13. (46384/75051,551)
14. (121435/196486,620)
15. (317921/514407,614)
16. (832328/1346735,678)
17. (2179063/3525798,716)
18. (5704861/9230659,719)
19. (14935520/24166179,738)
20. (39101699/63267878,798)
```

Figura 13: Resultado en consola de los pares ordenados de (m/n,t)

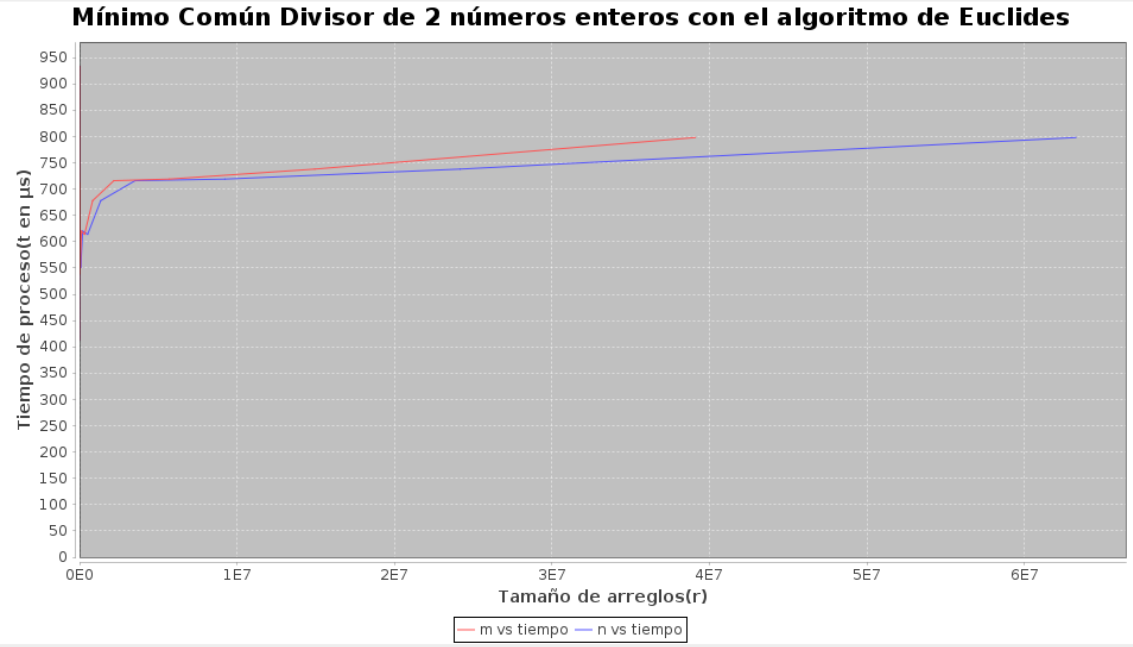

Figura 14: Gráfica con una clara tendencia a una complejidad **log(n)**

Análisis Asintótico

En comparación con el algoritmo empleado para la suma binaria, este algoritmo si cuenta con un peor caso, y este se muestra en las gráficas anteriores, donde se ingresan parejas de números que son consecutivos en la suseción de Fibonacci.

Basándonos en los puntos generados de la monitorización del desempeño con el mayor número de valores, se propone una ecuación que necesitará de una constante que la traslade de manera positiva en el eje de las ordenadas:

$$f(n) = log(n) + 800$$

Estas curva propuesta para la *f(n)*, al no ser una propuesta completamente precisa, ya mantiene los puntos generados por el programa, debajo de la curva. De esta manera ya cumple con la función de cota superior asintótica, sin embargo, para efectos visuales de la captura, se propone otra ecuación, la cual bien hubiera podido tomar un valor con constante c en la ecuación $g(n) = clog(n) + 800$, de $1 < c < \infty$. Por simplicidad y practícidad se propone $g(n) = 2log(n) + 800$, que por lo anterior explicado ya contiene a todos los posibles casos de ejecución por debajo de los límites que marca.

Por lo tanto:

$$g(n) = 2log(n) + 800$$

$$f(n) = log(n) + 800 \in O(g(n)$$

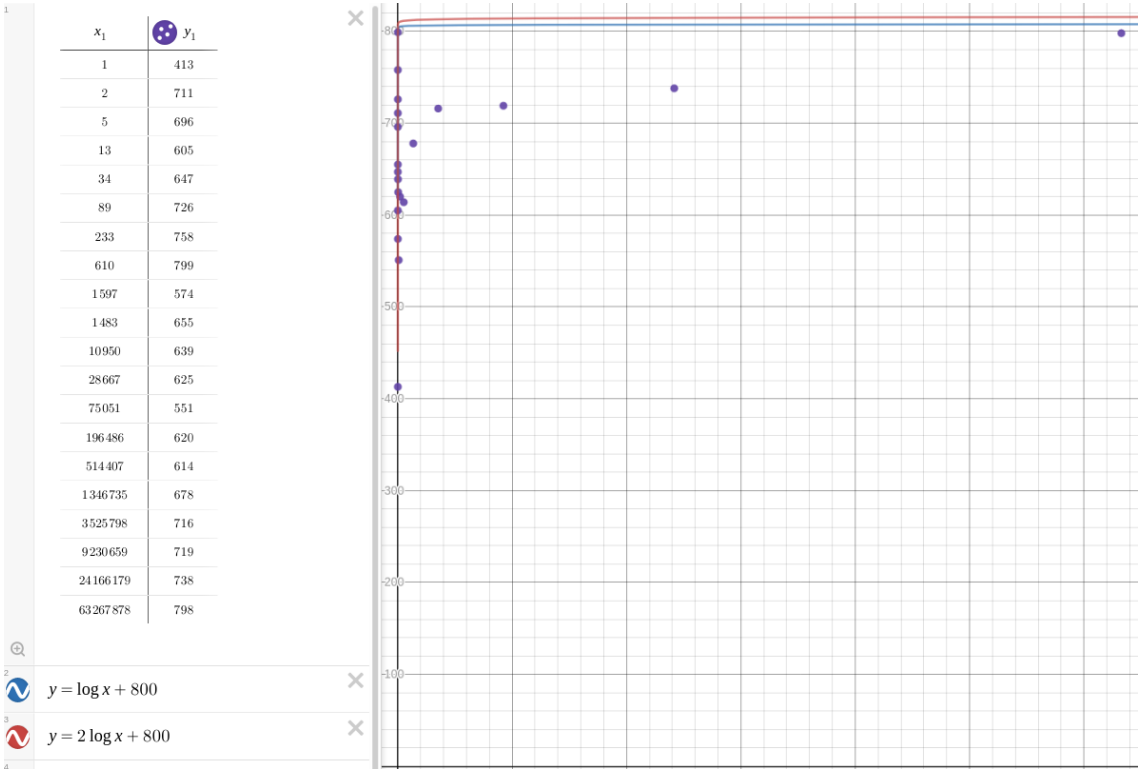

Figura 15: Curva propuesta para la acotación de la *f(n)* propuesta del comportamiento de los pares ordenados

Conclusiones

Rivero Ronquillo Omar Imanol

Al inicio del diseño cuando nos encontrabamos explorando las posibilidades de implementación del algoritmo de la suma binaria, el primer planteamiento involucraba la utilización de 4 if's para rellenar el arreglo de salida, sin embargo, encontramos más conveniente utilizar el algoritmo que habíamos visto en clases de diseño digital, reduciendo en gran medida la cantidad de código del algoritmo y posiblemente la complejidad del mismo. Posteriormente, nos trajo algunas complicaciones la forma de pasar los parametros a la función y esta hiciera las operaciones de forma automática sin tener que ingresar las cadenas inmensas para efectuar las sumas, al final utilizamos un random para generar los numeros y un archivo externo para indicar el tamaño que alcanzaria la potencia k. A la hora de la comprobación experimental, notamos que no había un peor caso para el algoritmo.

En el caso del algoritmo de Euclides, se utilizó el algoritmo proporcionado por el profesor. Y utilizamos la misma idea de considerar los valores externos para operar dentro de la función. El peor caso que comprobamos de forma experimental, fue cuando ingresabamos numero que son consecutivos en la sucesión de Fibonacci.

En conclusión, me pareció realmente interesante explorar la complejidad algoritmica de forma experimental, cosiderando todos los datos que nos fueron arrojados gracias a la forma en la que fue implementada la problematica en nuestro código.

Valle Martínez Luis Eduardo

La realización de la práctica nos permitió aplicar en un contexto real, las definiciones utilizadas para analizar la complejidad de un algoritmo.

Durante la elaboración de la práctica me enfrenté a algunos problemas derivado principalmente, del desuso de mi parte del lenguaje Java por ya algunos meses. Unido ha esto, existió al principio una confusión con el primer algoritmo y se generaron más métodos que realizaban tareas inecesarias, como la conversión de un número de entrada a cadena binaria. Tiempo más tarde se corrigieron y eliminaron estas funciones, tomando solo en cuenta las cantidades potencias que darían tamaño a los arreglos binarios, y el llenado del arreglo fue una simple función con random.A excepción de esto no hubo mayor complicación en la codificación.

Ya durante la ejecución, la primera vez que las funciones eran llamadas tardaban una cantidad de tiempo anormalmente grande, en comparación de las siguientes ejecuciones, y esto sin se dependientes del valor de entrada. Para evitar esto, se realizó dentro del código, una llamada a las funciones por primera vez, sin contabilizarlas en el registro de pares ordenados. Mostrándo un patrón consistente con los valores esperados de cada entrada.

Otra anomalía que no fue arreglada, se presentó en la graficación de los resultados del primer algoritmo con un conjunto de más de 15 valores y menor a 20, mostrando un pico extraño que desentonaba con la tendencia de comportamiento.

Las gráficas mostradas si tuvieron un resultado aproximado a los valores que esperabamos del desempeño de cada algoritmo.

Finalmente, es posible mejorar el segundo algoritmo mediante el uso de memoización, al igual que en la implementación de algunos algoritmos que genera el valor de la n posición en la sucesión de Fibonacci. Sin embargo esta aproximación mostraría su mejor optimización(con respecto a los recursos y tiempo de procesamiento) cuando la entrada son, precisamente, números pertenecientes a la sucesión de Fibonacci.

Bibliografía

[1] Luna,B.[Benjamín Luna].(2020, Septiembre, 29). CLASE 2 - ANÁLISIS DE ALGORITMOS[Archivo de video]. Recuperado de https://www.youtube.com/watch?v=tqzxCTHfv7g.

[2] Luna,B.[Benjamín Luna].(2020, Septiembre, 30). CLASE 3 - ANÁLISIS DE ALGORITMOS[Archivo de video]. Recuperado de https://www.youtube.com/watch?v=ojiIVGUU-vI.

[3] Cormen,T., y Balkcom,D.,(2017). Notación θ grande (Big-θ). Khan Academy. Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation?modal=1.

[4] Cormen,T., y Balkcom,D.,(2017). Notación O grande (Big-O). Khan Academy. Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation?modal=1.

[5] Cormen,T., y Balkcom,D.,(2017). Notación Omega grande (Big-Ω). Khan Academy. Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-omega-notation?modal=1.