Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2022-2, 3CV11, Práctica 1, 2 de marzo de 2022

Práctica 1: Determinación experimental de la complejidad temporal de un algoritmo

Luis Francisco Renteria Cedillo, Denzel Omar Vazquez Perez.

*lrenteriac*1400@*alumno.ipn.mx*, *dvazquezp*1600@*alumno.ipn.mx*

Resumen: En el presente documento se plantean dos problemas junto con su respectiva solución cuyo objetivo es determinar el orden de complejidad temporal de los algoritmos dados. Posteriormente se presentará una gráfica que muestre una función que acote los límites de los pares ordenados que presentaron ambos algoritmos dadas unas condiciones iniciales.

Palabras Clave: Complejidad temporal, Euclides, Fibonacci, C.

1 Introducción

Un algoritmo se puede definir como una serie de instrucciones que representan un modelo para solucionar ciertos tipos de problemas. Los algoritmos se utilizan en gran parte de las ciencias de la computación. Es por eso que el diseño de algoritmos requiere una amplia gama de conocimientos de programaciń y aplicar en múltiples casos la creatividad.

Eficacia vs Eficiencia: un algoritmo es eficaz si puede encontrar una solución a un problema determinado, sin embargo, esta capacidad puede volverse muy compleja cuando se busca una solución óptima. Es aquí cuando entra la eficiencia que se define como la relación entre recursos utilizados de un sistema y los logros conseguidos con el mismo. Dicho de otra forma, un buen algoritmo es correcto, pero un gran algoritmo es correcto y además eficiente.

Dado este enfoque, el análisis de algoritmos toma un papel importante a la hora de desarrollar software, ya que, podemos calcular una aproximación del comportamiento de un algoritmo. Con esto podemos comparar diversos algoritmos y escoger el que mejor se adapte para lograr nuestro propósito.

2 Conceptos Básicos

2.1 Complejidad algorítmica

La complejidad algorítmica es una medida de cuánto tiempo tarda en completarse un algoritmo dada una entrada de tamaño n, calculando el resultado dentro de un límite de tiempo finito y práctico incluso para valores grandes de n. Si bien la complejidad generalmente se expresa en términos de tiempo también se analiza en términos de espacio, lo que se traduce en los requisitos de memoria del algoritmo.

2.1.1 Complejidad temporal

Se define como la eficiencia en tiempo siendo el numero de pasos ejecutados para resolver un problema con una entrada de tamaño n, se denota por T(n).

2.1.2 Complejidad espacial

Al igual que la anterior, existe la eficiencia en espacio que hace referencia a la cantidad de recursos de memoria que utiliza el algoritmo para resolver un problema con una entrada de tamaño n, se denotada por S(n).

2.2 Notación Θ

La notación Theta (Θ) es un modo de denotar lo que es el límite asintótico de la tasa de crecimiento del tiempo de la ejecución de un algoritmo considerando el límite superior e inferior. Básicamente define el comportamiento asintótico exacto, se puede describir en términos formales como:

$$\Theta(g(n)) = \{f(n) \mid \exists_n\ C_1,\ C_2 > 0\ \&\ n_0 > 0\ tal\ que$$

$$0 \leq\ C_1 g(n)\ \leq\ f(n)\ \leq\ C_2 g(n)\ \forall\ n \geq\ n_0\}$$

2.3 Notación $\mathcal{O}$

Dado un $f(n)$ que muestra la complejidad de un algoritmo en función de una entrada de tamaño n , la expresión $f(n) \in \mathcal{O}(g(n))$ indicando que $f(n)$ no crece más de prisa que $g(n)$, donde esta ultima es denominada cota superior asintótica. $\mathcal{O}(g(n))$ es un conjunto que que se puede describir en términos formales como:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists_n\ C_1 > 0\ \&\ n_1 > 0\ tal\ que$$

$$0 \leq\ f(n)\ \leq\ C_1 g(n)\ \forall\ n \geq\ n_1\}$$

2.4 Notación Ω

La notación Ω es lo inverso a $\mathcal{O}$, por lo que se puede definir como una función que sirve de cota inferior de otra función cuando el argumento tiende a infinito, en términos formales se puede describir como:

$$\Omega(g(n)) = \{f(n) \mid \exists_n \; C_1 > 0 \;\&\; n_1 > 0 \; tal \; que$$
$$0 \leq \; C_1 g(n) \; \leq \; f(n) \; \forall \; n \geq \; n_1\}$$

2.5 Algoritmo para comparar dos arreglos

Siendo un problema computacional que debe encontrar de un arreglo A con n elementos enteros positivos, un elemento K, donde si $K \; \in$ al subarreglo $A[0, ..., {}^{n}/_{2} - 1]$ tanto a $A[{}^{n}/_{2}, ..., n-1]$, se devuelve el valor del elemento K.
El pseudo-código para la implemetación de este es:

Busqueda(A,n):

```
for i←0 to n/2 − 1 do

    for i←n /2 to n − 1 do

        if A[i]==A[j] then
            Mostrar A[i], i, j
        end if
    end for
end for
Mostrar "No se encontraron coincidencias"
```

2.6 Algoritmo de Euclides

Es un algoritmo eficiente para encontrar el *mcd* (máximo común divisor) de dos números enteros positivos m y n. Este algoritmo fue propuesto por el matemático griego Euclides, en el libro *"The Elements"*.
El pseudo-código para la implemetación de este es:

Euclides(m, n):

```
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
end while
return m
```

3 Experimentación y Resultados

3.1 Algoritmo para comparar dos arreglos

El primer algoritmo, genera un numero aleatorio par que determina el tamaño de un arreglo A, una vez hecho esto, se genera múltiples nḿeros aleatorios que tienen un valor desde 0 hasta 3n, mismos que se utilizan para rellenar dicho arreglo A. Se crea la función de búsqueda que tiene como argumentos el arreglo A y su longitud n. Posteriormente se codifica dos ciclos for anidados, el primero controla el índice del subarreglo de la primera mitad y el segundo ciclo controla el índice del subarreglo correspondiente a la segunda mitad del arreglo. Si la condición de que el valor en el índice i es igual al valor en el índice j, la función se detiene y regresa tanto los índices como el valor repetido.

Ejecutando el código se obtiene un conjunto de pares ordenados, el eje de las abscisas representa la longitud n y el eje de las ordenadas el número de operaciones. Estos datos son guardados en un archivo csv.

A continuación se observa la gráfica de n vs número de operaciones:

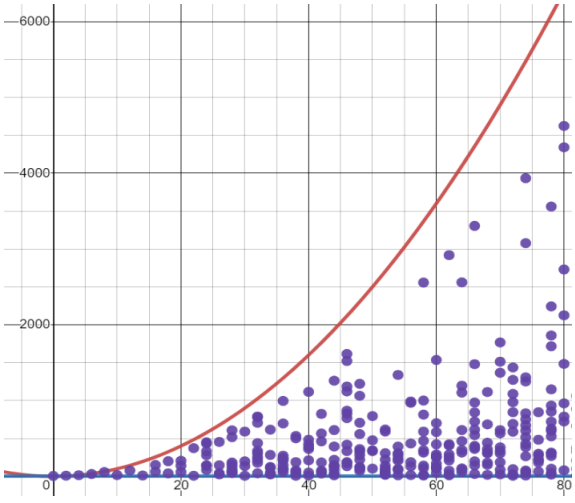

Figura 1: Gráfica de n vs número de operaciones

Se observa en el plano cartesiano que el algoritmo de búsqueda tiene, en el peor caso, un comportamiento de una función cuadrática. La cota superior g(n)= n^2 mientras que la cota inferior es la constante de 4, representando el mejor caso. Por lo tanto, el peor caso esta dado por T(n)=$n^2 \in O(n^2)$ y el mejor caso por T(n)=4 $\in \Omega(1)$

A continuación se muestra la salida de ejecucion del codigo, podemos apreciar los valores del arreglo y el valor repetido en sus posiciones i,j. Incluso podemos ver que la ultima ejecución no encontro coincidencias.

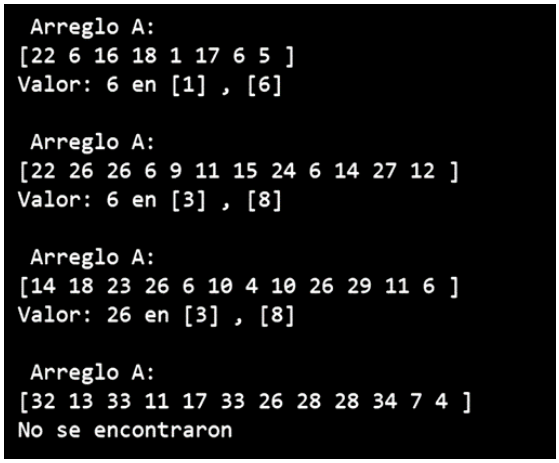

Figura 2: Resultados de los casos prueba en el algoritmo de Búsqueda.

3.2 Algoritmo de Euclides

Posteriormente al ejecutar el programa con la función creada con el nombre *"Euclides"*, se tiene que la asignación de valores de entrada m y n esta dada por la secuencia Fibonacci, por tanto se obtiene el número de operaciones, los resultados registrados se muestran en la Tabla 1.

Tabla 1: Complejidad temporal del algoritmo de Euclides

m	**n**	**T(n)**
1	1	6
1	2	10
2	3	14
3	5	18
5	8	22
8	13	26
13	21	30
21	34	34
34	55	38
55	89	42
89	144	46
144	233	50

Al considerar como peor caso los valores m y n de la Tabla 1, se procede a graficar los puntos sobre el plano cartesiano.

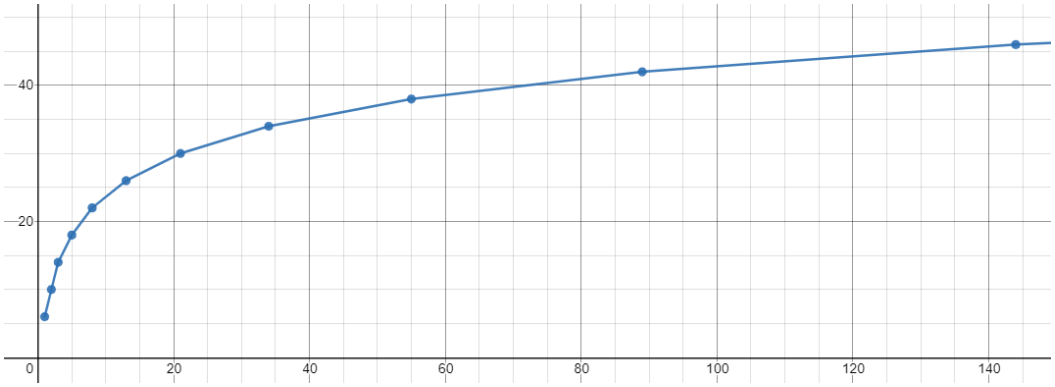

Figura 3: Resultados de los casos prueba en el algoritmo de Euclides.

A partir de la Figura 3 se observa que el comportamiento del el peor caso es logarítmico por tanto T(n) = K log(n),con el fin de acotar por arriba a la función original, se tiene que 20 log(n) es una cota superior, por lo que todo resultado experimental se encuentra acotado por abajo por T(n)=6 $\in \Omega(1)$ (mejor caso) y por arriba por T(n)=20 log(n) $\in O(\log n)$ como se ve en la Figura 4.

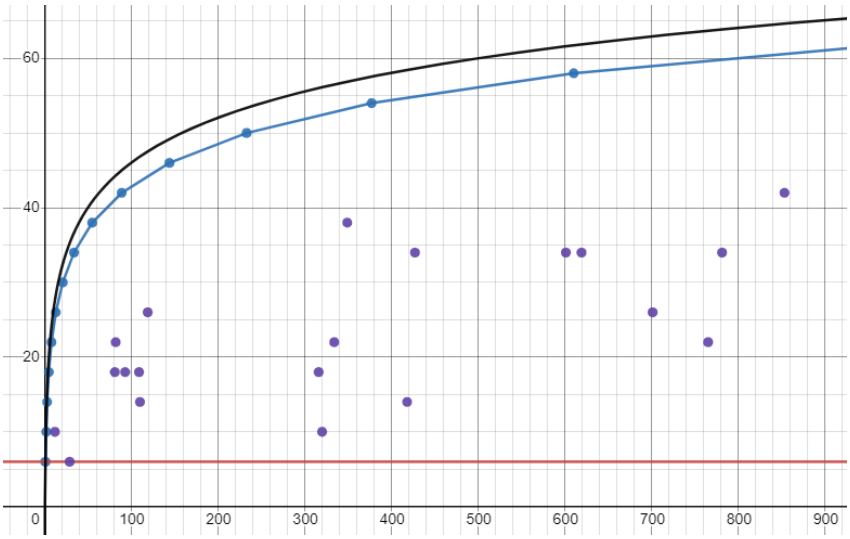

Figura 4: Gráfico del peor y mejor caso para el algoritmo de Euclides

4 Conclusiones

Luis Francisco Renteria Cedillo

Un algoritmo debe analizarse para determinar la cantidad de recursos que utiliza en el sistema. Es importante mencionar que existen diversas formas de medir la eficiencia de un algoritmo, pero el arquitecto del sistema debe dar prioridad sobre una, como por ejemplo, complejidad temporal por encima de complejidad espacial. En el análisis del algoritmo, fue vital comprender cuando una instrucción es ejecutada aunque no se vea explícitamente en el código. Como ejemplo tenemos que un ciclo for realiza una ejecución extra antes de salir del bucle, donde se verifica que una condición no se cumple, y por tanto, termina el ciclo.

Denzel Omar Vazquez Perez

El pensamiento y codificación de cada algoritmo influye directamente en la complejidad que tendrá este, siendo de ejemplo el primer problema de la

búsqueda de un elemento similar en los sub-arreglos previamente creado, donde los dos puntos anteriormente dichos son los intermediarios para determinar que tan eficaz sera la solución del problema, en este caso se propuso mostrar el peor caso con complejidad $O(n^2)$, sin embargo es importante mencionar que el haciendo uso de de la la estructura de datos hash es posible llegar a una complejidad $O(n)$. Para el segundo problema es sencilla su implementación haciendo el seguimiento del algoritmo proporcionado en la practica.

5 Bibliografía

Brassard, G. (1997). *Fundamentos de Algoritmia*. España: Ed. Prentice Hall.

Cormen, E. A. (2022). *Introduction To Algorithms*, 3Rd Ed. Phi.

Devopedia. 2022. *Algorithmic Complexity.* Version 8, February 19. Accessed 2022-02-19.

https://devopedia.org/algorithmic-complexity Luna, Benjamín. *Fundamentos para el analisis de eficiencia algorítmica*. Escuela Superior de Computo, IPN. México. 28 de febrero de 2022.

Sánchez, P. J. I. (2006). *Análisis y diseño de algoritmos: un enfoque teórico y práctico*. Servicio de Publicaciones y Divulgación Científica de la UMA.