TLATEMOANI
Revista Académica de Investigación
Editada por Eumed.net
No. 12 – Abril 2013
España
ISSN: 19899300
revista.tlatemoani@uaslp.mx

Fecha de recepción: 17 de enero de 2013
Fecha de aceptación: 20 de marzo de 2013

LA NOTACIÓN ASINTÓTICA EN EL CÓMPUTO CIENTÍFICO

José A. Cárdenas-Haro
Armando J. Vargas-Figueroa
Alberto F. Maupome-Polanco
Facultad de Ingeniería
Universidad Autónoma de Baja California
{antonio.cardenas, armando.vargas, amaupome}@uabc.edu.mx

Resumen

En el presente artículo se hace un análisis de la importancia de la notación asintótica en las ciencias de la computación y de como determinar el tiempo de ejecución matemático en segmentos de código de programas en diferentes situaciones. No en todos los casos es primordial buscar la optimización de los algoritmos dadas ciertas circunstancias que enumeramos. Se hace además una comparación de los diferentes tipos de límites asintóticos.

Palabras clave: Algoritmos, Complejidad Computacional, Análisis Asintótico, Cómputo Científico, Optimización.

Abstract

In this article an analysis is done about the importance of the asymptotic notation in computer science and on how to determine the mathematical execution time on code segments of programs in different situations. The algorithms optimizations are not necessary in every case; it depends on some circumstances that we describe here. Furthermore we do a comparison of the

different kinds of asymptotic limits.

***Keywords**: Algorithms, Computational Complexity, Asymptotic Analysis, Scientific Computing, Optimization.*

1 Introducción

Hoy en día con los grandes avances logrados en los últimos años en la informática y en la tecnología en general, es cada vez más común para la gente tener acceso a computadoras y al Internet. Un elemento clave en todo esto es precisamente el software o programas de computadora, dependiendo de la eficiencia de estos se obtendrá el máximo provecho del hardware en beneficio de los usuarios. Por ello es importante tener una métrica adecuada para la evaluación de la eficiencia de los algoritmos que son la base para la escritura del código de computadora, independientemente del lenguaje de programación a utilizarse. Esta métrica es la notación asintótica, que viene siendo una representación matemática redondeada de la cantidad de pasos o ciclos máquina requeridos para completar cierta tarea, asignada a través de un programa. El tiempo reloj no es adecuado para esto ya que dependiendo de la carga, de la cantidad de memoria primaria y del tipo de procesador (*Pentium II* vs Pentium *Core i7*) las variaciones en tiempo reloj pueden ser muy significativas.

2 Importancia de la notación asintótica

Por ejemplo, como notación escribir $n^2 + 5n + 4$ es demasiado detallado y se presta a confusiones, esto se vuelve más grave en ecuaciones con mas elementos; lo cual crea la necesidad de utilizar una notación más genérica, sin dejar de ser representativa de la cantidad de recursos computacionales requeridos por la función. Así nace la notación asintótica. En matemáticas la asíntota es una línea que continuamente se aproxima a una curva dada pero nunca la cruza o intersecta. Lo más relevante en el uso de la notación asintótica es para determinar la eficiencia de los algoritmos, es decir, sirve para

estimar matemáticamente la cantidad de recursos requeridos en la ejecución de un programa [5] [1] [10] [7] [9] [11]. Específicamente aquí nos estamos refiriendo a recursos de cómputo como son uso del procesador y memoria primaria y/o secundaria. Este tipo de notación se vuelve necesaria ya que no hay una computadora estándar o máquina modelo ideal universal para ser tomada como referencia para medir el tiempo de corrimiento de los algoritmos a usarse en programas de computadora. El análisis de algoritmos también es una herramienta para los diseñadores para estimar si una solución propuesta satisface las restricciones de recursos de un problema. La consideración principal para estimar el desempeño o eficiencia de un algoritmo es la cantidad de operaciones básicas requeridas por dicho algoritmo para procesar o resolver un problema dada una entrada de datos de cierto tamaño. Aquí el concepto de *razón de crecimiento* es muy importante para analizar y comparar algoritmos. La *razón de crecimiento* es la tasa a la cual el costo computacional de un algoritmo se incrementa conforme la cantidad de datos de entrada aumenta. Usualmente la cantidad de datos de entrada se especifica con la letra *n,* que indica el número o cantidad de elementos a procesar por el algoritmo para darnos la solución o respuesta que buscamos.

Por ejemplo, consideremos a los algoritmos de ordenamiento, muy utilizados cotidianamente en el mundo para ordenar grandes cantidades de información, ya sea por orden alfabético, cronológicamente, o por valores de menor a mayor (o viceversa); estos tipos de ordenamientos son muy utilizados por los gobiernos, bancos y grandes empresas en general. Existen diferentes técnicas para dichos ordenamientos, el método más sencillo y antiguo es el de la burbuja (o *Bubblesort*), que también es el más ineficiente, donde cada elemento se compara con todos para definir su posición destino; es decir, se requieren *n* comparaciones para cada uno de los *n* elementos, esto es n^2 comparaciones o pasos, lo cual se define asintóticamente como $O(n^2)$. Existen otros algoritmos de ordenamiento que son mucho más eficientes así como complicados como es el caso del *Quicksort*, *Mergesort*, *Heapsort*, *Bucketsort*, entre otros. Por ejemplo, el tiempo de corrimiento o ejecución del

algoritmo *Quicksort* es $O(n\log n)$ para el caso promedio (el peor caso es $O(n^2)$ pero la probabilidad de que este ocurra es casi cero). El tiempo de ejecución de otros algoritmos como *Mergesort* o *Heapsort* es $O(n\log n)$ siempre, sin embargo en la práctica *Quicksort* termina en promedio más rápido los ordenamientos.

Esto se debe precisamente a las constantes que se omiten en los análisis de los tiempos de ejecución, el término $O(n\log n)$ es un valor general y redondeado que manifiesta solo la parte del algoritmo que consume el máximo de los recursos (mas detalles en la sección 4).

3 ¿Es realmente importante buscar siempre el algoritmo óptimo?

Es común que en los algoritmos haya factores constantes, esto es, que no se ven afectados por la cantidad de datos a procesarse; a estos datos también se les conoce como “datos de entrada” o simplemente “entrada”. A manera de ejemplo, analizando la razón de crecimiento entre las funciones $f_1 = 9n, f_2 = 21n$ y $f_3 = 3n^2$ vemos que solo para valores de $n > 7$ la function f_3 supera a las otras. Trazando estas funciones vemos que f_1 y f_2 son líneas rectas que nunca se cruzan, es decir, se mantienen paralelas. Para un mismo algoritmo, el tamaño de la entrada (o cantidad de datos a analizarse) puede variar bastante según sea el problema a resolver, es por ello y también para simplificar el análisis, que se ignoran las constantes. Lo más importante es enfocarse en examinar la razón o tasa de crecimiento, a esto se le conoce como análisis asintótico. Sin embargo es importante recalcar que hay ciertas circunstancias específicas en donde la razón o tasa de crecimiento no es el criterio primordial a considerar a la hora de diseñar o comparar programas o algoritmos. Mencionamos a continuación algunas situaciones donde se presenta este dilema.

- Cuando la cantidad de datos a usarse en el programa son pocos, la razón o tasa de crecimiento no es tan importante, especialmente cuando

las constantes ocultas son grandes o muy grandes. Bajo estas condiciones podría ser más rápido un algoritmo de $O(n^2)$ que uno de $O(n)$.

- Hay circunstancias como en el caso de los algoritmos numéricos, donde la estabilidad y la exactitud son primordiales y la rapidez queda en segundo término . En estos casos es preferible sacrificar eficiencia y no afectar la precisión en los resultados [6].

- Cuando un programa va a ser ejecutado en muy raras ocasiones, no tiene caso invertir mucho tiempo en optimizarlo menos aún si la mejora en el tiempo reloj de su ejecución no es significativa. Bajo estas circunstancias es mejor basarnos en el algoritmo que sea más sencillo de implementar a la hora de escribir el programa.

- Una desventaja de escribir un programa complicado y de muchas líneas de código para implementar el algoritmo más eficiente posible, además de todo el tiempo que se le tiene que invertir, es que en un futuro probablemente será otra persona la encargada de darle mantenimiento y depurarlo. Esto podría resultar en grandes pérdidas de tiempo posteriores.

- Actualmente las computadoras personales tienen una capacidad de procesamiento que unas décadas atrás nadie se hubiese imaginado. Ni las super computadoras de aquel entonces tenían la capacidad que tienen ahora las computadoras de escritorio o las laptops. Bajo las actuales circunstancias no hay mucha diferencia en el tiempo reloj de ejecución de un algoritmo respecto de otro. Cierto programa en equis computadora podrá tardar cinco segundos en resolver un problema, mientras que otro programa más eficiente podría requerir solo de dos segundos en dar los mismos resultados en la misma computadora. Por

tres segundos de mejora en el tiempo reloj, la mayoría de la gente no se tomará la molestia de reescribir todo un programa.

Sin embargo, en el cómputo científico donde se manejan cantidades enormes de datos en super computadoras, generalmente es muy importante trabajar con los algoritmos más eficientes posibles; ya que cada minuto de uso de las supercomputadoras representa miles de dólares. En estos casos como la cantidad de datos de entrada a los programas es muy grande, la razón o tasa de crecimiento del algoritmo a usarse afecta bastante el tiempo reloj total de ejecución, en el rango de las horas, días o hasta meses según sea el caso.

4 Determinando el tiempo de ejecución

El tiempo de corrimiento o de ejecución de un algoritmo computacional es la cantidad de ciclos de CPU (Unidad Central de Procesamiento por sus siglas en Inglés) que se requieren para resolver el problema en cuestión. Sin embargo, en la determinación del tiempo asintótico de ejecución se recurre al redondeo en base a las secciones del código o programa que mas ciclos de CPU consumen. Normalmente los programas de computadora, especialmente los grandes, se componen de varias partes ya sean funciones u objetos, los cuales tienen cada una su determinado tiempo de ejecución. Para definir el tiempo de ejecución total hay que utilizar la regla del tiempo mayor de cualquiera de las partes. Por ejemplo, es correcto decir que $n^3 - 3n^2 + n - 7 \in O(n^3)$ aunque $n^3 - 3n^2 + n - 7 < 0$ cuando $n \leq 3$. Otro ejemplo más general sería con un programa de computadora que se componga de tres partes que sean inicialización, procesamiento y finalización. Supongamos que estas tres partes requieren un tiempo de $O(n^2), O(n^3)$ y $O(n \log n)$ respectivamente. Entonces el tiempo total de corrimiento de dicho programa de computadora sería $O(n^2 + n^3 + n\log n) = O(max(n^2, n^3, n\log n)) = O(n^3)$. En otras palabras, el tiempo de corrimiento de un algoritmo o programa de computadora está determinado por la parte que mas consume recursos o tiempo de procesamiento [3]. En cómputo todo este consumo de recursos se define en funciones positivas, no pueden ser funciones negativas.

Es algo así como las resistencias en los circuitos eléctricos, las resistencias negativas no existen a menos que estas contengan una fuente de poder, de lo contrario se violaría la ley de conservación de la energía. Usando funciones negativas en el análisis asintótico del tiempo de corrimiento de un algoritmo o programa de computadora, corremos el riesgo de cometer errores como el siguiente:

$O(n) = O(n + n^3 - n^3) = O(max(n, n^3, -n^3)) = O(n^3)$. Para el tiempo de ejecución de los algoritmos se utiliza la notación asintótica para definir el peor caso, el mejor caso y el caso promedio.

4.1 Cálculo del tiempo de ejecución de programas de computadora

Para determinar el tiempo de ejecución en notación asintótica de un programa, hay que identificar primero los lazos o ciclos y analizarlos de adentro hacia afuera en su ejecución.

Por ejemplo, analicemos el siguiente segmento de código escrito en lenguaje Python:

```
(1)   for i in range(0, n):
(2)   for j in range(0,n):
(3)     datos[i][j]=1
```

La línea 3 se ejecuta en un tiempo constante u $O(1)$ y de acuerdo a la línea 2 del código esto se repite *n* veces que en total es $O(n)$, a su vez las líneas 2 y 3 en conjunto se iteran otras *n* veces. Es decir, un total de $O(n^2)$ operaciones para llenar la matriz con unos. Analicemos ahora este otro segmento de código en Python:

```
(1)   i=0
(2)   while z != factor[i]:
(3)   i=i+1
(4)     if i>n: break
```

Este segmento de código busca a lo largo de un vector de datos un elemento con el mismo valor que *z*. Es probable que encuentre dicho elemento en el primer paso, o en el último, o nunca. Como tenemos que basarnos siempre en el peor caso, decimos que este segmento de código se ejecuta en un tiempo de orden $O(n)$.

4.2 Tiempo de ejecución de programas en paralelo

Hoy en día gracias a los avances tecnológicos es cada vez más común el cómputo paralelo y/o distribuido. Esto es que en lugar de usarse un solo procesador para la ejecución de un programa, se utilizan varios o muchos. Esto acelera el tiempo de terminación de la mayoría de los programas [2] [10]. Por ejemplo, la suma de *n* números es $O(n)$ si se hace de manera secuencial; pero en paralelo digamos con *n*/2 procesadores, el tiempo de ejecución se reduce a $O(\log n)$. Esto es porque cada uno de estos procesadores puede sumar de manera simultanea dos números y reducirse el total de sumandos a la mitad en cada iteración, obteniendo así el gran total de la suma en solo log *n* pasos. Otro ejemplo es con los algoritmos de ordenamiento donde el *Quicksort* se ejecuta en un tiempo promedio de $O(n\log n)$ de manera secuencial, esto es $O(n)$ comparaciones $O(\log n)$ veces [4]. En paralelo tenemos la ventaja de que las $O(n)$ comparaciones pueden ejecutarse en $O(1)$ quedando el tiempo promedio total en solo $O(\log n)$. La búsqueda del elemento mayor en un vector de datos se puede realizar de manera secuencial en un tiempo $O(n)$, en paralelo este tiempo se reduce a $O(1)$ ya que todas las comparaciones necesarias pueden hacerse de manera simultanea. En paralelo además del orden del tiempo de ejecución, se habla del orden de trabajo. Si bien el tiempo de ejecución se reduce, al final el trabajo es el mismo solo que distribuido entre varios o muchos procesadores. Podemos decir que el orden de trabajo de un algoritmo en paralelo es igual al orden del tiempo en su versión secuencial. En la sección 4.1 vemos como analizar el tiempo de ejecución en programas secuenciales, en paralelo esto cambia sustancialmente.

4.3 El uso de *T*(*n*) para el análisis

Lo importante aquí es saber como se incrementa el tiempo de corrimiento respecto al aumento en los datos de entrada. Normalmente se utiliza la function $T(n)$ como base al hacer el análisis para determinar el tiempo de ejecución en un algoritmo; esta se utiliza para representar las recurrencias en base a la cantidad *n* de datos a procesarse. Comúnmente en un programa de computadora vamos a tener iteraciones o recursiones, esto es, una función es ejecutada en múltiples ocasiones ya sea porque es llamada desde otra función o porque esta se llama a sí misma. Por ejemplo, en la recurrencia $T(n) = 2\mathrm{T}(n/2) + n$ se utiliza la técnica divide y vencerás y sirve para el análisis de algoritmos como en la FFT (Transformada Rápida de Fourier, por sus siglas en Inglés, la cual tiene muchas aplicaciones en física e ingeniería [12]), o en algoritmos de ordenamiento como el *Quicksort* o *Mergesort*. Si queremos saber cual es su orden de tiempo de ejecución, tenemos que resolverlo recursivamente. Por simplicidad y sin perder generalidad, supongamos que $n = 2^k$ donde $k \geq 1, k \in N$. Expandiendo esta recurrencia tenemos:

(1) $$T(n) = 2\left[\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right] + n$$

Desarrollando y reduciendo la ecuación obtenemos

(2) $$T(n) = 4T\left(\frac{n}{4}\right) + n + n$$

Haciendo una recursión mas tenemos

(3) $$T(n) = 8T\left(\frac{n}{8}\right) + n + n + n$$

Este proceso de expansión lo podemos hacer log *n* veces, resultándonos

$$T(n) = nT(1) + \sum_{i=1}^{\log n} n = n\left[T(1) + \sum_{i=1}^{\log n} 1\right] = n[1 + \log n] = \Theta(n \log n) \quad (4)$$

Estas recurrencias se pueden resolver también usando el teorema maestro [8], aunque este tiene sus limitaciones y no puede ser utilizado en todos los casos.

5 Los límites asintóticos

La notación *O* es únicamente para definir los límites superiores, en el total de recursos requeridos por determinada función o algoritmo de computadora; como se explica en la sección 4. Hay otros tipos de notación asintótica que se explican a continuación.

5.1 Notación Omega mayúscula

Contrario a la notación *O*, la notación Ω se usa para determinar los límites inferiores en el consumo de recursos. $f(n) = \Omega(g(n))$ o lo que es lo mismo $f(n) \in \Omega(g(n))$ si existe una constante real $c > 0$ y una constante entera $n_0 \geq 1$ tal que $f(n) \geq cg(n)$ para cada entero $n \geq n_0$.

5.2 Notación Theta

La notación Θ nos indica simultáneamente los límites superior e inferior de ejecución, cuando estos están determinados (en una misma función) por diferentes múltiplos reales positivos.

Es decir, $t(n) \in \Theta(f(n))$ si y solo si $t(n)$ pertenece simultáneamente a $O(f(n))$ y a $\Omega(f(n))$.

Expresado más formalmente decimos que $\Theta = (f(n)) = \Omega(f(n)) \cap O(f(n))$. A esto también se le conoce como orden exacto de $f(n)$ ya que su tiempo de

ejecución está acotado por arriba y por abajo al mismo tiempo.

5.3 Notación “o” pequeña

Esta notación es solo para definir que una función crece más despacio que otra de referencia.

La relación $f(n) \in o(g(n))$ nos dice que $g(n)$ crece mucho más rápido que $f(n)$ asumiendo que f y g son funciones de una variable. Como ejemplo podemos decir que $7n^2 \in o(n^3)$. En es$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ n formalmente está establecido que:

(5)

5.4 Notación Omega minúscula

La notación ω muy raramente se usa en ciencias de la computación. La relación de ω con Ω es similar a la relación de la “*o*” pequeña con la “*O*” grande en términos asintóticos.

La relación $f(n) = \omega(g(n))$ se cumple si para cualquier constante real $c > 0$, existe una constante entera $n_0 \geq 1$ tal que $f(n) > cg(n)$ para cada entero $n \geq n_0$.

Esto implica que:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad (6)$$

Por ejemplo, $\frac{n^3}{7} \notin \omega(n^3)$, pero $\frac{n^3}{7} \in \omega(n^2)$.

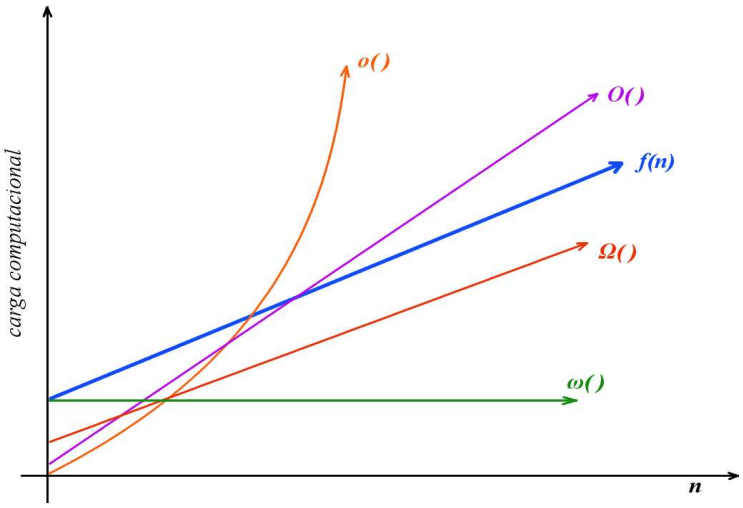

Figura 1. La presente gráfica muestra una comparación entre los diferentes tipos de notaciones, mostrándose como aumenta la carga computacional respecto al incremento de los datos de entrada.

6 Conclusiones

La notación asintótica es de suma importancia en ciencias de la computación para determinar el tiempo de ejecución de los algoritmos y/o hacer comparaciones entre ellos. Sirve de parámetro de referencia estándar, ya que no hay un modelo o máquina universal contra el que se puedan evaluar todos los algoritmos en su tiempo de ejecución o corrimiento.

Otro factor es que no es confiable usar el tiempo reloj como medida del tiempo de ejecución de los algoritmos. Esto se debe a que en una misma computadora para un mismo algoritmo, el tiempo reloj puede variar bastante dependiendo de diversos factores, como lo son el sistema operativo o la ejecución de ciertas interrupciones o procesos que se salen del control del usuario. Estas variaciones pueden ser bastante más significativas si se compara el tiempo de ejecución de programas en máquinas con capacidad distinta o con arquitecturas diferentes. Por ello la trascendencia en el uso de la notación matemática en todos los análisis de algoritmos.

Referencias bibliográficas

[1] Baase, S. (1988): "*Computer algorithms: introduction to design and analysis*" (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Bahi, J. M., Contassot-Vivier, S., and Couturier, R. (2007): "*Parallel Iterative Algorithms: From Sequential to Grid Computing*" (Chapman & Hall/CRC Numerical Analy & Scient Comp. Series). Chapman & Hall/CRC.

[3] Brassard, G., and Bratley, P. (1996): "*Fundamentals of algorithmics*". Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[4] Cárdenas-Haro, J. A., Galaviz-Yáñez, G., y Lopez-Mariscal, G. (2005): "*Análisis matemático del tiempo de corrimiento y aceleración del algoritmo de ordenamiento quicksort en paralelo*". Memorias de la XV Semana Regional de Investigación y Docencia en Matemáticas (2005), p. 27-32.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001): "*Introduction to Algorithms*", 2nd Ed. MIT Press, Cambridge, MA, 2001.

[6] Dussault, J.P. (1995): "*Numerical stability and eficiency of penalty algorithms*". SIAM J. Numerical Analysis. 32, 1 (1995), p. 296-317.

[7] Grenet, B., Koiran, P., and Portier, N. (2013): "*On the complexity of the multivariate resultant*". Journal of Complexity. Vol. 29, 2 (2013), pp. 142-157. Elsevier.

[8] Leighton, T. (1996): "*Notes on better master theorems for divide-and-conquer recurrences*". In Lecture notes, MIT (1996).

[9] Pan, L., and Prez-Jimnez, M. (2010): “Computational complexity of tissue-like p systems.” Journal of Complexity. Vol 26, 3 (2010), pp. 296-315. Elsevier.

[10] Parhami, B. (1999): “*Introduction to Parallel Processing: Algorithms and Architectures*”. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[11] Urli, T., Wagner, M., and Neumann, F. (2012): “*Experimental supplements to the computational complexity analysis of genetic programming for problems modelling isolated program semantics*”. In Parallel Problem Solving from Nature - PPSN XII, Vol. 7491 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 102-112.

[12] Wang, S. D., Xiao, J. Y., Liu, Z. Q., and Niu, J. L. (2008): “*Study on tire imprint image recognition based on FFT image differencing algorithm*”. In FSKD (2) (2008), J. Ma, Y. Yin, J. Yu, and S. Zhou, Eds., IEEE Computer Society, pp. 504-508.