

Eficiencia y complejidad

Santiago Burbano Puin
Ingeniería de Sistemas
sa.burbano@javeriana.edu.co

Julián Cárdenas García
Ingeniería de Sistemas
jandres.cardenas@javeriana.edu.co

Pontificia Universidad Javeriana
22 de agosto de 2022

1 Introducción

En las ciencias de la computación ha sido de gran importancia tratar con algoritmos que pueden tener un tiempo de ejecución de 10 segundos, hasta 10 años. De base, la eficiencia de un algoritmo depende de la entrada que se le introduzca; este tendrá una variación de tiempo t al incrementar la entrada n . Este, según la serie de operaciones que tenga, presentará una **Complejidad de tiempo**.

Tomemos de ejemplo un algoritmo de ordenamiento, de menor a mayor. Se introduce una lista con ocho elementos. $N=[11,10,7,9,5,4,1,3]$. El algoritmo se ejecuta tres veces, y nos da los siguientes resultados:

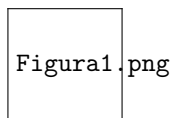


Figure 1: Ejecuciones del algoritmo de ordenamiento

En la *figura 1* se muestra que en todas las ejecuciones hubo variaciones en el tiempo. Esto se presenta gracias a factores que influyen en la práctica (procesamiento de la máquina, cantidad de operaciones, acceso de memoria, etc). Igualmente, es práctico encontrarle una descripción matemática que las simplifique. Cada ejecución puede ser expresada por una función en específico, aunque todas cumplen condiciones en las que permiten que sean clasificadas en una sola. Es para eso que está la **Notación asintótica**.

2 Complejidad de tiempo

La complejidad de tiempo $T(n)$ es expresada en términos de las operaciones usadas por el algoritmo cuando se introduce recursos de tamaño n . Cabe resaltar que está en términos de las operaciones ya que existen diferencias de tiempo entre máquinas para ejecutar un mismo algoritmo, según su poder de procesamiento.

En el siguiente ejemplo, vamos a usar un algoritmo que representa una sumatoria simple, para expresar su tiempo de complejidad:

```
int sumatoria(int n){  
    int total = 0;           //1  
    for(int i=1; i<=n; i++){ //1+(n+1)+n  
        total += i;         //n  
    }  
    return total;           //1  
}
```

- Se inicializa una variable total donde se almacenará constantemente las iteraciones de la sumatoria. Al igualar a cero se cuenta como una operación, donde se inicializa una única vez. (1)

- El ciclo for realiza una serie de condiciones y operaciones que permiten el recorrido de la sumatoria. Primero, inicializa la variable i para indicar la iteración que se encuentra, y se realiza una única vez (1). Cada vez que se llega a una nueva iteración se corrobora si llegó al fin del ciclo, en este caso llegando al valor de n ($n+1$). Por último, el for aumenta n veces el valor de iteración i (n). $(1+(n+1)+n)$
- En la variable total, estando en el for, se va sumando el valor de la iteración i n veces. (n)
- Se retorna el valor de la sumatoria. (1)

Finalmente, se suman todas las operaciones, para así obtener la complejidad de tiempo del algoritmo:

$$1 + 1 + (n + 1) + n + n + 1 = 3n + 4 = T(n)$$

De la anterior forma es como se obtiene la expresión de la complejidad de tiempo de un algoritmo. Esto se realiza de la misma manera en diferentes casos: ciclos anidados; sucesión de fibonacci mediante recursividad; búsqueda binaria, o accesos de memoria como la lista usada en el algoritmo de la figura 1.

3 Notación asintótica

La notación asintótica es una representación del límite máximo de tiempo que puede un algoritmo ejecutarse. Esta se obtiene a partir de la expresión de la complejidad de tiempo $T(n)$ de un algoritmo.

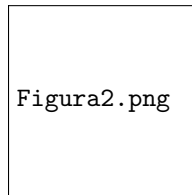


Figure 2: Crecimiento máximo del algoritmo de ordenamiento de la figura 1

En la figura 2 se muestran las mismas ejecuciones del algoritmo de ordenamiento usado en la figura 1. Todas estas tienen una complejidad de tiempo propio, pero aún así coinciden con una tendencia a tener un crecimiento de tiempo lineal. Por lo tanto, se determinó que la función $g(n)$ (recta lineal verde) es el límite de crecimiento de un algoritmo con $T(n)$. En este caso, se presenta una notación Big-O.

Existen varias notaciones asintóticas, sin embargo únicamente trataremos con Big-O y Little-o.

3.1 Big-O

Una complejidad de tiempo $T(n)$ se clasifica en una notación asintótica Big-O, donde se describe con una función de crecimiento $g(n)$, si y solo si existe una constante c positiva multiplicada por $g(n)$ es mayor o igual a $T(n)$, para todo n mayor o igual a n_0 :

$$T(n) = O(g(n)) \leftrightarrow T(n) \leq c \cdot g(n), n > 0, n \geq n_0$$

Con lo anterior, se dice que $g(n)$ domina $f(n)$, si y solo si es relación reflexiva y transitiva. Esto indica que no es una relación simétrica puesto que, aunque $g(n)$ domina $f(n)$, $f(n)$ no domina $g(n)$.

Esto puede demostrarse de la siguiente manera:

- f domina f , y puesto que $f = f$ se verifica que $f(n) \leq c \cdot f(n)$, donde $c = 1$. Por lo tanto, la relación es reflexiva.
- Tomemos en cuenta funciones f, g, h , donde g domina f , y h domina g . Esto verifica que $f(n) \leq c_1 \cdot g(n)$ y $g(n) \leq c_2 \cdot h(n)$, donde $c_1 > 0$ y $c_2 > 0$. Con esto se puede considerar que $f(n) \leq c_1 \cdot g(n) \leq c_1 c_2 \cdot h(n)$, y haciendo que $c = c_1 c_2$ se tiene que $f(n) \leq c \cdot h(n)$. Por lo tanto, la relación es transitiva.

Existen clasificaciones Big-O que se presentan comunmente en los algoritmos. Hay unos los cuales tienen un crecimiento bajo de la complejidad de tiempo, hasta otros que lo tienen bastante alto:

Propiedades:

Si $T_1(n) = O(g_1(n))$ y $T_2(n) = O(g_2(n))$, entonces:

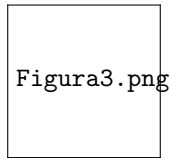


Figure 3: Gráfica representación notación asintótica

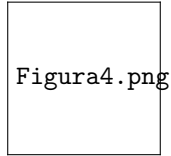


Figure 4: Notaciones Big-O comunes

- $T_1(n) + T_2(n) = O(\max(g_1(n), g_2(n)))$
- $T_1(n) \cdot T_2(n) = O(g_1(n) \cdot g_2(n))$

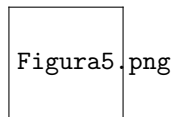
3.2 Little-o

La notación Little-o es igualmente de límite superior, aunque con una condición levemente deferente: la constante de c positiva multiplicada por la función de crecimiento $g(n)$ es estrictamente mayor a la complejidad de tiempo $T(n)$, para todo n mayor o igual a n_0 :

$$T(n) = o(g(n)) \leftrightarrow T(n) < cg(n), c > 0, n \geq n_0$$

3.3 Ejemplos

Ejemplo 1: Tenemos un algoritmo con complejidad de tiempo $T(n) = 2n^2$. Como vemos que el término de la función es de n^2 , es trivial decir que la función de crecimiento $g(n) = n^2$:

Figure 5: Límite máximo $g(n)$, de $T(n)$

Sin embargo, en la *figura 5* muestra que $g(n)$ no es mayor a $T(n)$, para todo $n \geq n_0$. Entonces, ¿qué sucede si lo multiplicamos por 4?

Al multiplicar, podemos ajustar la función decrecimiento $g(n)$ ser el límite superior de la complejidad de tiempo $T(n)$, para cumplir con la condición de Big-O. Finalmente, podemos concluir que $T(n) = 2n^2 = O(n^2)$.

Ejemplo 2: Tenemos un algoritmo con $T(n) = n + \log n + 1$. Al tener más de un término diferente, ¿cuál es su complejidad?. Para eso comparamos y elegimos el término con mayor crecimiento:

Vemos que el término con mayor crecimiento es lineal. Por lo tanto, podemos que decir que la función $g(n) = n$, ya que cumple con la condición de $T(n) \leq c \cdot g(n)$.

4 Cálculo del error

Vamos a imponer en un algoritmo que cuando sea posible es que pequeños cambios en los datos de entrada producen correspondientemente pequeños cambios en los resultados finales. Un algoritmo que satisface esta propiedad es estable; de lo contrario, inestable. Algunos presentan el caso de ser estables en ciertas selecciones de los datos de entrada:

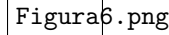
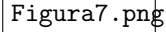


Figure 6: Límite máximo multiplicado por una constante

Figure 7: Complejidad de cada término de $T(n)$

condicionalmente estable.

Se puede considerar que se presenta crecimiento del error relacionado con la estabilidad del algoritmo, suponiendo un error $E_0 > 0$ que se introduce en alguna operación para el cálculo del resultafo final, y que E_0 en n operaciones después se denota como E_n .

- En un algoritmo estable, el crecimiento del error es de comportamiento lineal, donde $E_n \approx CnE_0$, donde C es una constante independiente de n .
- En un algoritmo inestable, el crecimiento del error es de comportamiento exponencial, donde $E_n \approx C^n E_0$.

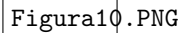


Figure 8: Crecimiento del error en algoritmos estables e inestables

5 Problema de aplicación: Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo utilizado en la Teoría de grafos, donde se busca la distancia de menor costo, recorriendo desde un nodo origen hasta el resto de nodos en un grafo.

A continuación, se muestra el pseudocódigo del algoritmo:

En el pseudocódigo se muestran elementos importantes que recalcar:

- La entrada del algoritmo es el grafo $G = (V, E)$ con el conjunto de vértices V , y el conjunto de aristas E . Por lo tanto, la cantidad de vértices y aristas son $|V|$ y $|E|$.
- Q es una cola que contiene los vértices no visitados.
- S es la cola con los vértices ya visitados.

Para este algoritmo, la complejidad algorítmica Big-O depende de las estructuras de datos usadas en la ejecución.

Caso 1

- Si el grafo $G = (V, E)$ es representado por una matriz de adyacencia.
- Si la cola de prioridad Q es representada por una lista no ordenada.

Ahora se indican las notaciones de las operaciones principales del algoritmo:

1. La cola de prioridad Q es inicializada llenándola con las $|V|$ vértices. $O(|V|)$
2. Al usar una matriz de adyacencia, se actualizan todas las distancias $|V|$ veces. $O(|V|)$

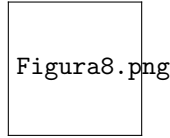


Figure 9: Pseudocódigo del algoritmo de Dijkstra

3. Se itera $|V|$ veces la cola Q , hasta que esta quede vacía. $O(|V|)$

Finalmente, la notación Big-O resulta: $O(|V|) + O(|V|) \cdot O(|V|) = O(|V|) + O(|V|^2) = O(|V|^2)$

Caso 2

- Si el grafo $G = (V, E)$ es representado por una lista de adyacencia.
- Si la cola de prioridad Q es representada por un montículo binario mínimo.

Ahora se indican las notaciones de las operaciones principales del algoritmo:

1. La cola de prioridad Q es inicializada llenándola con las $|V|$ vértices. $O(|V|)$
2. Al usar una lista de adyacencia, todos los vértices pueden ser recorridos mediante recorrido de anchura. se iteran todos los vecinos de cada vértice y se actualizan sus distancias, tomando un tiempo de $|E|$. $O(|E|)$
3. Se itera $|V|$ veces la cola Q , hasta que esta quede vacía. $O(|V|)$
4. Con el montículo mínimo, permite remover el vértice con menor distancia y recalculan las distancias en un tiempo de $\log|V|$. $O(\log|V|)$

Finalmente, la notación Big-O resulta: $O(|V|) + O(|E| \cdot \log|V|) + O(|V| \cdot \log|V|) = O(|V|) + O((|E| + |V|) \cdot \log|V|) = O(|E| \cdot \log|V|)$, desde que $|E| \geq |V| - 1$, siendo G un grafo conexo.

Comparando las notaciones Big-O de los dos casos, las funciones de límite de crecimiento resultaron de la siguiente manera:

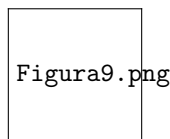


Figure 10: Límite máximo de cada uno de los dos casos

Evidentemente podemos ver que el segundo caso tiene una menor de complejidad computacional.

References

- [1] The Programming Art. (2020, 27 septiembre). ¿Cómo funciona la notación asintótica? Desde Big-O hasta Little-Omega [Vídeo]. YouTube. <https://www.youtube.com/watch?v=HcDV5MGGrRE>
- [2] Burden, A. M., Faires, J. D., & Burden, A. M. (2015). Numerical Analysis (10.a ed.). Cengage Learning.
- [3] Rosen, K. H. (2018). Discrete Mathematics and Its Applications (8.a ed.). McGraw-Hill Education.
- [4] Villalpando, J. Francisco (2003). Análisis asintótico con aplicación de funciones de Landau como método de comprobación de eficiencia en algoritmos computacionales. e-Gnosis, (1),0.[fecha de Consulta 20 de Agosto de 2022]. ISSN: . Disponible en: <https://www.redalyc.org/articulo.oa?id=73000115>
- [5] Barceló, P. (2009, 1 abril). Auxiliar 3: Ordenes, Funciones y Notación Asintótica. www.u-cursos.cl. Recuperado 20 de agosto de 2022, de https://www.u-cursos.cl/ingenieria/2009/1/CC3101/1/material_docente/bajar3Fid.material3D214494

- [6] B. (2021, 13 octubre). Understanding Time Complexity Calculation for Dijkstra Algorithm. <https://Www.Baeldung.Com>. Recuperado 20 de agosto de 2022, de <https://www.baeldung.com/cs/dijkstra-time-complexity>