Complejidad algorítmica

La complejidad algorítmica es una métrica teórica que nos ayuda a describir el comportamiento de un algoritmo en términos de tiempo de ejecución (tiempo que tarda un algoritmo en resolver un problema) y memoria requerida (cantidad de memoria necesaria para procesar las instrucciones que solucionan dicho problema). Esto nos ayuda a comparar entre la efectividad de un algoritmo y otro, y decidir **cuál es el que nos conviene implementar**.

A la idea del tiempo de ejecución se le conoce como **complejidad temporal**, y a la idea de la memoria requerida para resolver el problema se le denomina **complejidad espacial.** Dichos valores se encuentran en función del **tamaño del problema** (valor o valores dictados por el número de elementos con los que un algoritmo trabaja), aunque *en algunos casos no*. Por ejemplo, ¿tarda y consume lo mismo un algoritmo en ordenar cien o diez mil elementos existentes en memoria? Obviamente que no. Por tanto se habla de la complejidad algorítmica como una función de valor n, y no como un valor en específico.

Existen varias consideraciones antes de definir y entender correctamente la complejidad algorítmica. Imaginemos que estamos evaluando la complejidad temporal de un algoritmo, ¿tardará lo mismo en ejecutarse en un procesador Pentium III que en un Xeon E7? Y además, ¿estará el algoritmo ejecutando más o menos líneas escrito en Javascript que en C? Esto no quiere decir que mi algoritmo sea mejor o peor en un lenguaje u otro, pero sí que la complejidad temporal no llega a ser una buena métrica a secas para evaluar la efectividad, ya que parece obvio que cierto algoritmo tardará la mitad de tiempo en ejecutarse en un procesador lo doble de rápido. Entonces, ***¿cómo interpretar la complejidad de un algoritmo?***

La solución ideal para entender lo que realmente es la complejidad algorítmica es pensar en el **ritmo de crecimiento**, donde evaluaremos *cómo crece el número de instrucciones necesarias* para resolver el problema en función del tamaño del mismo. De esta manera nos olvidamos del lenguaje utilizado, el tiempo de ejecución, la cantidad de memoria consumida, el sistema en donde se corra el algoritmo y el estilo de programación implementado.

Podemos analizar el siguiente código escrito en Java para ilustrar lo anterior:

```
1
2  public class ejemplo1 {
3
4      final static byte[] n = { 4, 2, 5, 1, 6, 0, 6, 9, 5, 4, 5, 0 };
5      static int resultado = 0;
6
7      public static void main(String[] args) {
8
9          resultado = (n.length)*8;
10         resultado += 3;
11         System.out.println("El resultado es: "+resultado);
12
13     }
14
15 }
```

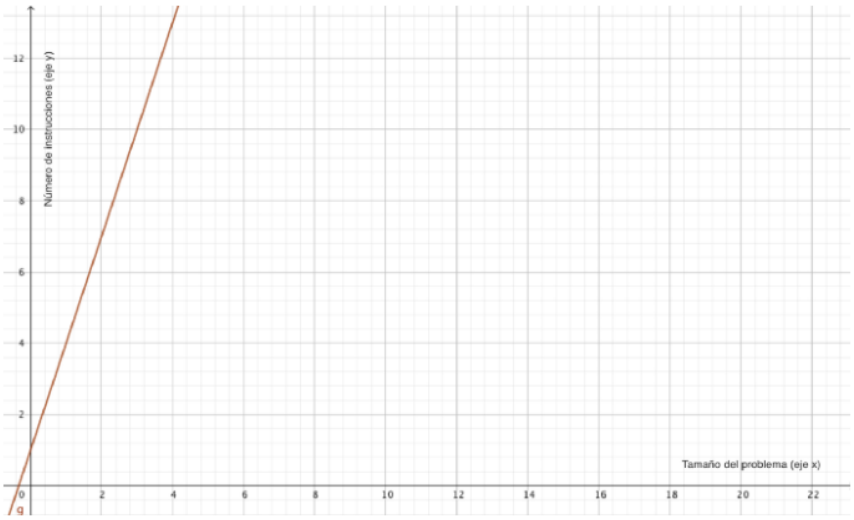
Al ser ejecutado, el algoritmo realiza tres instrucciones para resolver el problema planteado (líneas 9, 10 y 11), independientemente del tamaño de la entrada de datos n, este tan solo elabora un par de operaciones aritméticas. Por tanto, no importa el tamaño del problema, el ritmo de cambio se mantiene constante. La función estaría descrita por ƒ(x) = 3 y su complejidad algorítmica con la notación Θ(3).

Representación gráfica de la función $f(x) = 3$.

Continuemos con un segundo ejemplo:

```
1
2   public class ejemplo2 {
3
4       final static byte[] n = { 4, 2, 5, 1, 6, 0, 6, 9, 5, 4, 5, 0 };
5       static int resultado = 0;
6
7       public static void main(String[] args) {
8
9           for(int i=1; i<=n.length; i++){
10              resultado += i;
11          }
12
13          for(int j=n.length; j>=1; j--){
14              resultado *= 5;
15              resultado /= 2;
16          }
17
18          System.out.println("El resultado es: "+resultado);
19
20      }
21
22  }
```

Nota que el algoritmo consiste en dos bucles uno enseguida del otro, el primero ejecutando una instrucción (línea 10) y el segundo ejecutando dos instrucciones (líneas 14 y 15). Dichos bucles son ciclados n veces y finalmente seguidos por una instrucción mas (línea 18). Podemos inferir que el algoritmo ejecuta 1n + 2n + 1 instrucciones, es decir ƒ(x) = 3n + 1, por lo que, a diferencia del ejemplo anterior, observamos un ritmo de crecimiento lineal Θ(3n + 1).

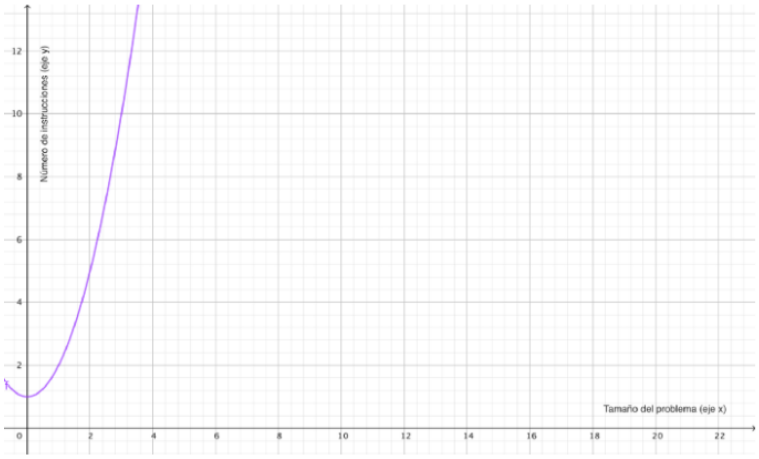


Representación gráfica de la función $f(x) = 3n + 1$.

Finalizaré brevemente la idea con un tercer ejemplo más:

```
1
2  public class ejemplo3 {
3
4      final static byte[] n = { 4, 2, 5, 1, 6, 0, 6, 9, 5, 4, 5, 0 };
5      static int resultado = 0;
6
7      public static void main(String[] args) {
8
9          for(int i=1; i<=n.length; i++){
10
11             for(int j=n.length; j>=1; j--){
12             resultado = i*7;
13             resultado /= 2;
14             }
15
16         }
17
18         System.out.println("El resultado es: "+resultado);
19
20     }
21
22 }
```

Observamos dos bucles for anidados, el primero ciclando n veces una instrucción for, que a su vez itera n veces dos instrucciones mas. En seguida, se tiene una instrucción de impresión y el algoritmo finaliza. Podemos concluir que ƒ(x) = 1n * 2n + 1, es decir ƒ(x) = $2n^2$ + 1.



Representación gráfica de la función $f(x) = 2n^2 + 1$, que en realidad corresponde a $f(x) = n^2 + 1$, pero para objetivos didácticos funciona, solo se abre un poco más.

Dicho esto, vemos que el número de instrucciones a ejecutar para que dicho algoritmo resuelva el problema incrementa rápidamente a partir de un conjunto de valores de tamaño cuatro, lo que en términos de complejidad, se dice que el algoritmo guarda una relación cuadrática $\Theta(2n^2 + 1)$.

De esta manera podríamos comprobar que la complejidad de un algoritmo se encuentra en función del tamaño del problema, pero como mencione hace un momento; *esto no siempre sucede así*, especialmente en los algoritmos de búsqueda y ordenamiento.

Los algoritmos de búsqueda, como su nombre lo indica, sirven para localizar un elemento o conjunto de elementos que guarden una relación o sean estrictamente iguales a otro elemento determinado.

Examinemos este último código para terminar de ilustrar el concepto:

```
1
2   public class ejemplo4 {
3
4       final static byte[] n = { 4, 2, 5, 1, 6, 0, 6, 9, 5, 4, 5, 0 };
5       static int numero_a_buscar = 1, i = 0;
6       static boolean encontrado = false;
7
8       public static void main(String[] args) {
9
10          while(encontrado==false){
11
12              if(n[i]==numero_a_buscar){
13                  System.out.println("El número coincidio en la posición "+i+".");
14                  encontrado = true;
15              }
16              i++;
17
18          }
19
20      }
21
22  }
```

El algoritmo consiste en encontrar un valor determinado dentro de un arreglo, para lo que cicla indefinidamente un conjunto de instrucciones, o mejor dicho, hasta que el valor a buscar sea encontrado. Por tanto, sería ambiguo expresar su complejidad algorítmica como una función determinada, ya que dicha función puede ser diferente en el mejor y peor de los casos.

El *mejor de los casos* refiere a que el valor a buscar se encuentre en el primer índice del arreglo, por tanto el algoritmo solo necesitaría de 4 instrucciones para solucionar el problema; $f(x) = 4$ y su complejidad algorítmica en el mejor caso esta descrita como $\Omega(4)$.

Análogamente, la complejidad algorítmica en el *peor de los casos* significaría que el valor a buscar se encuentre en el último elemento del arreglo, de esta manera, el bucle ejecutaría las cuatro instrucciones en su interior n veces; $f(x) = 4n$ y su complejidad algorítmica en el peor caso esta descrita como $O(4n)$.

En términos de notación, en la complejidad algorítmica se suele emplear una contracción que exprese de igual manera la naturaleza del algoritmo en cuestión, es decir, para una complejidad polinomial $\Theta(5n^2 + 8n + 3)$, se sintetizará la expresión tomando solo la literal con el mayor grado relativo, quedando expresado como $\Theta(n^2)$.

$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

Órdenes de complejidad comunes

Esta abreviatura es mejor conocida como orden de complejidad, y de esta manera logramos agrupar todas las complejidades que crecen de igual forma, es decir, que pertenecen al mismo orden.

En conclusión, la complejidad algorítmica o ritmo de crecimiento es una métrica que te permite como programador evaluar la factibilidad de las diferentes soluciones de un problema, y poder decidir con un argumento matemático cuál es mejor mediante comparaciones.

Este tipo de herramienta teórica tiene aparentemente poca utilidad en el día a día de un programador habitual, pues requiere de algoritmos mucho más complejos y pilas de datos muy grandes, y a menos que trabajes con Big Data, Machine Learning o hardware con recursos muy limitados como Arduino donde la optimización es crucial, la complejidad algorítmica tiene poca importancia a nivel práctico. Sin embargo, es un concepto bastante básico e importante de conocer, pues te permite dimensionar y pensar acerca del comportamiento de tus soluciones y la manera en la que los solucionas, y no solo codear un algoritmo que termine costándote más en un futuro.

Fuente:

https://medium.com/@joseguillermo_/qu%C3%A9-es-la-complejidad-algor%C3%ADtmica-y-con-qu%C3%A9-se-come-2638e7fd9e8c