Complejidades asintóticas.

Román Castellarin

2016

1. Introducción

En este apunte se va a introducir el concepto de **complejidad asintótica**. En las *Ciencias de la Computación* y en la programación en general, el análisis de la complejidad asintótica es muy importante, ya que nos permitirá responder preguntas del estilo:

- ¿Cuánto tiempo correrá un programa dada una entrada en particular?
- ¿Cuánta memoria ocupará?
- ¿Es resoluble mi problema en tiempo y memoria razonables?

Éstas son las bases para comparar diferentes programas, algoritmos y estructuras de datos. Comprender la complejidad asintótica de algo es comprender su **eficiencia**. En ambientes de programación competitiva, esto es de vital importancia, ya que usualmente tenemos limitaciones en **tiempo de ejecución** y **memoria**.

1.1. Ejemplo práctico

Supongamos que nos dan un problema computacional que, como siempre, debemos resolver; en el cual debemos procesar una cantidad n de registros. Se nos ocurren dos soluciones distintas que llamaremos T_1 y T_2, y programamos ambas. Luego confeccionamos una tabla con sus tiempos de ejecución:

n	10	20	30	40	50
T_1	5s	10s	15s	20s	25s
T_2	0.3s	1.2s	2.7s	4.8s	7.5s

Cuadro 1: Tiempos de ejecución de ambos programas

Notemos que el segundo algoritmo *pareciera* ser significativamente más rápido que el primero. Sin embargo, nos hubiera gustado poder **predecir** la tabla primero para poder luego implementar directamente el algoritmo más eficiente. Para esto vamos a analizar los algoritmos antes de programarlos.

Cuando analizamos complejidades, lo hacemos en función del **tamaño de la entrada**, en nuestro caso, n. Por lo general, dados dos algoritmos distintos, o ambos rinden en tiempo aproximadamente muy similar (y en ese caso elegimos el más fácil de implementar) o uno es evidentemente más veloz que el otro. Tratando de extrapolar los valores de la tabla, habríamos llegado a las siguientes leyes por pura observación:

$$T_1(n) = 0{,}5s \cdot n$$
$$T_2(n) = 0{,}003s \cdot n^2$$

Decimos que el primer algoritmo es *lineal*, ya que demora proporcionalmente al tamaño de la entrada, mientras que el segundo es *cuadrático*, por razones obvias. A la constante de proporcionalidad la llamamos *constante computacional* y depende de cada algoritmo.

Al medir la eficiencia de un algoritmo, tendemos a comparar el **peor tiempo de ejecución**. Supongamos que el máximo número de registros que nos pueden dar es $n = 2000$, entonces el primer algoritmo tardaría $T_1(2000) = 1000s$, mientras que el segundo tardaría $T_2(2000) = 12000s$. ¡El primer algoritmo corre muchísimo más rápido!

Mientras más grande sea n, más ventaja le sacará el programa lineal al cuadrático, ya que la regla general es que un algoritmo más simple corre más rápido ("para algún n suficientemente grande" = "asintóticamente"). Sin embargo, no hay que olvidarse que si la entrada siempre es pequeña, un algoritmo con peor complejidad asintótica y constante pequeña puede aventajar a un algoritmo con mejor complejidad y constante grande.

2. Cotas asintóticas

Para decir que un algoritmo f corre como mucho tan lento como g, escribimos:

$$f \in O(g) \overset{def}{\iff} \exists\, n_0, k > 0 : \forall n > n_0, f(n) \leq k \cdot g(n)$$

Y decimos: "f pertenece a O de g", "f es O de g" o "f crece a lo sumo tan rápido como g". Ésta es la única notación realmente importante, y que es necesaria entender al menos coloquialmente: si f es $O(g)$, f es igual o más eficiente que g.

Ejemplo: $T_1 \in O(n)$, $T_2 \in O(n^2)$, también resulta $T_1 \in O(n^2)$, aunque $T_2 \notin O(n)$.

Análogamente, para decir que un algoritmo f corre al menos tan lento como g, escribimos:

$$f \in \Omega(g) \overset{def}{\iff} \exists\, n_0, k > 0 : \forall n > n_0, f(n) \geq k \cdot g(n)$$

Y decimos "f es al menos tan compleja como g", y otras construcciones verbales...Notemos que si $f \in \Omega(g)$ entonces $g \in O(f)$.

Ejemplo: $T_1, T_2 \in \Omega(n)$, $T_2 \in \Omega(n^2)$ pero $T_1 \notin \Omega(n^2)$.

En particular, cuando un algoritmo f está acotado por arriba por g ($f \in O(g)$), pero también por abajo ($f \in \Omega(g)$), lo notamos así:

$$f \in \Theta(g) \overset{def}{\iff} f \in \Omega(g) \text{ y } f \in O(g)$$

Imaginemos un algoritmo, T_3, que es idéntico a T_1 para n impar, e idéntico a T_2 para n par. Resulta: $T_1 \in \Theta(n)$, $T_2 \in \Theta(n^2)$ pero $T_3 \notin \Theta(n)$, $T_3 \notin \Theta(n^2)$, ya que $T_3 \in O(n^2)$ y $T_3 \in \Omega(n)$.

Es importante notar la siguiente propiedad transitiva: $f \in O(g)$ y $g \in O(h) \implies f \in O(h)$. Esto nos servirá a la hora de simplificar complejidades. Ejemplo: $5n^2+9n+8 \in O(n^2)$, ya que...

$$\begin{aligned} 5n^2 + 9n + 8 &\leq 5n^2 + 9n^2 + 8n^2 \\ &= 22n^2 \\ &\in O(n^2) \end{aligned}$$

3. Calculando complejidades

3.1. Operaciones O(1)

Todas las operaciones que tardan un tiempo constante independientemente del tamaño de la entrada son $O(1)$. Esto incluye todas las operaciones aritméticas, lógicas y de bits; lectura y salida de variables sueltas (cabe recordar que los *strings* están compuestos de varios caracteres), y toma de decisiones (if), etc...

Ej: a++, if(p == !q), cout«42«endl.

3.2. Regla de la suma

Cuando un número fijo de operaciones se realicen en serie, la complejidad asintótica total será igual a la mayor de las complejidades de las operaciones, como se vio en el ejemplo del capítulo anterior.

El siguiente programa consiste de tres pasos: el primero y el tercero son $O(1)$, pero el for del medio depende linealmente de la entrada, n, entonces es $O(n)$. La regla de la suma nos dice que $O(1) + O(n) + O(1) = O(1 + n + 1) = O(n)$

```
a = 0;                          //O(1)
for(int i = 0; i < 5*n; ++i)    //O(n)
        a += i;
a = a * ( a+1 );                //O(1)
```

3.3. Regla del producto

Imaginemos ahora que modificamos el código así:

```
for(int i = 0; i < 5*n; ++i)            //O(n)
        for(int j = 0; j < m; ++j)      //O(m)
                a += i - j ;            //O(1)
```

El *for* interior tiene una complejidad de $O(m)$, y como se repite $O(n)$ veces, todo el programa es $O(nm)$. Es decir, la regla del producto nos dice $O(a) \cdot O(b) = O(ab)$. Esto implica que si tenemos k *fors* anidados, cada uno con complejidad $O(n)$, juntos tendrán una complejidad total de $O(n^k)$.

3.4. Regla de la potencia

Si se tiene una función recursiva f de k niveles, donde en cada llamada se recursa b veces —e ignorando las recursiones— cada llamada tiene una complejidad **fija** de $O(a)$, entonces $f \in O(a \cdot b^k)$.

```
void f(int k, int m){
        if( k == 0 ) return;    // al ultimo nivel cortar.
        f(k-1, m);
        f(k-1, m);          // b = 3 recursiones por llamada.
        f(k-1, m);
        for(int i = 0; i < m; ++i)  // O(m^2) por llamada.
                for(int j = 0; j < m; ++j)
                        cout<<i+j<<endl;
}
```

Por lo tanto, $f \in O(m^2 \cdot 3^k)$

Notemos que como m es fijo, $O(m^2)$ también lo es, ya que m no se modifica durante la ejecución de la función. Distinto sería si la complejidad por llamada fuera de $O(k^2)$, ya que k varía de llamada en llamada. A continuación un ejemplo:

3.5. Operaciones más complejas

Hay algunos algoritmos cuya cantidad de cómputos crece proporcionalmente al logaritmo de la entrada, o a su factorial, o a funciones aún muchísimo más complejas y difíciles de aproximar rápidamente utilizando las reglas arriba descriptas. Analicemos un ejemplo:

```
void f(int n){
        if( n == 0 ) return;
        f( n/2 );
        f( n/2 );
        for(int i = 0; i < n; ++i) cout<<i<<endl;
}
```

Tratando de contar exactamente la cantidad de cómputos terminaríamos en algo así:

$$n+2\left(\frac{n}{2}+2\left(\frac{n}{4}+2\left(\frac{n}{8}+\dots\right)\right)\right) = \langle\text{cambiando la variable } n \text{ por } 2^k \text{ y así simplificar el análisis}\rangle$$

$$2^k+2\left(\frac{2^k}{2}+2\left(\frac{2^k}{4}+2\left(\frac{2^k}{8}+\dots\right)\right)\right) = \underbrace{2^k+2\left(2^{k-1}+2\left(2^{k-2}+2\left(2^{k-3}+\dots\right)\right)\right)}_{k \text{ recursiones}} =$$

$$\underbrace{2^k+2^1\cdot 2^{k-1}+2^2\cdot 2^{k-2}+2^3\cdot 2^{k-3}+\dots}_{k \text{ términos}} = \underbrace{2^k+2^k+2^k+2^k+\dots}_{k \text{ términos}} =$$

$$k\cdot 2^k = \langle\text{deshaciendo el cambio de variable } 2^k \text{ por } n\rangle$$

$$log_2(n)\cdot n \in O(n\, log\, n)$$

Adicionalmente,

- Por lo general un algoritmo $O(n\, log\, n)$ es suficiente para cualquier problema.
- $\underbrace{1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\dots}_{n \text{ términos}} \in O(log\, n)$.
- $1+\frac{1}{k}+\frac{1}{k^2}+\frac{1}{k^3}+\frac{1}{k^4}+\dots = \frac{k}{k-1} \in O(1)$, para $k>1$.
- La cantidad de subconjuntos (y combinaciones) de un conjunto de n elementos es exactamente $2^n \in \Theta(2^n)$.
- La cantidad de maneras distintas de ordenar n elementos es exactamente $n! \in O(n^n)$.
- La manera más rápida de ordenar n elementos comparándolos, es $\Omega(n\, log\, n)$, alcanzado en el caso promedio por quicksort y siempre por mergesort. Aunque quicksort tiene constante computacional más pequeña (es más rápido) y usa $O(1)$ de memoria adicional, en lugar del $O(n)$ que utiliza mergesort.
- Todos los logaritmos son proporcionales entre sí, es decir, $log_a \in \Theta(log_b)$, para cualquier a, b positivos distintos de 1. Por esta razón casi nunca escribimos la base.
- $log(ab) = log(a) + log(b)$ y por consecuente $log(a^k) = k \cdot log(a)$.

4. Valores máximos

A continuación se presenta una tabla con los valores máximos estimados que puede alcanzar el tamaño de la entrada para cada tipo:

n	Peor algoritmo solucionador	Comentario
≤ 10	$O(n!)$, $O(n^6)$	ej.: Enumerar permutaciones.
≤ 16	$O(2^n \cdot n^2)$	ej.: Problema del viajero con DP.
≤ 20	$O(2^n \cdot n)$	ej.: DP con bitmask.
≤ 100	$O(n^4)$	ej.: 4 fors anidados.
≤ 400	$O(n^3)$	ej.: Floyd Warshall.
≤ 2000	$O(n^2 \cdot log\, n)$	ej.: 2 for anidados + estructura de datos de tipo árbol.
$\leq 10^4$	$O(n^2)$	ej.: Ordenamiento por burbujeo/selección/inserción.
$\leq 10^6$	$O(n \cdot log\, n)$	ej.: Quicksort/mergesort.
$\leq 10^8$	$O(n)$	Casi siempre, $n \leq 10^6$ por cuello de botella en la entrada/salida.
$\leq 2^{10^8}$	$O(log\, n)$, $O(1)$	La entrada está implícita, porque sino ocurriría lo de arriba.

Cuadro 2: Estimaciones muy aproximadas de n para 3s de ejecución.
Para estas estimaciones, se consideró que una computadora moderna, junto a un algoritmo medianamente optimizado, puede correr alrededor de 10^8 operaciones en 3s de ejecución.

5. Ejercicios

Calcular la complejidad:

- de la memoria necesaria para almacenar un grafo de V vértices y E aristas en sus representaciones de matriz de adyacencia y lista de adyacencia respectivamente.
- de la memoria necesaria para almacenar un árbol binario completo de n hojas.
- de la altura esperada de un árbol binario completo de n nodos.
- de insertar un elemento al final de un vector (push_back).
- de insertar un elemento al principio de un vector (push_front).
- máxima estimada que puede tener un algoritmo que deba procesar 10^5 datos en 3s.
- ```
  bool lower_bound(int A[], int n, int k){
          int a = -1,  b = n;
          while( b - a > 1 ){
                  int m = (a + b) / 2;
                  if( A[m] < k )  a = m;
                  else            b = m;
          }
          return b;
  }
  ```
- ```
  // S es una cadena de caracteres, n es su longitud
  int i;
  for( i = 0; i < n; ++i )
          while( S[i] == 'a' )
                  i++;
  ```

- ```
  for(int i = 1; i <= n; ++i)
          for(int j = 0; j < n; j += 5)
                  cout<<A[j]<<endl;
  ```

- ```
  for(int i = 1; i <= n; ++i)
          for(int j = 0; j < n; j += i)
                  cout<<A[j]<<endl;
  ```

- ```
  bool esPrimo(int n){
          for(int k = 2; k * k <= n; ++k)
                  if( !(n % k) ) return false;
          return true;
  }
  ```

- ```
  // la funcion se llama con k=n, es decir, f(n, n)
  void f(int n, int k){
          if( !k ) return;
          f(n, k/2);
          for(int i = 0; i < n; i += k)
                  cout<<i<<endl;
  }
  ```

- ```
  // AVISO: MUY DIFICIL
  // la funcion se llama con k=n, es decir, f(n, n)
  void f(int n, int k){
          if( !k ) return;
          f(n, k/2);
          f(n, k/2);
          for(int i = 0; i < n; i += k)
                  cout<<i<<endl;
  }
  ```
  ```