2

TIEMPO DE COMPLEJIDAD EN ALGORITMOS

Lic. Rodolfo Catunta

ENERO 2023

UNIVERSIDAD CATÓLICA BOLIVIANA

Contenidos I

1 Introducción

Introducción

¿Por qué es importante?

El cálculo o estimación del tiempo de ejecución y el análisis asintótico de estos, son importantes para evitar perder el tiempo realizando algoritmos lentos para un determinado problema y así evitar veredictos del tipo **TLE** (Time Limit Exceeded)

Tiempo de Ejecución o Tiempo de Complejidad

El **tiempo de ejecución** nos dirá la cantidad de tiempo **exacta** que un algoritmo tardará en ejecutarse, en cambio el **tiempo de complejidad** nos indica un **estimado** de cuanto tiempo tardará el algoritmo en ejecutarse. En la práctica basta con calcular el tiempo de complejidad.

Introducción

Tiempo de Ejecución (Cormen)

El tiempo de ejecución de un algoritmo para una entrada en particular es el *número de instrucciones y accesos a los datos ejecutados*. Calcular el tiempo de ejecución matemáticamente puede hacerse asignando un tiempo constante c_k a cada k linea de nuestro código y analizando cuantas veces se ejecuta dicha linea.

Tiempo de Complejidad (Laaksonen)

El tiempo de complejidad de un algoritmo estima cuanto tiempo usará nuestro algoritmo para una entrada en particular. La idea es representar la eficiencia como una función cuyo parámetro es el tamaño de la entrada (u algún otro dato de entrada). Si se calcula el tiempo de complejidad, nosotros podemos saber cuán rápido será un algoritmo sin implementarlo.

2

Notación Asintótica

Notación O

Definición

La notación O describe una cota superior asintótica. Más formalmente. Para una función dada $g(n)$, denotaremos $O(g(n))$ como el conjunto de funciones

$$O(g(n)) = \{f(n) : \text{ existen constantes positivas } c \text{ y } n_0 \text{ tales que}$$

$$0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$$

Uso Práctico

Esta notación nos indica que la función no crece más rápido que cierto factor, usualmente basado en el término de mayor orden. Por ejemplo. la función $6n^3 + 200n^2 - 20n + 10$ es una función $O(n^3)$ pero además es una función $O(n^c)$ para todo $c \geq 3$.

Notación Ω

Definición

La notación Ω describe una cota inferior asintótica. Más formalmente. Para una función dada $g(n)$, denotaremos $\Omega(g(n))$ como el conjunto de funciones

$$\Omega(g(n)) = \{f(n) : \text{ existen constantes positivas } c \text{ y } n_0 \text{ tales que}$$

$$0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}$$

Uso Práctico

Esta notación nos indica que la función crece al menos tan rápido que cierto factor, usualmente basado en el término de mayor orden. Por ejemplo. la función $6n^3 + 200n^2 - 20n + 10$ es una función $\Omega(n^3)$ pero además es una función $\Omega(n^c)$ para todo $c \leq 3$.

Notación Θ

Definición

La notación Θ describe una cota asintótica ajustada. Más formalmente. Para una función dada $g(n)$, denotaremos $\Theta(g(n))$ como el conjunto de funciones

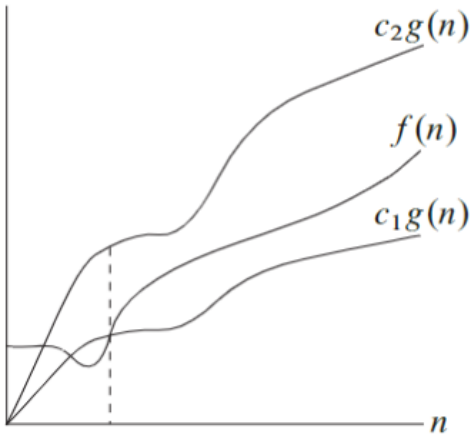
$$\Theta(g(n)) = \{f(n) : \text{ existen constantes positivas } c_1, c_2 \text{ y } n_0 \text{ tales que}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}$$

Uso Práctico

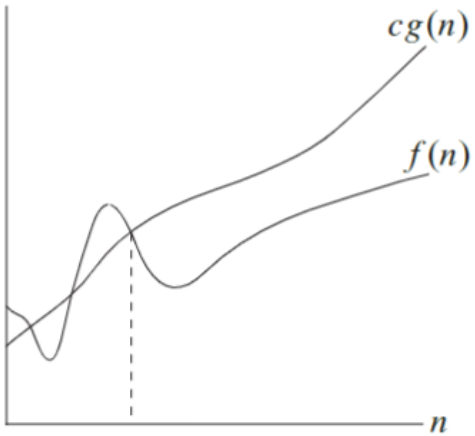
Esta notación nos indica que la función crece precisamente como cierto factor, usualmente basado en el término de mayor orden. Por ejemplo. la función $6n^3 + 200n^2 - 20n + 10$ es una función $\Theta(n^3)$. Nota que si puedes demostrar que una función es $O(f(n))$ y también $\Omega(f(n))$ entonces puedes afirmar que es $\Omega(f(n))$.

Ilustración



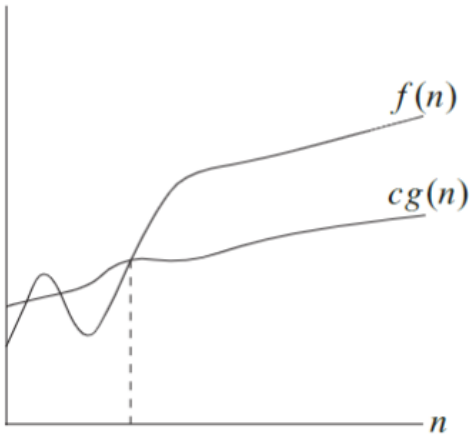
$f(n) = \Theta(g(n))$

(a)



$f(n) = O(g(n))$

(b)



$f(n) = \Omega(g(n))$

(c)

3

Reglas de Cálculo

Ciclos

Los ciclos **(while, for y do-while)** anidados generan que el tiempo de ejecución del algoritmo crezca. Por ejemplo si se tienen k ciclos anidados, cada uno realizando n vueltas, entonces el tiempo de complejidad del algoritmo es $O(n^k)$.
Por ejemplo el siguiente código tiene un tiempo de complejidad de $O(n)$.

```
1 for(int i=0;i<n;i++){
2     // ...
3 }
```

Y el siguiente códgio tiene un tiempo de complejidad de $O(n^2)$.

```
1 for(int i=0;i<n;i++){
2     for(int j=0;j<n;j++){
3         // ...
4     }
5 }
```

Orden de Magnitud

Recordemos que el tiempo de complejidad no mide el tiempo de ejecución exacto, solo nos muestra el **orden de magnitud**.
Por ejemplo en el siguiente código caca ciclo mostrado tiene una complejidad de $O(n)$.

```
1 for(int i=0;i<3*n;i++){
2     // ...
3 }
4
5 for(int j=0;j<n/2;j++){
6     // ...
7 }
```

Y el siguiente códgio tiene un tiempo de complejidad de $O(n^2)$.

```
1 for(int i=0;i<n;i++){
2     for(int j=i+1;j<n;j++){
3         // ...
4     }
5 }
```

Fases

Si un código posee varias fases, su tiempo de complejidad será igual al de la fase con tiempo de complejidad más largo.
Por ejemplo en el siguiente código hay 3 fases cada una de $O(n)$, $O(n^2)$ y $O(nlog(n))$,respectivamente. Así que el tiempo de complejidad total es de $O(n^2)$.

```
1  vector<int> v;
2  for(int i=0;i<n;i++){
3      v.push_back(i);
4  }
5
6  for(int i=0;i<n/2;i++){
7      for(int j=i+1;j<n;j+=5){
8      // ...
9      }
10 }
11
12 sort(v.begin(),v.end());
```

Varias Variables

Si un código posee varias variables y su tiempo de complejidad depende de ellas, entonces el tiempo de complejidad puede poseer varias variables.
Por ejemplo en el siguiente código hay dos ciclos con distinta cantidad de vueltas, asi la complejidad es $O(nm)$

```
1 for(int i=0;i<n;i++){
2     for(int j=0;j<m;j++){
3     // ...
4     }
5 }
```

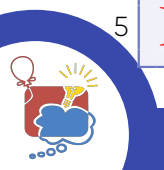
Recursiones

El tiempo de complejidad de una función recursiva, depende de la cantidad de llamadas que se haga a la función y el tiempo de complejidad del caso base.
Por ejemplo el siguiente código tiene un tiempo de complejidad $O(n)$

```
1 void f(int n){
2     if(n==1) return;
3     f(n-1);
4 }
```

Sin embargo el siguiente código tiene un tiempo de complejidad de $O(2^n)$.

```
1 void f(int n){
2     if(n==1) return;
3     f(n-1);
4     f(n-1);
5 }
```



4

Clases de Complejidad

Clases de Complejidad

Constante $O(1)$

Este tiempo de complejidad ocurre cuando la operación no depende del tamaño de la entrada. Por ejemplo, una fórmula matemática.

Logarítmica $O(\log n)$

Usualmente este tiempo se da cuando en cada paso se divide el tamaño de la entrada en 2 o más partes.

Raíz Cuadrada $O(\sqrt{n})$

Más lento que una complejidad $O(\log n)$ pero más rápido que una $O(n)$.

Clases de Complejidad

Constante $O(n)$

En la mayoría de problemas es el tiempo de complejidad a alcanzar pues casi siempre es necesario almacenar los datos en alguna estructura y acceder a ellos al menos una vez.

Linearitmica $O(n \log n)$

Se da cuando se debe ordenar los datos o cuando se usa n veces una estructura de datos en forma de árbol.

Cuadrática $O(n^2)$

Se da cuando se tienen dos ciclos anidados. Note que es posible pasar por todos los pares de elementos de un conjunto usando esta complejidad.

Clases de Complejidad

Cúbica $O(n^3)$

Se da cuando se tienen tres ciclos anidados. Note que es posible pasar por todas las tripletas de elementos de un conjunto usando esta complejidad.

$O(2^n)$

Se da cuando se itera sobre todos los subconjuntos de un conjunto de n elementos.

$O(n!)$

Se da cuando se itera sobre todas las permutaciones de un conunto de n elementos.

5

Estimación de la Eficiencia

Eficiencia Estimada

n	Peor Complejidad	Comentario
$\leq [10 \dots 11]$	$O(n!), O(n^6)$	e.g. Enumerar Permutaciones
$\leq [17 \dots 19]$	$O(2^n \cdot n^2)$	e.g. DP en TSP
$\leq [18 \dots 22]$	$O(2^n \cdot n)$	e.g. DP con técnica de Bitmask
$\leq [24 \dots 26]$	$O(2^n)$	e.g. Iterar sobre los subconjuntos
≤ 100	$O(n^4)$	e.g. DP con 4 dimensiones
≤ 450	$O(n^3)$	e.g. Floyd-Warshall
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort
$\leq 200K$	$O(n\sqrt{n})$	e.g. Ver si un número es primo n veces
$\leq 4.5M$	$O(n \log n)$	e.g. Merge Sort
$\leq 10M$	$O(n \log \log n)$	e.g. Criba de Eratostenes
$\leq 100M$	$O(n), O(\log n), O(1)$	Límite de la mayoría de problemas.

¡Gracias!