Complejidad Computacional

Lautaro Lasorsa

Curso OIA UNLaM 2021

Introducción

¿Qué tan rápido es un algoritmo?

Algo importante en la programación en general, y en la competitiva en particular, es tener una forma de medir qué tan rápido es un programa

Formas de hacerlo

La primer aproximación sería una empírica, es decir ejecutar el programa y medir el tiempo que tarda en ejecutarse con el input que nos interese.
Sin embargo, si queremos decidir qué algoritmo utilizar antes de hacer el programa, o bien una forma de estimar cuanto va a tardar antes de probarlo, vamos a necesitar una aproximación teórica a la cuestión.

Aproximación teorica

Las operaciones realizadas como función del input

Como la velocidad depende también del hardware que utilicemos, lo que vamos a medir es la cantidad de operaciones que realiza nuestro programa.
A su vez, estas operaciones van a depender no solo del programa en si sino del input, por eso la vamos a expresar como función del input.

Ignoramos las constantes

Como lo que nos interesa especialmente es ver el orden de magnitud de esta función, vamos a ignorar las constantes que se encuentren sumando o multiplicando.

Primer ejemplo: Operaciones atómicas

Veamos el siguiente código:

```
1 int n,m;
2 cin>>n>>m;
3 cout<<n*m<<endl;
```

Acá podemos ver 3 operaciones, primero se declaran n y m, luego se los lee por consola y por último se imprime su producto por pantalla.
Las 3 son operaciones atómicas, es decir decimos que cada una toma una operación, por tanto, la cantidad de operaciones es 3, una constante.
En este caso, diremos que su Complejidad Computacional es 1, y lo anotaremos $\mathcal{O}(1)$

Segundo ejemplo: Ciclos

Veamos el siguiente código:

```
1 int n,num,suma = 0;
2 cin>>n;
3 for(int i = 0;i<n;i++){
4     cin>>num;
5     suma+=num;
6 }
7 cout<<suma<<endl;
```

Lo interesante que incorporamos en este caso es que hay un ciclo for. Cuando tenemos ciclos, para calcular la complejidad total lo que debemos hacer es multiplicar la complejidad de lo que está adentro del ciclo por la cantidad de veces que se ejecuta el ciclo.
Por ejemplo, en este caso el cuerpo del ciclo es $\mathcal{O}(1)$, y se ejecuta n veces. Por tanto, su complejidad será $\mathcal{O}(n * 1) = \mathcal{O}(n)$

Tercer ejemplo: Ciclos anidados

Veamos el siguiente código:

```
1 int n,m;
2 cin>>n>>m;
3 for(int i = 1;i<=n;i++){
4     for(int j = 1;j<=m;j++){
5         cout<<i*j<<endl;
6     }
7 }
```

En este caso vemos el ejemplo de ciclos anidados. Si vemos el ciclo de afuera, se ejecuta $\mathcal{O}(n)$ veces. Por su parte, en su cuerpo está el ciclo de adentro, que se ejecuta $\mathcal{O}(m)$ veces, y su cuerpo tiene $\mathcal{O}(1)$ Por tanto, la complejidad total queda: $\mathcal{O}(n * m * 1) = \mathcal{O}(n * m)$

Usamos la función de complejidad

Cantidad de operaciones por segundo

Como el límite nos lo indican en tiempo y no en cantidad de operaciones, nos importa cuantas operaciones se realizan por segundo. Lo usual es que el juez entre 100 y 400 millones de operaciones por segundo. Notar que, aunque ignoramos los detalles, no todas las operaciones toman exactamente el mismo tiempo.

¿Cómo usamos la función?

La forma en la que usamos la función de complejidad que hemos definido previamente es tomar las cotas de las variables de nuestro problema, el tiempo límite y nuestra función de complejidad y ver si entra en tiempo.
Es decir, ver si: $\mathcal{O}(n_1, n_2, ...) \leq \mathit{TiempoLimite} * 400{,}000{,}000$

Cuarto ejemplo: Uso

Supongamos que para nuestro problema el tiempo límite son 2 segundos. Por tanto, podremos realizar hasta 800.000.000 de operaciones.
A su vez, las variables interesantes (que influyen en la complejidad) son n y m, con las siguientes cotas:
$1 \leq n \leq 100{,}000, 1 \leq m \leq 100{,}000$
Si nuestra complejidad resulta $\mathcal{O}(n+m)$ Entonces, para ver, el peor caso tomaremos $n = 100{,}000$, $m = 100{,}000$, resultando en que realiza:
$\mathcal{O}(100{,}000 + 100{,}000) \sim 200{,}000 \leq 800{,}000{,}000$ operaciones
Un código con está complejidad sería aceptado como solución del problema.
En cambio, si la complejidad fuera $\mathcal{O}(n * m)$ se realizarían
$\mathcal{O}(100{,}000 * 100{,}000) \sim 10{,}000{,}000{,}000 > 800{,}000{,}000$ operaciones
En este caso, el programa es demasiado lento y daría Límite de Tiempo Excedido como veredicto.

Quinto ejemplo: Funciones (1)

Veamos el siguiente código:

```
1  int SumaDivisores(n){
2      int suma = 0;
3      for(int i = 1;i<=n;i++){
4          if(n%i==0) suma += i;
5      }
6      return suma;
7  }
8  int main(){
9  int n; cin>>n;
10 for(int i = 1;i<=n;i++){
11 cout<<SumaDivisores(n)<<endl;
12 }}
```

(sigue en siguiente diapositiva)

Quinto ejemplo: Funciones (2)

Al igual que pasa con los ciclos, lo que debemos hacer cuando llamamos funciones es considerar la complejidad del cuerpo de la función. En este caso, la función *SumaDivisores*(i) es $\mathcal{O}(i)$, por tanto, la complejidad total sería $\mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(\frac{n*(n-1)}{2}) = \mathcal{O}(n^2)$

Complejidades útiles de saber (1)

Vector

Crear o copiar un vector V tiene complejidad $\mathcal{O}(|V|)$ Si el contenido del vector son a su vez otras estructuras, se deberá sumar el coste asociado. Por ejemplo, crear una matriz de $n \times m$ tiene complejidad $\mathcal{O}(n * m)$

Función Sort

```
1 sort(v.begin(),v.end());
```

Este código tiene complejidad $\mathcal{O}(|V| * log(|V|))$

Función Reverse

```
1 reverse(v.begin(),v.end());
```

Este código tiene complejidad $\mathcal{O}(|V|)$

Complejidades útiles de saber (2)

Copiar estructura

En general, y salvo que se indique lo contrario, copiar el contenido de una estructura tiene una complejidad igual al tamaño de la estructura (considerando el tamaño de las cosas que contiene)

Cola

```
1 Q.push(x); Q.front(); Q.pop();
```

Cada una de las operaciones listadas arriba tiene $\mathcal{O}(1)$

Cola de prioridad

```
1 PQ.push(x); PQ.top(); PQ.pop();
```

Cada una de las operaciones listadas arriba tiene $\mathcal{O}(log(|PQ|))$ ($|PQ|$ es el tamaño de PQ en el momento de realizar la operación)

Complejidades útiles de saber (3)

Set

```
S.insert(x); S.erase(x); S.find(x)
```

Cada una de las operaciones listadas arriba tiene $\mathcal{O}(log(|S|))$

Mapa

```
M[x] = a; b = M[x]; M.find(x);
```

Cada una de las operaciones listadas arriba tiene $\mathcal{O}(log(|M|))$

Complejidades útiles de saber (4)

Iteración

```
1 for(auto & x : S){O(A)}
2 for(auto & x : M){O(A)}
```

En ambos casos (iterando un *set* o un *map*) la complejidad será $\mathcal{O}(|S| * A)$ o $\mathcal{O}(|M| * A)$
Notar que, en este caso, como usamos el operador &, no copiamos todo el contenido de las estructuras contenidas en el *set* o *map* y por tanto no debemos considerar su tamaño en la complejidad computacional.
Si no usasemos el operador &, deberiamos contar aparte el costo de copiar todo el contenido de la estructura 1 vez.

Ejercicios

Problema 1

¿Cuál es la complejidad de la siguiente función? (n>=0)

```
1 int Fun(int n){
2     if(n==0)  return 1;
3     return Fun(n-1) + Fun(n-1);
4 }
```

Problema 2

Considerando la función definida antes, ¿Cuál es la complejidad del siguiente código?

```
1 for(int i = 1;i<=n;i++){
2     cout<<i<<" -> "<<Fun(i)<<endl;
3 }
```