Complejidades Asintóticas

Redigonda Maximiliano

Complejidades Asintóticas

Para que el **juez** acepte (AC) nuestro código para un problema, no sólo necesitaremos correctitud, sino también **eficiencia**.

Para que nuestro programa sea eficiente, debe utilizar sus recursos (tiempo, memoria), de una manera razonable.

Los límites de tiempo y memoria están dados para todos los problemas de la competencia.

Complejidades Asintóticas

Nos gustaría responder preguntas como:

- Cuánto **tiempo** tardará mi programa en el peor caso?
- Cuánta **memoria** utilizará mi programa en el peor caso?

Tenemos dos opciones:

1. Contar cantidad de operaciones realizadas.
2. Estimar de una forma matemáticamente aceptable.

#1 - Contar operaciones

Para problemas extremadamente sencillos, puede funcionar.

Pero:

- No todas las operaciones cuestan lo mismo!
- Si el programa es ligeramente largo puede ser tedioso!
- No se requiere tanta exactitud!
- Nunca sabemos de verdad lo que nuestro compilador hace con nuestro código.

Por todas estas, la #1 no es la recomendable.

#2 - Estimar

Ya que no necesitamos mucha precisión, estimar parece ser una buena idea.

Medimos el coste de un programa como una función del tamaño de la entrada, por ejemplo, $T(n) = n^2 + n \log_2(n)$.

Generalmente, la idea es ignorar constantes y términos menos significativos.

La notación “o grande”, tiene una definición formal.

Ejemplos

```
for(int i = 0; i < n; ++i){
    printf(“Hola mundo!\n”);
}
```

Ejemplos

```
for(int i = 0; i < 5*n; ++i){
    printf(“Hola mundo!\n”);
}
```

Ejemplos

```
for(int i = 0; i < n; i += 5){
    printf(“Hola mundo!\n”);
}
```

Ejemplos

```
for(int i = 0; i < n; ++i){
    for(int j = 0; j < m; ++j){
        printf(“Hola mundo!\n”);
    }
}
```

Ejemplos

```
for(int i = 0; i*i < n; ++i){
    printf(“Hola mundo!\n”);
}
```

Ejemplos

```
for(int i = 1; i < n; i *= 2){
    printf(“Hola mundo!\n”);
}
```

Ejemplos

```
for(int i = 1; i < n; ++i){
    for(int j = 0; j < n; j += i){
        printf(“Hola mundo!\n”);
    }
}
```

Ejemplos

```
for(int i = 1; i < n; i *= 2){
    for(int j = 0; j < n; j += i){
        printf(“Hola mundo!\n”);
    }
}
```

Ejemplos

```
void f(int n){
    if(n <= 1) return;
    f(n - 1);
    f(n - 1);
}
```

Complejidad Amortizada

Complejidad amortizada es el costo total por operación, evaluado sobre una secuencia de operaciones.

En un vector, cuando la función push_back excede la capacidad reservada, duplica el espacio reservado (copiando los valores a una nueva posición de memoria).

Un push_back puede tomar O(n) en ejecutarse, sin embargo, como el espacio reservado se duplica, las siguientes n-1 llamadas cuestan O(1). Es decir, n llamadas, O(n), lo cual es O(1) amortizado.

Complejidad Amortizada

Se suele estimar que 1 segundo = c * 10^8 operaciones “sencillas”.

Hay muchas cosas extra que tenemos que tener en cuenta (estamos utilizando memoria dinámica? Nuestro algoritmo es cache-friendly? Es fácilmente optimizable?).

Algunos recursos suelen mostrar una tabla como la que veremos a continuación.

Complejidad Amortizada

Complejidad	Máximo N
O(1)	Infinito
O(log n)	$2^{100000000}$
O(n)	100000000 (10^8)
O(n log n)	10000000 (10^7)
O(n^2)	10000 (10^4)
O(n^3)	700
O(n^4)	150
O($n^2\,2^n$)	20
O(n!)	12