

Cálculo de complejidades

La notación asintótica nos da una idea de cuántos recursos usa nuestro algoritmo, además, no es dependiente de los componentes que use la computadora en la que corre nuestro algoritmo.

Funciones de operaciones

Para entenderla, primero introducimos la función $T(n)$. Esta función toma como entrada el tamaño de la entrada y retorna cuantas operaciones realizara.

Por ejemplo:

Si tuviéramos el siguiente código

Código 1.

```
int n, s = 0; cin >> n;
for (int i = 0; i < n; i++){
    s++;
}
```

Supongamos que las asignaciones, lectura, sumas y comparaciones valen como una operación. Entonces si $n = 3$, podemos “desglosar” el for de la siguiente manera

Código 2.

```
int n, s = 0; cin >> n;
int i = 0;
(i < n); s++; i++; //i = 1
(i < n); s++; i++; //i = 2
(i < n); s++; i++; //i = 3
(i < n);           // 3 no es menor que 3
```

Si contamos el número de operaciones tenemos 14 operaciones, entonces, de manera más general cuantas operaciones haría un for?

El for que se muestra en el código 1 tiene una función de operaciones $T(n) = 2n + 2$

1 asignación

```
int i = 0;
```

$n + 1$ comparaciones

```
(i < n);
```

n sumas

```
i++;
```

Fuera del for tenemos 3 operaciones

```
int n, s = 0; cin >> n;
```

Y finalmente el for ejecuta una suma n veces.

```
s++;
```

Por lo tanto la función de operaciones del código 1 es

$$\begin{aligned}T(n) &= 1 + n + 1 + n + 3 + n \\T(n) &= 3n + 5\end{aligned}$$

Si $n = 3$:

$$T(3) = 3(3) + 5 = 14 \text{ operaciones}$$

Ahora bien, por qué no es conveniente usar/calcular la función $T(n)$?

- Es complicada de calcular
- No es una muy buena forma de comparar algoritmos

Por ejemplo: supongamos que tenemos dos algoritmos de ordenamiento y sus respectivas funciones $T(n)$.

$$\begin{aligned}T_A(n) &= 3n^2 + 2n + 10 \\T_B(n) &= 50n\log_2 n + 4n + 4\end{aligned}$$

¿Cuál es más rápido?

A primera vista parecería que A es más rápido, pues si lo probamos con 4 datos tenemos

$$T_A(4) = 3 * 4^2 + 2 * 4 + 10 = 66 \text{ operaciones}$$

En cambio B con 4 datos

$$T_B(4) = 50 * 4 * \log_2 4 + 4 * 4 + 4 = 420 \text{ operaciones}$$

Pero, mientras n sea mas grande, el algoritmo B sera mas rapido que el algoritmo A, siempre existirá un punto en el que B se más eficiente que A.

Aún así, supongamos que tenemos una supercomputadora que ejecutara el algoritmo A y hace 10^{10} operaciones por segundo, en cambio, tenemos una computadora normal que ejecutara B y hace 10^7 operaciones por segundo.

A se ejecutará en una computadora 1000 veces más rápido que la computadora que ejecuta B. Si tenemos que ordenar 10^7 valores, cuál algoritmo es más rápido?

$$\begin{aligned}T_A(10^7) &= 3(10^7)^2 + 2(10^7) + 10 \approx 3 * 10^{14} \text{ operaciones} \\T_B(n) &= 50(10^7)\log_2(10^7) + 4(10^7) + 4 \approx 11.66 * 10^9 \text{ operaciones}\end{aligned}$$

Calculamos el tiempo

$$\begin{aligned}t_A &= \frac{3 * 10^{14} \text{ op}}{10^{10} \text{ op/s}} = 30000 \text{ s} = 8.33 \text{ horas} \\t_B &= \frac{11.66 * 10^9 \text{ op}}{10^7 \text{ op/s}} = 1166 \text{ s} = 19.43 \text{ mins}\end{aligned}$$

Ahora quedó muy claro que el algoritmo B es mucho más rápido que A.

Por limitaciones del hardware no nos conviene pensar en número de operaciones, si no, en el crecimiento de la función de operaciones.

Funciones de crecimiento

La notación asintótica nos da una idea más general del crecimiento de una función, es más fácil de calcular, nos permite comparar algoritmos (con algunas desventajas), nos da una idea del tiempo de ejecución y para nuestro fin, nos permite (saber antes de programar) si nuestro algoritmo pasará en tiempo y memoria.

- Notación O-grande (cota superior)

Decimos que una función $f(n) = O(g(n))$ si existe una constante c tal que:

$$f(n) \leq cg(n)$$

Es decir que $f(n)$ nunca pasara $c g(n)$, $g(n)$ es como un límite que nos indica que nuestro algoritmo nunca hará más de $c g(n)$ operaciones.

- Notación Ω-grande (cota inferior)

Decimos que una función $f(n) = \Omega(g(n))$ si existe una constante c tal que:

$$f(n) \geq cg(n)$$

Es decir que $f(n)$ siempre estará por encima de $c g(n)$, $g(n)$ es como un límite que nos indica que nuestro algoritmo siempre hará más de $c g(n)$ operaciones.

- Notación Θ-grande (caso medio)

Decimos que una función $f(n) = \Theta(g(n))$ si se cumple

$$f(n) = O(g(n)) \text{ y } f(n) = \Omega(g(n))$$

Este se considera un caso promedio de nuestro algoritmo.

Reglas para medir la complejidad

Estas reglas solo nos servirán para calcular la complejidad O-grande de nuestro algoritmo.

1. Regla de la suma

Tiene sentido que si ejecutamos dos algoritmos diferentes, uno después del otro, se sumen sus funciones de operaciones, es decir:

$$T(n) = T_1(n) + T_2(n)$$

Al calcular la complejidad $O(T(n))$ solo nos interesa el término más significativo, es decir:

$$T(n) = O(\max(T_1(n), T_2(n)))$$

Por ejemplo:

$$T(n) = 3n^2 + 5n + 3$$

$$T(n) = O(3n^2)$$

Entonces, cuál es la complejidad de $T(n) = n \log_2 n + n! + 30n$?

$$T(n) = O(n!)$$

Para determinar el término dominante o mayor se puede usar la siguiente lista:

$$n! > 2^n > n^3 > n^2 > n \log_2 n > n > \sqrt{n} > \log_2 n > cte$$

Todo esto es válido al comparar el crecimiento de las funciones, se recomienda usar un graficador para comprobar que esto es verdad.

2. Producto por constante

Esta regla dice que se omiten las constantes, es decir:

$$O(cf(n)) = O(f(n))$$

Siguiendo con el ejemplo anterior tendríamos la complejidad:

$$T(n) = O(3n^2) = O(n^2)$$

Además, la complejidad de una constante es $O(1)$, es decir que nuestro algoritmo siempre hará el mismo número de operaciones sin importar el tamaño de la entrada.

Por ejemplo, tenemos un algoritmo que hace un cálculo como encontrar la hipotenusa de un triángulo rectángulo.

3. Regla del producto

Las multiplicaciones usualmente se dan en los ciclos, porque se hace los que está dentro del for n veces.

Por ejemplo si tuviéramos el siguiente fragmento de código:

Código 3.

```
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        a++;
    }
}
```

La complejidad del ciclo de la variable j es de $O(n * 1)$, pues, se ejecuta una suma n veces.

La complejidad del ciclo de la variable i es de $O(n^2)$, porque ejecuta el for del j n veces.

Podemos concluir que si tenemos k ciclos anidados su complejidad es de $O(n^k)$.

4. Condicionales

Dado que queremos una cota superior, tiene sentido que seamos pesimistas al medir la complejidad, es por eso que, si nos encontramos un if tomamos la rama de peor complejidad.

5. Funciones recursivas

Por lo general, la complejidad de una función recursiva tiende a ir por dos caminos:

- Fuerza bruta: exponencial $O(c^n)$.
- Divide y vencerás (D&C): logarítmica $O(\log_2 n)$ o lineal logarítmica $O(n \log_2 n)$.

Por ejemplo, tenemos la secuencia de fibonacci que se define de la siguiente forma:

$$f_n = f_{n-1} + f_{n-2}; \quad f_0 = 0 \text{ y } f_1 = 1$$

Sus primeros términos son: {0, 1, 1, 2, 3, 5, 8, 13, 21, }.

Podemos hacer la siguiente función recursiva para calcular el n-ésimo término:

Código 4.

```
int fib(int n){
    //casos base
    if(n == 1) return 1;
```

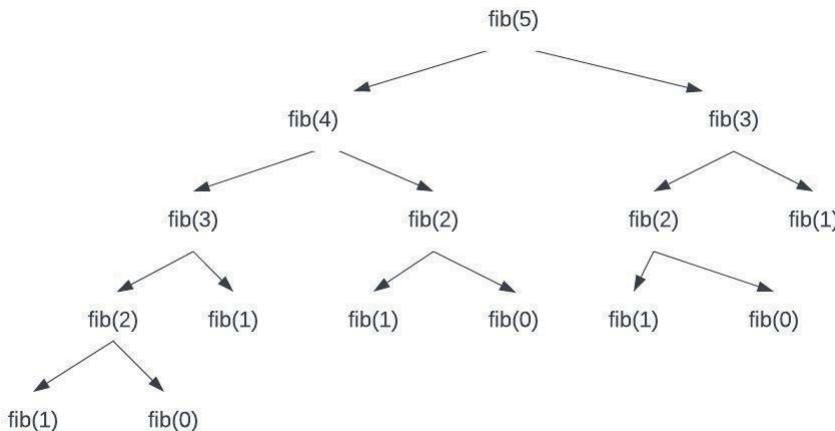
```

if(n == 0) return 0;

return fib(n - 1) + fib(n - 2);
}

```

Esta función genera el siguiente árbol de recursión si $n = 5$:



Para calcular la complejidad, se debe contar cuantas veces la función se llama a sí misma y se lo eleva a n , así, la complejidad de la función $\text{fib}(n)$ es $O(2^n)$.

6. Múltiples variables

Se puede tener el caso en el que la complejidad de nuestro programa dependa en dos o más variables, en ese caso es recomendable expresar la complejidad en función de las variables. Por ejemplo:

- El recorrido de un grafo con V vértices y E aristas tiene una complejidad $O(V + E)$, esta sigue siendo una complejidad lineal.
- Introducir una matriz de n filas y m columnas tiene una complejidad de $O(nm)$, esta sigue siendo una complejidad cuadrática.

Complejidades comunes y estimación de eficiencia

A continuación se muestra una tabla con algunas de las complejidades más comunes, con el tamaño de entrada máximo que soportan, además, se da un ejemplo de un algoritmo con esta complejidad.

n	Peor complejidad	Ejemplo
$n \leq 11$	$O(n!)$	enumerar permutaciones
$n \leq 18$	$O(2^n n^2)$	TSP con DP y máscaras de bits

$n \leq 22$	$O(\text{for } (2^n))$	Enumerar subconjuntos
$n \leq 400$	$O(n^3)$	Floyd-Warshall
$n \leq 2000$	$O(n^2 \log n)$	2 Loops anidados + Búsqueda binaria
$n \leq 10^4$	$O(n^2)$	Recorrer una matriz cuadrada
$n \leq 10^6$	$O(n \log n)$	Construir una árbol de segmentos
$n \leq 10^8$	$O(n), O(\log n), O(1)$	Recorrer un arreglo
$n \leq 10^{18}$	$O(\log n), O(1)$	Búsqueda binaria o fórmula

Esta información es con el supuesto de que la computadora haga 10^8 op/s.

Entonces, por ejemplo, si un problema nos dice calcular el n -ésimo término de fibonacci.

Si $n \leq 22$; la recursión definida previamente bastará como cualquier algoritmo de menor complejidad

Si $n \leq 10^6$; la recursión acabara de correr en un tiempo mayor al tiempo de vida del universo, pero podemos usar un algoritmo de complejidad $O(n \log_2 n)$ o menor. Usualmente la solución iterativa en $O(n)$.

Si $n \leq 10^{18}$; el algoritmo lineal ya no funcionaría y tendríamos que usar $O(\log_2 n)$ o menor.

Esto se puede hacer con recurrencias lineales + exponenciación de matrices, aunque también existe una fórmula con la cual tendríamos una complejidad constante sacrificando precisión de cálculo por ser números reales.

Con solo ver los límites del problema, podemos tener una idea de que clase de algoritmo buscar y cuales evitar para resolver el problema.

Ejercicios

1. Medir la complejidad de las siguientes funciones de tiempo

- a) $T(n) = 3n^2 + 5n + 3$
- b) $T(n) = 7n \log_2 n + 45$
- c) $T(n) = 2^n n$
- d) $T(n) = 3n! + 40 \log_2 n$