Clase 4: Análisis Asintótico

Eficiencia

La eficiencia de un algoritmo describe el ratio de crecimiento del consumo de recursos como tiempo de CPU o memoria, respecto el tamaño de entrada del mismo.

Benchmarks

Una forma de medir la eficiencia de un algoritmo es realizar un benchmark, que vendría a ser realizar una prueba de rendimiento para medir de manera experimental la cantidad de recursos que un programa utiliza mediante su ejecución con un conjunto de casos de prueba.

Realizar benchmarks tiene una desventaja cuando lo que se quiere es diseñar o analizar algoritmos y es que los resultados serán distintos dependiendo de la máquina o dispositivo en que se ejecute el benchmark. No obstante son muy útiles, estos se suelen utilizar en otras etapas del desarrollo de software, ya sea para realizar comparaciones de implementaciones específicas, o para diagnosticar cuellos de botella junto a otras técnicas como pruebas de carga o profiling.

Tiempo de ejecución

El análisis de algoritmos tiene por objetivo encontrar una expresión matemática que describa el ratio de crecimiento del uso de recursos computacionales en función del tamaño de la entrada. Este enfoque permite caracterizar la eficiencia de un algoritmo, de forma independiente a una implementación o el hardware en que el algoritmo pueda ser ejecutado.

Cuando hablamos de tiempo de ejecución nos referimos a la cantidad de instrucciones que tiene el algoritmo, puesto que si tiene X instrucciones, cada una de estas tendrá que ser ejecutada por la CPU en algún momento. Para entender esto partamos con un algoritmo bien conocido: búsqueda lineal.

```java
public static int buscar(int[] arreglo, int elemento) {
    for (int i = 0; i < arreglo.length; i++) {
        if (arreglo[i] == elemento) {
            return i;
        }
    }
    return -1;
}
```

Listing 1: Búsqueda Lineal en Java

Podemos analizar el tiempo de ejecución del algoritmo linea a linea:

```
1. El ciclo itera en el peor de los casos en todo el arreglo,
desde la primera posición 0, hasta la última N-1. Así que diremos
que esta primera linea en peor caso hará N instrucciones, porque
depende del largo del arreglo.
    a. Es un statement if que evalua una condición que compara si dos
    enteros son iguales. Diremos que tiene un costo constante y
    puesto que es una constante diremos que es 1.
    b. Es un statement return, dicho statement retorna una expresión
    constante, por lo que diremos que es 1.
2. El ciclo ya ha terminado y retorna una expresión constante
(-1) cuando el algoritmo no encuentra el elemento en el arreglo.
```

Si bien esto se ve bastante verboso, en la práctica lo que importa es la cantidad de instrucciones asociadas a cada linea:

```
1. N
    a. 1
        i. 1
2. 1
```

Dado esto podemos aseverar lo siguiente:

1. El peor caso es cuando el arreglo no tiene el elemento que el algoritmo va a buscar, puesto que itera en todo el arreglo de inicio a fin.

2. Puesto que itera de inicio a fin, nunca entra dentro del statement if para realizar el return de la linea 4.

Por lo que podemos expresar el tiempo de ejecución del algoritmo con la siguiente expresión:

$$T(N) = N * 1 + 1 = N + 1$$

Por lo tanto, podemos concluir que búsqueda lineal tiene un comportamiento lineal en su tiempo de ejecución en función del tamaño de entrada.

Calcular el tiempo de ejecución de un algoritmo calculando el total de instrucciones es un buen ejercicio y requiere práctica, de ahora en adelante cada algoritmo que se diseñe o implemente, tendrá que ser analizado.

Big O

Tener el número "exacto" de instrucciones que un algoritmo realiza es innecesario para analizar algoritmos de forma objetiva. Principalmente porque en la práctica la cantidad de instrucciones depende del hardware y la arquitectura de la CPU. Es por esto que extenderemos la idea anterior, con otra idea: Big O, una forma de realizar análisis asintótico.

El análisis asintótico es un método para describir el comportamiento límite de una función cuando el argumento tiende a cierto valor o al infinito. La notación Big O permite describir el tiempo de ejecución o uso de memoria de un algoritmo mediante una cota superior.

Para una función $g(n)$ denotamos $\mathcal{O}(g(n))$, que se pronuncia Big O de la función g(n). Esta notación describe un conjunto de funciones:

$$\mathcal{O}(g(n)) = \{f(n) : \text{existen las constantes c y } n_0\text{, tales que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$$

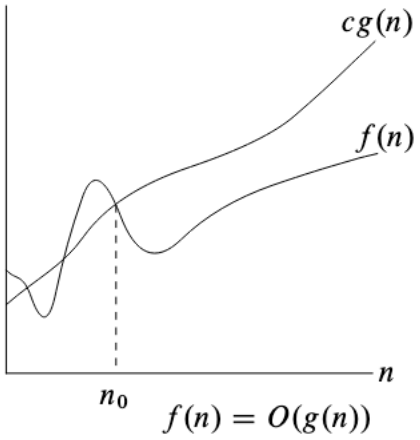

Figure 1: Big O describe un conjunto de funciones que se comportan como cota superior de una función en particular.

En la práctica Big O, lo utilizamos para describir el peor caso de un algoritmo. La cota superior que Big O nos da, nos dice que el tiempo de ejecución del algoritmo en cuestión no crecerá más rápido que $\mathcal{O}(f(n))$

Si volvemos al resultado que obtuvimos de búsqueda lineal:

$$T(N) = N + 1 = \mathcal{O}(n)$$

Lo que nos dice lo que queríamos saber desde un principio: búsqueda lineal tiene un tiempo de ejecución que se comporta de manera lineal en función del tamaño de entrada.

El método para encontrar el Big O de una expresión $T(N)$, es descartar las constantes aditivas y multiplicativas, para quedarnos con las variables de mayor grado.

Vamos con un par de ejemplos más:

1. $T(N) = 5N + 20 = \mathcal{O}(N)$
2. $T(N) = 20N^2 + 5N + 20 = \mathcal{O}(N^2)$
3. $T(N) = 2^N + N^{20} = \mathcal{O}(2^N)$
4. $T(N) = 3N + \log N = \mathcal{O}(N)$
5. $T(N) = 25 \log N + 27\sqrt{N} + 20142 = \mathcal{O}(\sqrt{N})$
6. $T(N) = 3^N + N! = \mathcal{O}(N!)$
7. $T(N) = 100 = \mathcal{O}(1)$
8. $T(N, M) = 5N^2 + 8M + 20N = \mathcal{O}(N^2 + M)$

Análisis de transformación de decimal a binario

Analicemos entonces el algoritmo que estudiamos la clase anterior para transformar un número entero decimal N en su representación en sistema binario:

```
1   public static String decimalToBinary(int N) {
2       String binario = String.valueOf(N % 2);
3
4       while (N / 2 > 0) {
5           N = N / 2;
6           binario = (N % 2) + binario;
7       }
8
9       return binario;
10  }
```

Listing 2: Decimal a binario

Partamos calculando el tiempo de ejecución.

```
1. 1
2. Log(N)
    a. 1
    b. 1 (asumiremos que es constante de momento,
        en la próxima clase estudiaremos este caso en detalle)
3. 1
```

Por lo tanto el tiempo de ejecución en el peor caso es:

$$T(N) = 2 \log N + 1$$

Si aplicamos la notación Big O a la expresión anterior obtenemos que $T(N) = \log N$, es decir que el algoritmo tiene un comportamiento logarítmico!

Notación Omega

La notación Omega describe una cota ajustada de forma superior e inferior, se suele utilizar para describir la cota inferior de un algoritmo, es decir el mejor caso.

$$\Omega(g(n)) = \{f(n) : \text{existen las constantes } c \text{ y } n_0, \text{ tales que } 0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}$$

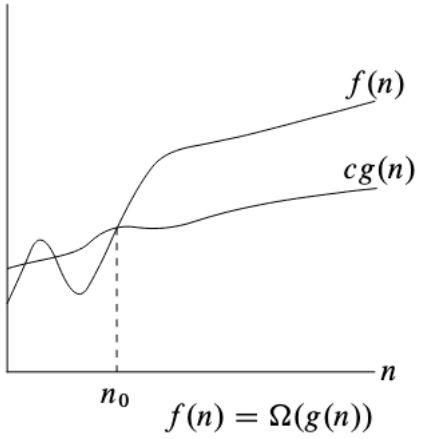

Figure 2: Omega describe un conjunto de funciones que se comportan como cota inferior de una función en particular.

Notación Theta

La notación Theta describe una cota ajustada de forma superior e inferior, se suele utilizar para describir que un algoritmo tiene cota superior e inferior iguales.

$$\Theta(g(n)) = \{f(n) : \text{existen las constantes } c_1,\ c_2 \text{ y } n_0, \text{ tales que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}$$

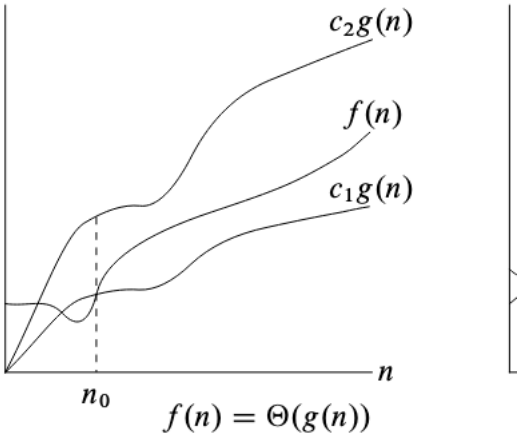

Figure 3: Theta describe un conjunto de funciones que se comportan como cota inferior y superior de una función en particular.

Análisis Amortizado

El análisis amortizado lo utilizaremos en situaciones donde tenga más sentido analizar el costo promedio de todas las operaciones de un algoritmo, en particular en situaciones donde algunas operaciones son costosas pero poco frecuentes.

Funciones comunes

Durante el transcurso del curso vamos a ver que hay funciones que son bien comunes:

1. Función constante: $f(n) = 1$
2. Función logaritmo: $f(n) = \log n$
3. Función raíz cuadrada: $f(n) = \sqrt{n}$
4. Función lineal: $f(n) = n$
5. Función linearítmica: $f(n) = n \log n$
6. Función cuadrática: $f(n) = n^2$
7. Función cúbica: $f(n) = n^3$
8. Función exponencial: $f(n) = 2^n$
9. Función factorial: $f(n) = n!$