Complejidad Algorítmica

Por: Sebastián Sánchez.

Cuando hablamos de la complejidad de un algoritmo, hablamos de la cantidad de recursos que el algoritmo utiliza. Estos recurso pueden ser espaciales o temporales, esto nos indica que debemos analizar la **complejidad espacial** y la **complejidad temporal** de un algoritmo antes de implementarlo.

La **complejidad espacial** se refiere a cuánta memoria necesita un algoritmo para ejecutarse completamente. En la actualidad, este análisis no es “demasiado” importante en una aplicación de escritorio, ya que las computadoras poseen una gran cantidad de memoria; pero en una aplicación web con miles de clientes concurrentes o una aplicación móvil, es importante analizar la complejidad espacial.

Complejidad Temporal

La complejidad temporal es un análisis prácticamente teórico, nos dice cuanto se demora el algoritmo en terminar. Para realizar este análisis debemos encontrar una función de tiempo de ejecución, utlizando una medida adimensional de tiempo mínimo **t** que nos dice el tiempo que tarda el procesador en hacer una operación sencilla, como una asignación, una declaración o una expresión simple. El resultado es típicamente expresado en notación O grande. Esto suele ser útil para comparar algoritmos, especialmente cuando se necesita procesar una gran cantidad de datos, hablaremos de esta notación más adelante.

Como ejemplo:

```
int funcion(int n)
{
     int i = n % 2;      //T(declaracion) + T(asignación) = 2t
     return n*i;         //T(retorno) = 1t
}                        //T(funcion) = 3t
```

La funcíon de tiempo de ejecución para el algoritmo “funcion” es: `T(n) = 3`
Esta función es constante, ya que sin importar los recursos de entrada el tiempo de ejecución siempre será el mismo.

Ahora, ¿cómo podemos aplicar esto en las estructuras de datos? Como ejemplo podemos analizar el algoritmo de inserción en una cola o lista enlazada, agregando el ultimo elemento insertado al final de todos los elementos existentes.

```
void insertar(T elemento)
{
     Nodo *nuevo = new Nodo(elemento); //2t
     //T(if) = T(condicion) + el peor de los casos
     if (cabeza == nullptr)
     {
          cabeza = nuevo; //1t
     }
     else  //Este es el peor de los casos
     {
          Nodo *aux = cabeza; //2t
          //Si la lista tiene n elementos
          //Esto se repetiría n - 1 veces
          while (aux->siguiente  != nullptr)
          {
               aux = aux->siguiente; //2t
          }
          //T(while) = (n - 1) * 2t
          aux->siguiente = nuevo; //2t
     }
}
//T(insertar) = 2t +  1t + max(1t, 2t + (n – 1) * 2t + 2t)
//T(insertar) = 3t + max(1t, 2t + 2tn)
//T(insertar) = 3t + 2t + 2tn = 2n + 5;
```

La función de tiempo de ejecución de “insertar” es `T(n) = 2n + 5.`
Esto nos dice cuanto tiempo el algoritmo tardará en ser ejecutado tomando en cuenta la cantidad de elementos que poseé la lista. La anterior situación nos sirve para analizar las propiedades de esta estructura y decidir si implementarla o no.

Ahora realizaremos un ejemplo y veremos como las matemáticas discretas nos ayudarán a encontrar ciertas funciones de tiempo de ejecución.

Dado el algoritmo recursivo:

```
int recursivo(int n)
{
     if ( n <= 1)
     {
          return 1;      //Para n <= 1
                         //T(condicion) + T(retorno);
     }
     else
     {
          return recursivo(n-1) + recursivo(n-1);
          //Para n > 1,  T(return) ...
          //... + T(recursivo(n-1)) + T(recursivo(n-1))
     }
}
```

Analizando el algoritmo anterior, si la entrada es 0 o 1, la función es $T(n) = 2$, pero si $n > 1$ entonces la función sería $T(n) = 1 + T(n-1) + T(n-1)$, una relación de recurrencia.
Para $n <= 1$ $T(n) = 2$
Para $n > 1$ $T(n) = 1 + 2*T(n-1)$

Estableciendo la relación de recurrencia $a_n = T(n)$, $a_{n-1} = T(n-1)$.

Resolvemos $a_n = 1 + 2a_{n-1}$

$a_n - 2a_{n-1} = 1$

a_n = solución homogénea + solución particular
$a_n^{(h)}$:
$a_n - 2a_{n-1} = 0$
$r - 2 = 0$
$r = 2$ entonces $a_n^{(h)} = c2^n$
donde c es una constante.
$a_n^{(p)}$:
asumimos $a_n^{(p)}$ como una constante A
sustituímos $a_n^{(p)}$ en la ecuación
$A - 2A = 1$
$-A = 1$
$A = -1$ entonces $a_n^{(p)} = -1$

$a_n = c2^n - 1$.

Aplicando la condición inicial $T(1) = 2$
$a_1 = 2$
$2 = c2^1-1$
$2 = 2c - 1$ entonces $c = 3/2$

$a_n = 3/2 * 2^n - 1$

Entonces : Para $n <= 1$ $T(n) = 2$;
Para $n > 1$ $T(n) = 3/2 * 2^n - 1$

Analizando otra versión del mismo algoritmo:

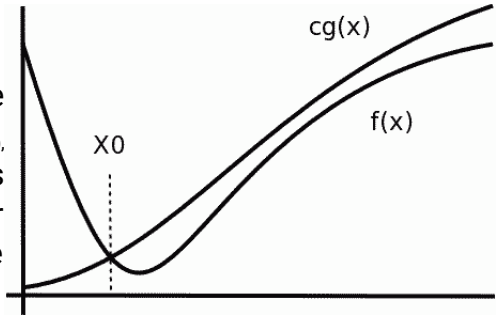
```
int recursivo(int n)
{
     int i, j = 1, z = 0;
     for (i = 0; i < n; i++)
     {
          j += z;
          z = j;
     }
     return j;
}
```

La función de tiempo de ejecución para la nueva versión de nuestro algoritmo es: `T(n) = 2n + 3`. Pero, ¿cómo decidimos que alternativa utilizar? Para esto analizaremos el orden de cada algoritmo.

Orden de un Algoritmo

Una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende al infinito, o sea, un análisis para valores grandes. Se le llama `Orden de g(x)`, `O(g(x))` o `notación O grande`.

Una definición formal es:



Una función `f(x)` pertenece a `O(g(x))` cuando existe una constante positiva `c` tal que apartir de un valor x_0, `f(x)` no sobrepasa `g(x)`. Quiere decir que la función `f` es inferior a `g` a partir del valor dado salvo por un factor constante. Esto es un poco confuso, pero al momento de calcular el orden puede quedar más claro.

Aunque `O(g(x))` está definido como un conjunto se acostumbra a escribir `f(x) = O(g(x))` en lugar de `f(x) ∈ O(g(x))`.

Las reglas para O son las siguientes:

- Las constantes no importan `O(c*f(n)) = O(f(n))`.
- Regla de la suma:
  `O(f(n) + g(n)) = max(O(f(n)), O(g(n)))`
  `O(f(n)) + O(g(n)) = O(f(n) + g(n))`
- Multiplicación:
  `O(f(n))*O(g(n)) = O(f(n)*g(n))`

Aplicando estas reglas encontraremos el orden del primer algoritmo, “funcion”:

`O(T(n)) = O(3) = O(c)`
`T(n) = O(c)`
El algoritmo es de orden *constante*.

Ahora el de la inserción en la lista enlazada:

`O(T(n)) = O(2n + 3) = O(2n) = O(n)`
`T(n) = O(n)`
El algoritmo es de orden *lineal*.

Ahora, analizamos las dos alternativas para el algoritmo recursivo:

$$T_1(n) = 3/2 * 2^n - 1 \qquad y \qquad T_2(n) = 2n + 3$$

El orden de T_1: $O(3/2 * 2^n - 1) = O(3/2 * 2^n) = O(2^n) = O(c^n)$
$T_1 = O(c^n)$

El orden de T_2: $O(2n + 3) = O(2n) = O(n)$
$T_2 = O(n)$

El orden de T_1 es exponencial, mientras T_2 es lineal. Analizando el orden de ambas funciones, podemos concluir que el segundo algoritmo es más eficiente, ya que para cualquier n y c positivos, $n < c^n$.

De esta manera se clasifican algunos algoritmos:

$O(c)$	constante
$O(n)$	lineal
$O(\log n)$	logarítmico
$O(n \log n)$	loglineal
$O(n^2)$	cuadrático
$O(n^c)$	polinómico
$O(c^n)$	exponencial
$O(n!)$	factorial

Es recomendable evitar los cuadráticos y superiores.

Conociendo esto, podemos analizar la complejidad de la busqueda, inserción, eliminación y distintas operaciones de cualquier estructura de datos que conocemos. Usando este análisis encontramos que la inserción en un arreglo es de tiempo constante, mientras que en una lista simplemente enlazada es lineal, este criterio es importantísimo para decidir que estructura utilizar y cuando utilizarla.

Imaginate que deseas guardar millones de registros; guardar todo en una estructura donde la inserción, busqueda y eliminación sea lineal consumirá demasiados recursos- Es aquí donde el análisis se vuelve indispensable.