Introducción al análisis de algoritmos

Pedro O. Pérez M., PhD.

Programación de estructuras de datos y algoritmos fundamentales
Tecnológico de Monterrey

pperezm@tec.mx

08-2022

¿Cómo analizamos los algoritmos?

- Cuando tenemos varios algoritmos para resolver un mismo problema, necesitamos una forma de determinar la mejor opción.
- La respuesta es el análisis asintótico de complejidad.
- Pero, ¿qué es la complejidad de un algoritmo?
  - Es la medida de los recursos que necesita un algoritmo para su ejecución.
  - Complejidad temporal: El tiempo que necesita un algoritmo para terminar su ejecución.
  - Complejidad espacial: La cantidad de memoria que requiere un algoritmo durante su ejecución.

- El tiempo de ejecución de un algoritmo depende de:
  - Factores externos: La computadora donde se va a realizar la ejecución, el compilador (o interprete) usado, la experiencia del programador, los datos de entrada.
  - Factores internos: El número de instrucciones asociadas al algoritmo.
- Entonces, ¿cómo podemos estudiar el tiempo de ejecución del algoritmo?

- Análisis empírico (a posteriori):
  - Generando ejecuciones del algoritmo para distintos valores de entrada y cronometrando el tiempo de ejecución.
  - Factores internos: Los resultados dependen de factores externos e internos.
- Análisis analítico (a priori):
  - Obtener una función que represente el tiempo de ejecución del algoritmo para cualquier valor de entrada.
  - Depende solo de los factores internos.

¿Big Ω, Big Θ?, Big O?

- Cuando analizamos un algoritmos debemos tener en cuenta tres situaciones:
  - El mejor de los casos (Cota inferior - Ω(n))
  - El caso promedio (Cota promedio - Θ(n))
  - El peor de los casos (Cota superior - O(n))

Notación O	Nombre
$O(1)$	Constante
$O(\log \log(n))$	log log
$O(\log(n))$	Logarítmica
$O(n)$	Lineal
$O(n \log n)$	n log n
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(n^m)$	Polinomial
$O(m^n)$ m >= 2	Exponencial
$O(n!)$	Factorial

N	10	100	1,000	10,000	100,000
$O(1)$	1 μs	1 μs	1 μs	1 μs	1 μs
$O(\log n)$	3 μs	7 μs	10 μs	13 μs	17 μs
$\sqrt{n}$	3 μs	10 μs	31 μs	100 μs	316 μs
n	10 μs	100 μs	1,000 μs	10,000 μs	100,000 μs
$n \log n$	33 μs	664 μs	10,000 μs	133,000 μs	1.6 *seg*
n^2	100 μs	10,000 μs	1 *seg*	1.7 min	16.7 min
n^3	1 *ms*	1 *seg*	16.7 min	11.6 *dia*	31.7 *año*
2^n	1.024 *ms*	$4*10^{16}$ *año*	$3.39*10^{287}$ *año*	...	...
$n2^n$	10.24 *ms*	$4*10^{18}$ *año*	...	...	...
$n!$	4 *seg*	$2.95*10^{144}$ *año*	...	...	...

La sentencias simples son aquellas que ejecutan operaciones básicas, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño está relacionado con el tamaño del problema. La inmensa mayoría de las sentencias simples requieren un tiempo constante de ejecución y su complejidad es $O(1)$.

Ejemplos:

$x \leftarrow 1$

$y \leftarrow z + x + w$

print x

read x

Los condicionales suelen ser $O(1)$, a menos que involucren un llamado a un procedimiento, y siempre se debe tomar la peor complejidad posible de las alternativas del condicional, bien en la rama afirmativa o bien en la rama positiva. En decisiones múltiples (*switch*) se tomará la peor de todas las ramas.

Ejemplo:

if $a > b$ **then**
  for $i \leftarrow 1$ **to** n **do**
    $sum \leftarrow sum + 1$
  end for
else
  $sum \leftarrow 0$
end if

En los ciclos con un contador explícito se distinguen dos casos: que el tamaño n forme parte de los límites del ciclo, con una complejidad basada en n, o que dependa de la forma como avanza el ciclo hacia su terminación.

Si el ciclo se realiza un número constante de veces, independientemente de n, entonces la repetición solo introduce una constante multiplicativa que puede absorberse, lo cual da como resultado $O(1)$.
Ejemplo:

for $i \leftarrow 1$ **to** k **do**
  sentencias simples $O(1)$
end for

Si el tamaño n aparece como límite de las iteraciones, entonces la complejidad será: n * $O(1) \rightarrow O(\text{n})$.

Si los ciclos son aninadados...

Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    sentencias simples $O(1)$
  end for
end for

En este caso, la complejidad sería: n * n * $O(1) \rightarrow O(n^2)$.

Para ciclos anidadados pero con variables independientes:
Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** i **do**
    sentencias simples $O(1)$
  end for
end for

$$\sum_{i=1}^{n} \sum_{j=1}^{i} \mathrm{O}(1) = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

A veces aparecen ciclos multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores):
Ejemplo:

$c \leftarrow 1$
while $c < n$ **do**
  $c \leftarrow c * 2$
end while

El valor inicial de la variable c es 1, y llega a 2^n al cabo de n iteraciones $\rightarrow \log_2 n$.

Y la combinación de los anteriores:
Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  $c \leftarrow n$
  while $c > 0$ **do**
    $c \leftarrow c/2$
  end while
end for

Se tiene un ciclo interno de orden $O(\log_2 n)$ que se ejecuta n veces en el ciclo externo; por lo que, el ejemplo es de orden $O(n \log_2 n)$.

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí.
Ejemplo:

$$
\begin{aligned}
&a \leftarrow 10\\
&b \leftarrow 20\\
&c \leftarrow FACTORIAL(a)\\
&z \leftarrow a + b + c
\end{aligned}
$$

Si se tiene un ciclo con un llamado a una función:
Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  $x \leftarrow FACTORIAL(i)$
end for

Si hay un ciclo que se realiza n veces, lo que generaría una complejidad O(n); pero como en su interior hay un llamado a la función *FACTORIAL*, la complejidad del ciclo es multiplicado por la complejidad de la función; en este caso sería $O(\mathrm{n}) * O(\mathrm{n}) \rightarrow O(n^2)$

Si hay dos o más llamadas a funciones:
Ejemplo:

$QUICKSORT(array, n)$
$DISPLAY(array, n)$

La complejidad del $QUICKSORT$ es de complejidad $O(n \log_2 n)$ y que $DISPLAY$ simplemente muestra el contenido del arreglo en la pantalla con una complejidad de O(n), la complejidad total será mayor de los dos llamadas a las funciones, $O(n \log_2 n)$.

- La función *floor*(x) devuelve el entero más pequeño o igual a x. Por ejemplo, $floor(3.3) = floor(3.99999) = floor(3.5) = 3$.
- La función *ceil*(x) devuelve el entero más grande o igual a x. Por ejemplo, $ceil(3.3) = ceil(3.99999) = ceil(3.5) = 4$.

- $\log_b a$ es una función estrictamente creciente.
- $\log_b 1 = 0$.
- $\log_b b^a = a$.
- $\log_b(XY) = \log_b X + \log_b Y$
- $\log_b X^a = a \log_b X$
- $X^{\log_b Y} = Y^{\log_b X}$
- $\log_c X = \frac{\log_b x}{\log_b c}$

1. $\sum_{i=1}^{n} c = c * n$
2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6}$

Listing 1: Return the greatest element of an array

```
int maxVal(int *A, int n) {
  int val = A[0];
  for (i = 1; i < n; i++) {
    if (A[i] > val) {
      val = A[i];
    }
  }
  return val;
}
```

Listing 2: Calculate the average of the elements of an array

```
double average(int* A, int n) {
  int acum = 0;
  for (int i = 0; i < n ; i++) {
    acum = acum + A[i];
  }
  return (acum / (double) n);
}
```

Listing 3: Calculate exponentiation by squaring

```
double pow2(double x, int n) {
  double result = 0;
  while (n > 0) {
    if (n % 2 == 1) {
      result = result * x;
    }
    n = n / 2;
    x = x * x;
  }
  return result;
}
```

Listing 4: Perform multiplication of square matrices

```
void multMat(int** A, int** B, int** C, int n) {
  for (int i = 0; i < n, i++) {
    for (int j = 0; j < n; j++) {
      C[i,j] = 0;
      for (int k = 0; k < n; k++) {
        C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
      }
    }
  }
}
```

Listing 5: Calculate the fibonacci number of n

```
int fibonacci(int n) {
  int previous, current, aux;

  previous = 1;
  current = 1;
  while (n > 2) {
    aux = previous + current;
    previouse = current;
    current = aux;
    n = n - 1;
  }
  return current;
}
```

Para poder analizar la eficiencia de los algoritmos recursivos, se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva. Para ello, utilizaremos el método de análisis inductivo.

Listing 6: Calculate the power x to n

```
double pow(double x, int n) {
  if (n == 0) {
    return 1;
  } else {
    return x * pow(x, n - 1);
  }
}
```

Listing 7: Calculate the power x to n

```
int enigma(int n) {
  if (n <= 0) {
    return 1;
  } else {
    return enigma(n - 1) + enigma(n - 1);
  }
}
```

Listing 8: Calculate the fibonacci number of n

```
int fibonacci(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Listing 9: Calculate exponentiation by recursive squaring

```
double pow2(double x, int n) {
  if (n < 0) {
    return pow2(1/x, -n);
  } else if (n == 0) {
    return 1;
  } else if (n == 1) {
    return x;
  } else if (n % 2 == 0) {
    return pow2(x * x, n / 2);
  } else {
    return x * pow2(x * x, (n - 1) / 2);
  }
}
```

Resumiendo...

Para poder analizar la eficiencia de los algoritmos recursivos, se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva. Normalmente se comportan de una de las siguientes formas:

- $O(n)$ - Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.
- $O(\log_b n)$ - Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.
- $O(C^n)$ - Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.
- $O(n^{\log_b c})$ - Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.