



CS2013

Programación III

Unidad 3: Semana 7

Guía para el análisis asintótico.

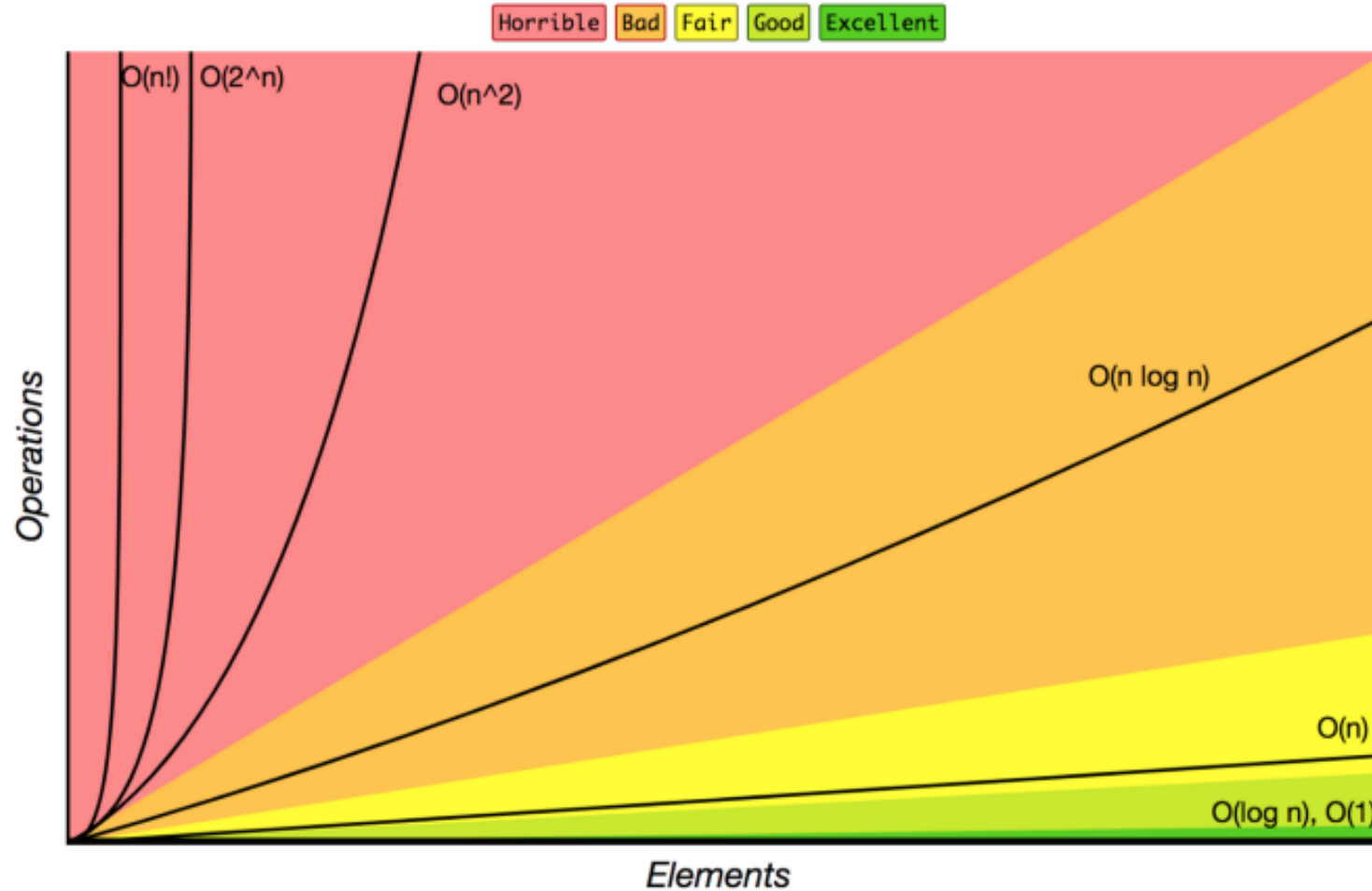
Rubén Rivas

Finalidad de la sesión

Definir reglas generales que ayudan determinar el tiempo de un algoritmo



Orden de Crecimiento



Fuente: <https://medium.com/@cparusso/what-the-hell-is-big-o-notation-9b90d9f9cd14>



Bucles

- El tiempo de ejecución de un bucle usualmente dura el tiempo de ejecución de las instrucciones ejecutado por el número de iteraciones.



```
1 for (int i = 0; i < n; ++i) // n iteraciones
2     m = m + 2;             // tiempo constante = C
3
```

$$Big\ O = C \times n = Cn = O(n)$$

Bucles anidados

- Se analiza desde la instrucción más interna hasta el bucle más externo y se calcula multiplicando las instrucciones internas por el número de iteraciones de cada bucle.



```
1 for (int i = 0; i < n; ++i)           // n iteraciones
2     for (int j = 0; j < n; ++j) // n iteraciones
3         m = m + 2; // tiempo constante = C
4
```

$$Big O = C \times n \times n = Cn^2 = O(n^2)$$

Instrucciones consecutivas

- Se adiciona el tiempo de cada instrucción.

```
1 x = x + 1;           // constante C_0
2 for (int i = 0; i < n; ++i) // n iteraciones
3     m = m + 1;        // constante C_1
4 for (int i = 0; i < n; ++i) // n iteraciones
5     for (int j = 0; j < n; ++j) // n iteraciones
6         k = k + 2;    // constante C_2
7
```

$$Big O = C_0 + C_1n + C_2n^2 = O(n^2)$$

If/else

Se elabora la función de tiempo sumando el número de instrucciones de todos sus componentes y se calcula el ***Big O***.



```
1 if (n >= 0)           // constante C_0
2     k = k + 1         // constante C_1
3 else
4     for (int i = 0; i < n; ++i) // n iteraciones
5         if (m >= 0)     // constante C_2
6             m = m + 1; // constante C_3
7
```

$$Big\ O = C_0 + C_1 + (C_2 + C_3)n = C_4 + C_5n = O(n)$$

Logarítmica

Ocurre cuando a un tiempo constante de cada interacción de un bucle el número de instrucciones se reduce a una fracción de número anterior (usualmente $\frac{1}{2}$).



```
1 for (int i = 1; i < n; i *= 2) // Ejemplo 1
2     cout << i;
3 for (int i = n; i > 1; i /= 2) // Ejemplo 2
4     cout << i;
5 // si n = 8 → las funciones imprimen 3 veces  $\approx \log(n)$ 
6 // si n = 20 → las funciones imprimen 4 veces  $\approx \log(n)$ 
7
```

$$\text{Big O} = O(\log n)$$

Exponencial

Ocurre cuando a un input n el numero de instrucciones crece en forma exponencial.



```
1 int fibonacci(int n) {  
2     if (n <= 1) return n; // constante C_0 = O(1)  
3     return fibonacci(n-1) + fibonacci(n-2); // recursiva  
4 }  
5 // Complejidad es: T(n) = O(1) + T(n-1) + T(n-2)  
6
```

Factorial

Ocorre cuando para n instrucciones el tiempo crece de forma factorial.



```
1 void fact(int n) {  
2     for (int i = 0; i < n; ++i) // n  
3         fact(n-1); // b  
4 }  
5 // Complejidad es:  $O(n*b) \rightarrow O(n*(n-1)*b') \rightarrow O(n*(n-1)*(n-2)*b'')$   
6
```

Recursividad – función decreciente

```
1 void fun1(int n) {
2     // condicion basica
3     if (n == 1) return;
4     // condicion recursiva
5     fun1(n - 1);
6 }
7 // T(n) = C + T(n - 1)
8 // O(n)
9
10 void fun2(int n) {
11     // condicion basica
12     if (n == 1) return;
13     // condicion iterativa
14     int a = 0;
15     for (int i = 0; i < n; ++i)
16         a += i;
17     // condicion recursiva
18     fun2(n - 1);
19 }
20 // T(n) = C + n + T(n - 1)
21 // O(n^2)
22
```

```
1 void fun3(int n) {
2     // condición básica
3     if (n == 1) return;
4     // condición iterativa
5     int a = 0;
6     for (int i = 0; i < n; ++i)
7         for (int j = 0; j < n; ++j)
8             a += i + j;
9     // condición recursiva
10    fun3(n - 1);
11 }
12 // T(n) = C + n^2 + T(n - 1)
13 // O(n^3)
14
15 void fun4(int n) {
16     // condicion basica
17     if (n == 1) return;
18     // condicion iterativa
19     int a = 0;
20     for (int i = 0; i < n; ++i)
21         for (int j = 0; j < n; ++j)
22             a += i + j;
23     // condicion recursiva
24     fun4(n - 1);
25     fun4(n - 1);
26 }
27 // T(n) = C + n^2 + 2 T(n - 1)
28 // O(2^n * n^2)
29
```

Teorema master de función decreciente

Dada un función f que satisfaga el siguiente patrón recursivo:

$$f(n) = af(n - b) + cn^d$$

$$\text{if } a < 1 \Rightarrow O(n^d)$$

$$\text{if } a = 1 \Rightarrow O(n * n^d)$$

$$\text{if } a > 1 \Rightarrow O(a^{n/b} * n^d)$$

Recursividad – función dividida

```
1 void fun1(int n) {
2     // condicion basica
3     if (n == 1) return;
4     // condicion recursiva
5     fun1(n/2);
6 }
7 // T(n) = C + T(n/2)
8 // O(log(n))
9
10 void fun2(int n) {
11     // condicion basica
12     if (n == 1) return;
13     // condicion iterativa
14     int a = 0;
15     for (int i = 0; i < n; ++i)
16         a += i;
17     // condicion recursiva
18     fun2(n/2);
19 }
20 // T(n) = C + n + T(n/2)
21 // O(n)
22
```

```
1 void fun3(int n) {
2     // condicion basica
3     if (n == 1) return;
4     // condicion iterativa
5     int a = 0;
6     for (int i = 0; i < n; ++i)
7         for (int j = 0; j < n; ++j)
8             a += i + j;
9     // condicion recursiva
10    fun3(n/2);
11 }
12 // T(n) = C + n^2 + T(n/2)
13 // O(n^2)
14
15 void fun4(int n) {
16     // condicion basica
17     if (n == 1) return;
18     // condicion iterativa
19     int a = 0;
20     for (int i = 0; i < n; ++i)
21         a += i + j;
22     // condicion recursiva
23     fun4(n/2);
24     fun4(n/2);
25 }
26 // T(n) = C + n + 2 T(n/2)
27 // O(n log2(n))

```

Teorema master de función división

Dada un función f que satisfaga el siguiente patrón recursivo:

$$f(n) = af(n/b) + cn^d$$

- Siendo $n = b^k$ donde k es un número entero positivo, $a \geq 1$, b es un entero mayor que 1 y c y d son números real, siendo c y d positivo, entonces:

$$f(n) = \begin{cases} O(n^d) & \text{si } a < b^d \\ O(n^d \log n) & \text{si } a = b^d \\ O(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Algunas formulas útiles

Serie Aritmética:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n-1)}{2}$$

Serie Geométrica:

$$\sum_{k=1}^n x^k = 1 + x + x^2 \dots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$$

Algunas formulas útiles

Serie Geométrica genérica:

$a = \text{valor inicial}$

$r = \text{ratio}$

$s = a, ar, ar^2, ar^3, ar^4, \dots ar^n, t_n = ar^{n-1}$

$$\sum_{i=1}^n ar^i = a + ar + ar^2 \dots + ar^n = a \left(\frac{1 - r^n}{1 - r} \right) = a \left(\frac{r^n - 1}{r - 1} \right)$$

Algunas formulas útiles

Serie Armónica:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \log n$$

Otras formulas:

$$\sum_{k=1}^n k^p = 1^p + 2^p + \dots + n^p \approx \frac{1}{p-1} (x \neq 1)$$


Algunas formulas útiles

Otras formulas:

$$\sum_{k=1}^n \log k = n \log n$$

Ejercicio #1

¿Cuál es el tiempo de ejecución de fun()?



```
1 int fun(int n) {  
2     int count = 0;  
3     for (int i = 0; i < n; ++i) {  
4         for(int j = 0; j < n; ++j) {  
5             count += 1;  
6         }  
7     }  
8     return count;  
9 }  
10
```

Ejercicio #1

```
1 int fun(int n) {  
2     int count = 0; // constante C_0  
3     for (int i = 0; i < n; ++i) { // n iteraciones  
4         for(int j = 0; j < n; ++j) { // n iteraciones  
5             count += 1; // constante C_1  
6         }  
7     }  
8     return count; // constante C_2  
9 }  
10
```

$$Big O = C_0 + C_1n^2 + C_2 = O(n^2)$$

Ejercicio #2

Cual es el tiempo de ejecución del siguiente algoritmo?



```
1 void recursive(int n) {  
2     if (n <= 1) return;  
3     cout << "utec ";  
4     recursive(n/2);  
5     recursive(n/2);  
6 }  
7
```

Solución

Solución:

La función se puede expresar como:

$$T(n) + C + 2T(n/2)$$

Sigue el patrón:

$$f(n) = af(n/b) + c$$

donde $c = C$, $a = 2$ y $b = 2$ por tanto $O(n^{\log_b a})$ si $a > 1$ seria:

$$\therefore O(n^{\log_2 2}) = O(n)$$

Ejercicio #3

- Determinar el big O del algoritmo



```
1 void loop(int n, int k) {  
2     for (int i = 0; i < n; ++i) {  
3         for (int j = 1; j < n; j = j*k) {  
4             cout << "utec ";  
5         }  
6     }  
7     cout << endl;  
8 }  
9
```


Solución



```
1 void loop(int n, int k) {  
2     for (int i = 0; i < n; ++i) {           // O(n)  
3         for (int j = 1; j < n; j *= k) {    // O(logk(n))  
4             cout << "utec ";  
5         }  
6     }  
7     cout << endl;  
8 }  
9  
10 // La ecuación es:  $O(n) * O(\log k(n)) + C$   
11 //  $O(n * \log k(n))$   
12
```