# Tarea 1 Análisis de Algoritmos

**Javier Arriagada Inostroza , Ignacio Sanhueza Becar**

May 27, 2020

**Pregunta 1:** a), e), b), d), c)

**Pregunta 2:**

a) **T(n) = 4T(n/4) +5n**

Por Teorema Maestro: a=4, b=4, c=1;
Analizando:

$$log_4 4 = 1$$

Como

$$c = log_4 4 = 1$$

nos queda una complejidad de

$$O(nlogn)$$

b) **T(n) = 4T(n/5) +5n**

Por Teorema Maestro: a=4, b=5, c=1;
Analizando:

$$log_5 4 = 0,8$$

Como

$$c > log_4 4 = 0.8$$

nos queda una complejidad de

$$O(n)$$

c) **T(n) = 5T(n/4) +4n**

Por Teorema Maestro: a=4, b=5, c=1;
Analizando:

$$log_4 5 = 1.1$$

Como

$$c < log_4 = 1.15$$

nos queda una complejidad de

$$O(n^{log_4 5})$$

d) $\mathbf{T(n) = 4T(\sqrt{n}) + log^5(n)}$

Sea $k = logn$

$$n = 2^k$$

$$T(2^k) = 4T(2^{k/2}) + log^5(2^k)$$

$$T(2^k) = 4T(2^{k/2}) + k^5$$

Definiendo:

$$S(m) = T(2^k)$$

$$S(m) = 4T(m/2) + m^5$$

Teorema Maestro

$$a = 4; b = 2; c = 5$$

$$log_2 4 = 2$$

$$c > log_2 4 = 2$$

Por lo tanto, la complejidad es de $O(m^5)$
Para $T(n) = O(log^5 n)$

**Pregunta 3:**

**a)** Definimos $f(n) = n^{\frac{n}{2}}$ y $g(n) = 2^n$

$$\lim_{x \to \infty} (\frac{2^{\frac{n}{2}}}{2^n}) = 0$$

Por lo tanto la afirmación es **FALSA**, ya que, para que $2^{\frac{n}{2}} \in O(2^n)$ el límite debe ser mayor a cero

**b)** Definimos $f(n) = n^{\frac{3}{2}}$ y $g(n) = nlog^2(n)$

$$\lim_{x \to \infty} (\frac{n^{\frac{3}{2}}}{nlog^2(n)}) = \infty$$

La afirmación es **FALSA**

**c)** Si f(n) = O(g(n)) entonces log(f(n)) = O(log(g(n)))

Para todo $a, b < 1$, tenemos que $log_a(n) \in O(log_b(n))$.

Por lo tanto es **VERDADERO**

**d)** Las complejidades tienen una relación de orden dada por:

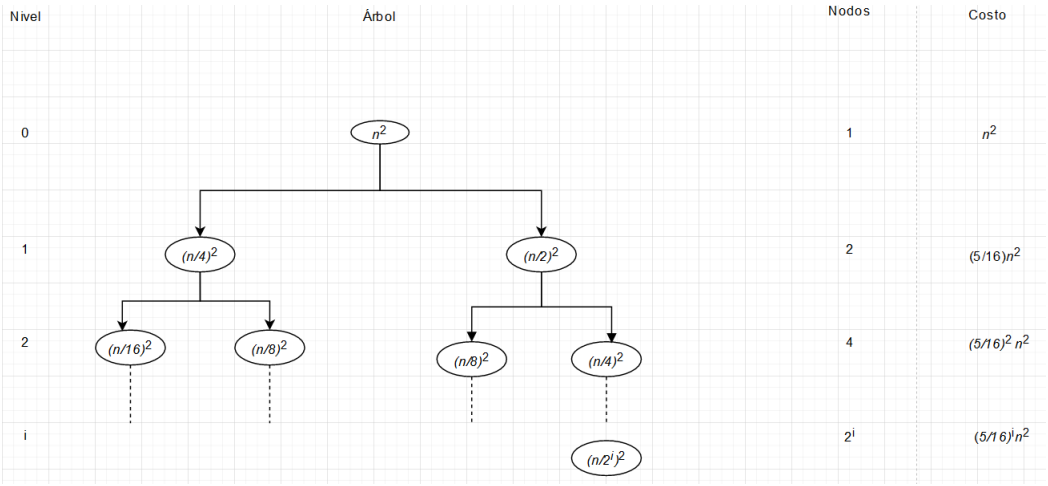
$O(1) \subset O(logn) \subset O(n) \subset O(nlogn) \subset O(n^2) \subset O(c^n) \subset O(n!)$

$O(n^2) \subset 1,0000001^2$

La afirmación es **FALSA**

**Pregunta 4:**

**a)**




Como el árbol es desbalanceado, no podemos asegurar que las recurrencias lleguen al mismo tiempo a algún $T(1) = C$

Profundida del árbol ($logn$)

Luego Tenemos que: Hojas: $2^{logn} = n$

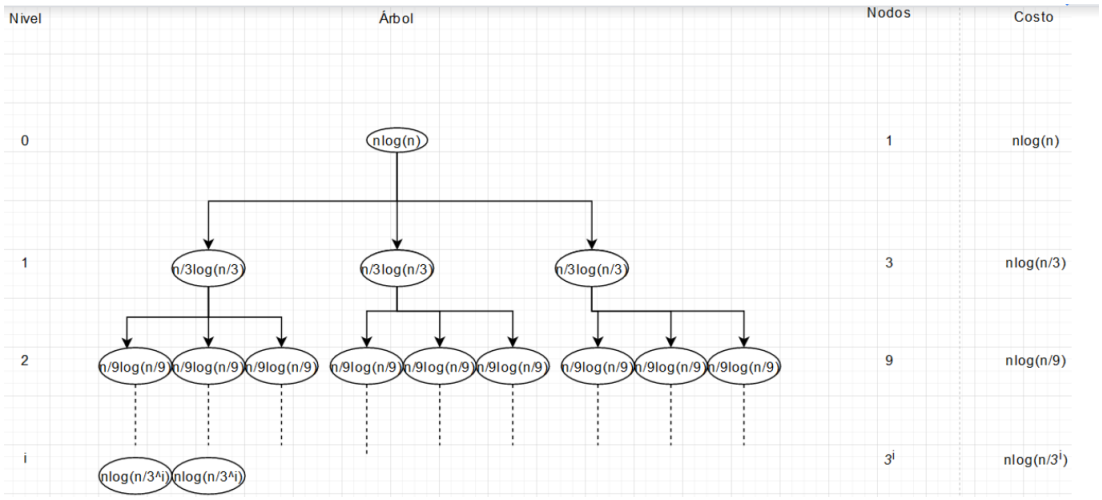
$$n^2 + (\frac{5}{16})^2 n^2 + (\frac{5}{16})^3 n^2 \leq n^2 \sum_{i=0}^{\infty} (\frac{5}{16})^i = (\frac{16}{1}) n^2 = O(n^2)$$

Ahora por substitución

$$T(n) \leq T(n/4) + T(n/2) + n^2$$
$$\leq k(n/4)log(n/4) + k(n/2)log(n/2) + n^2$$
$$= (k(n/4)log(n) - k(n/4)log(4)) + (k(n/2)log(n) - k(n/2)log(2)) + n^2$$
$$= kn^2 - k((n/4)log(4) + (n/2)log(1/2)) + n^2$$

Por lo tanto, la complejidad es $O(n^2)$

**b)**




En algún momento llegaremos a T(1) = C

El camino más largo desde la raíz a la hoja es

$$(nlogn) \to (nlog(n/3)) \to (nlog(n/9)) \to ... \to C.$$

Luego Tenemos que:

$$nlog(n/3^i) = C$$
$$nlog(n) - nlog(3^i) = C$$
$$nlog(n) - C = nlog(3^i)$$
$$nlog(n) - C = inlog(3)$$
$$i = \frac{nlog(n) - C}{(nlog(3))}$$
$$i = \frac{nlog(n)}{(nlog(3))} + \frac{C}{(nlog(3))}$$

En resumen nos queda una suposición de O(nlogn)

Es posible probar mediante substitución que O(nlog(n)) es una cota superior de la siguiente forma:

$$T(n) \leq 3T(n/3) + nlogn$$
$$\leq k(n/3)log(n/3) + nlogn$$
$$= (k(n/3)log(n) - k(n/3)log(3)) + nlogn$$
$$= knlog(n) - k((n/3)log(3)) + nlogn$$
$$= knlog(n) - kn(log(3)) + nlogn$$
$$\leq knlog(n)$$
$$T(n) = O(nlog(n)); k \geq c/(log(3))$$

**Pregunta 5:**

**a)** T(n) = 4T(n/2) + 100n ;k=1

Usando el método iterativo se tiene que:

$$T(n/2) = 4T(n/4) + 50n$$

$$T(n) = 4 * 4T(n/4) + 300n; k = 2$$

$$T(n/4) = 4T(n/8) + 25n$$

$$T(n) = 16 * 4T(n/8) + 700n; k = 3$$

Luego la recurrencia tiene la forma:

$$T(n) = 4^k T(n/2^k) + 100n(2^k - 1)$$

Con este supuesto se procede al método de sustitución

Asumiendo caso base T(1)= C

$$T(n/2^k = 1) = C$$

$$n/2^k = 1$$

$$n = 2^k$$

$$k = log(n)$$

reemplazando k en T(n)= $4^k T(n/2^k) + 100n(2^k - 1)$

$$T(n) = 4^{log2(n)} T(n/2^{log(n)}) + 100n(2^{log(n)} - 1)$$

$$T(n) = n^2 + T(1) + 100n(n - 1)$$

$$T(n) = n^2 + C + 100n^2 - 100n = 101n^2 - 100n + C*$$

$$T(n) \subset O(n^2)$$

Usando inducción:

Caso base: n=1

$$T(1) = 101(1)^2 - 100(1) + C = 1 + C = C$$

Paso inductivo

$$T(n) = 4T(n/2) + 100n$$

$$= 4(101(n/2)^2 - 100(n/2) + C) + 100n$$

$$101n^2 - 200n + 100n + 4C = 101n^2 - 100n + 4C = 101n^2 - 100n + C*$$

Asi queda demostrado.

**b)** $T(n) = 4Y(\sqrt{n}) + log^5(n)$

Como se vio en la respuesta de 4.-D, Analizamos sobre $O(log^5(n)$

T(n)$\geq clog^5, n > n_0$ y $c > 0$

Para (n=1) $T(1) = 1 > clog^5(1)$

Verdadero para todo c¿0

Por hipótesis inductiva asumimos que $m < n$

$$T(m) \geq cm^2 log^5(n)$$

Luego

$$\begin{aligned} T(n) &= 4T(\sqrt{n}) + log^5(n) \\ &\geq 4clog^5(\sqrt{n}) + log^5(n) \end{aligned}$$

**Pregunta 6:**

a)

```
int F(int x, int y){
int m = 1;                  //1
if(y == 0){                 //1
return 1;                   //1
}
while(y > 0){               //n/2
if(y % 2 ){                 //1
m *= x;                     //1
}
x *= x;                     //1
y /= 2;                     //1
}
return m;                   //1
}
```

Analizando el bucle, nos damos cuenta que cada vez que se ejectua, este se divide en dos, osea que a la hora de recorrer el Y, lo recorremos Y/2 veces, por lo tanto, tiene una complejidad de O(logn)

b)

```
int recurse(int n) {
for (i = 0; i < n*n; i += 2) {      //n^2
procesar(i); // O(n)}               //n
if (n <= 0)                         //1
return 1;                           //1
else
return recurse(n-3);}               //T(n-3)
```

Peor caso: que se ejecute el for. Lo que nos daría una recurrencia del estilo $T(n) = T(n-3) + n^3$, porque cada vez que enra en el bluce, lo recorrerá $n^2$ veces, y ademas realizará una operacion de costo $n$ y $n * n^2 = n^3$

Finalmente tenemos que la complejidad para esta recurrencia es de $O(n^3)$

**Pregunta 7:**

a)

```
nCuadrado(pair<int,int> A) {
int luz;                                                    //
    1
for (i = 0; i < A.size; i ++) {                             //
    n
A[i].first <- pair.first
A[i].second <- pair.second
}
sort(A[].first);                                            //
    n^2

int i=0;                                                    //
    1

for (j = 1; j < A.size; j ++) {                             //
    n
        if(arr[i].second<=arr[j].first){
            z=j;
            luz++;
        }
        else if(arr[i].second<=arr[j].second){
            z=j;

        }
}
```

El sort dentro de nCuadraro, en el codigo es un bluce anidado, por lo tanto la función que domina al resto es de orden $O(n^2)$, y por consecuencia el código corre en un tiempo de $O(n^2)$ más claro $1 + n + n^2 + 1 + n(5)$

```
nLogaritmica(pair<int,int> A) {
int luz;                                                    //1
for (i = 0; i < A.size; i ++) {                             //n
A[i].first <- pair.first
A[i].second <- pair.second
}
sort(A[].first);                                //nlogn

int i=0;                                                    //1

for (j = 1; j < A.size; j ++) {                             //n
        if(arr[i].second<=arr[j].first){                    //1
            z=j;                                            //1
            luz++;                                          //1
        }
        else if(arr[i].second<=arr[j].second){              //1
            z=j;                                            //1

        }
}
```

El sort dentro de nLogaritmo, en el codigo es el de la stl, que tiene un tiempo de peor caso de $O(nlogn)$, sumando todas las operaciones tenemos $1 + n + nlogn + 1 + n(5)$, luego el tiempo en complejidad es de $O(nlogn)$

Ambos algoritmos reciben un arreglo de pares, el cual es ordenado comparando sólo el primer valor de cada par, A[].first. En el primer caso se ordena en tiempo de $O(n^2)$ y en segundo caso $O(nlogn)$. Además se asume que el valor de luz es 1, porque siempre entrará a lo menos una persona en la habitación. Luego se compara el arreglo de la siguiente forma

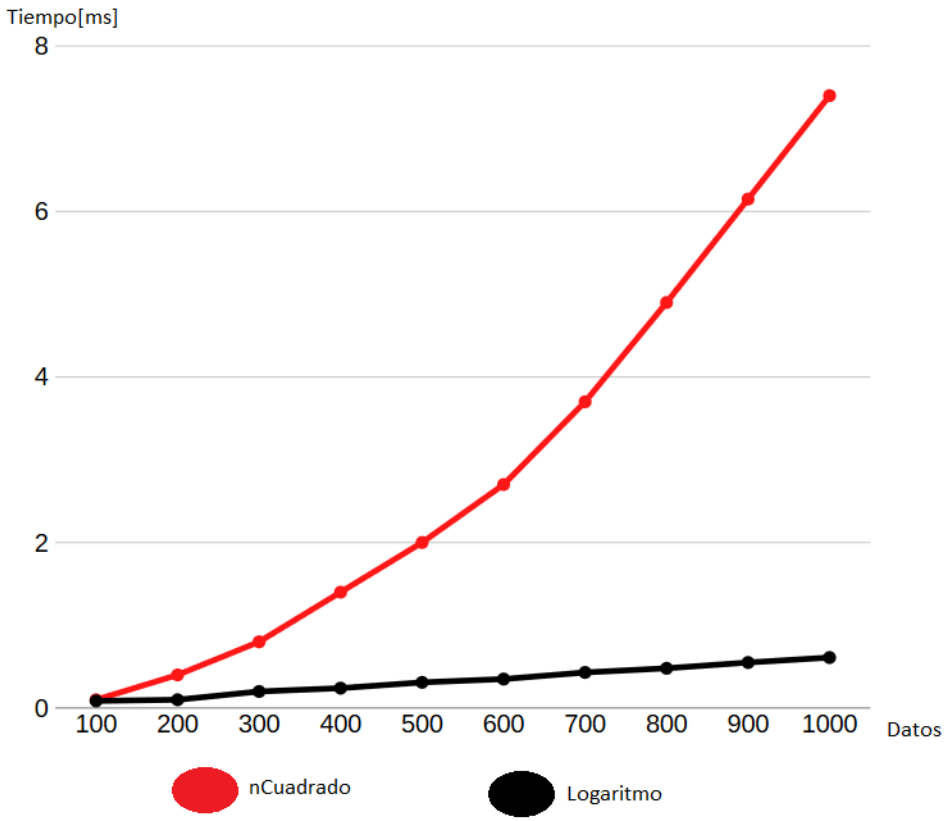
* Sí A[i].second $\leq$ A[i+1].first, significa que una persona primero sale para que entre otra, por lo tanto se apaga y se prende la luz, añadiendo uno a la variable luz, y ahora el arreglo queda en la posición A[i+1]

* Sí A[i].second $\leq$ A[i+1].second, significa que una persona entra mientras ya tenemos otra dentro de la habitación, pero sabemos que el tiempo de estancia de la persona que llegó es mayor al de la persona que esta en la sala, por lo tanto sólo cambiamos el arreglo de posición a A[i+1], y seguimos iterando

* Es claro que el arreglo sólo itera una vez, dado que al estar ordenado, no es necesario hacer nada más que los puntos anteriores. En cualquier otro caso, el arreglo solo itera hasta llegar al tamaño del vector

| Datos | nCuadrado | nLog |
|---|---|---|
| 100 | 128321[ns] | 86132[ns] |
| 200 | 461104[ns] | 128611[ns] |
| 300 | 777037[ns] | 210365[ns] |
| 400 | 1433277[ns] | 248826[ns] |
| 500 | 1997314[ns] | 315432[ns] |
| 600 | 2749995[ns] | 352521[ns] |
| 700 | 3688144[ns] | 437049[ns] |
| 800 | 4923741[ns] | 487213[ns] |
| 900 | 6252382[ns] | 555822[ns] |
| 1000 | 7480855[ns] | 615674[ns] |

Table 1: Tiempos ejecución algoritmos



*Los tiempos en el gráfico fueron pasados a milisegundos para poder analizar de manera más fácil. Viendo el gráfico se nota más claro lo mencionado al analizar el código, el algoritmo nCuadrado se demora más en ejecutar , porque la operacion de ordenamiento es mas costosa que la que hace nLogaritmo