Análisis de Algoritmos: Proyecto 1

Fabián Cid Escobar, Rodolfo Fariña Reisenegger

Profesora: Cecilia Hernández Rivas

Mayo 27, 2020

1. Introducción

En el presente informe se entregarán las soluciones a los problemas a resolver entregados por la profesora. Los temas abarcados por los problemas eran sobre análisis asintótico, análisis de correctitud, recurrencias y un problema a resolver con código.

2. Ejercicios a resolver

1. Ordene de menor a mayor orden asintótico las siguientes funciones:

   a) $3^{2^{1000000}}$
   b) $n\sqrt[3]{n}$
   c) $3^{0.1n}$
   d) n^2
   e) $log(n)^{2\ log(n)}$

Para este ejercicio, proponemos el orden a < b < d < e < c

Notamos que $3^{2^{1000000}}$ es una constante, por lo cual a) es de orden O(1), como todas las otras funciones dependen de algún valor *n* éstas necesariamente van a ser de orden mayor.

También sabemos que $n\sqrt[3]{n} = n^{4/3}$ y esto es menor que n^2, por esto, b < d.

Comparamos *d* con *e*. Nos damos cuenta que $log(n)^{2\ log\ (n)} = n^{2\ log\ (log(n))}$ por propiedades de los logaritmos. Como el exponente va creciendo con n, entonces necesariamente e es mayor que d. Luego e > d

Con esto, ya sabemos que a < b < d < e.

Ahora veamos $log(n)^{2\ log(n)}$ con $3^{0.1n}$. Podemos hacer $3^{0.1n} = (3^{0.1})^n \approx 1.116^n$

Por propiedades de exponenciales podemos decir que

$n^{2\ log\ (log\ n)} = e^{2\ log\ (log\ n) * log\ n}$ …(1) y que $1.116^n = e^{n\ log\ 1.116}$ …(2)

Entonces, comparamos los exponentes de (1) y (2) $2\ log\ (log\ n) * log\ n$ con $n\ log\ 1.116$. Con esto, es más fácil ver las complejidades asintóticas.

Nos damos cuenta que 2 log(log n) * log n representa una complejidad temporal poly-logarítmica y log-logarítmica, lo cual es siempre menor que una complejidad de tiempo lineal. Es importante notar que, estas no representan las complejidades reales, puesto que estamos analizando los exponentes luego de hacer $e^{(\ )}$ para así facilitar el análisis. Por esto, podemos decir que para n muy grande, $3^{0.1n} > log(n)^{2\ log(n)}$. Por esto, concluimos que e < c.

Finalmente, podemos decir que el orden será a < b < d < e < c.

2. Resuelva las siguientes recurrencias:

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

Notamos que tiene la forma para utilizar el Teorema del Maestro con:

$$f(n) = 5n \in \Theta(n)\ ,\ a = 4\ ,\ b = 4$$

Ahora vemos que $log_b(a) = log_4(4) = 1$ y que pertenece al segundo caso, donde

$$f(n) = 5n \in \Theta(n^{log_4(4)} log^k(n) = n^1)\ con\ k = 0,\ entonces\ T(n) \in \Theta(n^1 log(n))$$

Comprobamos que f(n) sea Θ(n) con el teorema del límite:

$$\lim_{n\to\infty}\left(\frac{5n}{n}\right) = 5$$

Como el límite es $0 < 5 < \infty$, entonces f(n) es Θ(n). Por lo tanto:

$$T(n) \in \Theta(nlog(n))$$

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

Aquí también se puede utilizar el Método del Maestro con:

$$f(n) = 5n \in \Theta(n)\ ,\ a = 4\ ,\ b = 5$$

Vemos que $log_b(a) = log_5(4) < 1$ y que pertenece al tercer caso con $\varepsilon \approx 0.2$ debido a que:

$$5n \in (n^{log_5(4)+\varepsilon}) \wedge 4n \leq c * 5n \quad con\ c > 4/5$$

Por teorema del límite comprobamos que $5n \in (n^{log_5(4)+\varepsilon})$:

$$\lim_{n\to\infty}\left(\frac{5n}{n^{log_4(5)\ +\ \varepsilon}}\right) = \lim_{n\to\infty}\left(\frac{5n}{n}\right) = 5$$

Como el límite es $0 < 5 < \infty$, entonces f(n) es $\Theta(n) \to \Omega(n) \wedge O(n)$. Por lo tanto:

$$T(n) \in \Theta(n)$$

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

Notamos que esta recurrencia también puede ser resuelta por Teorema del Maestro con:

$$f(n) = 4n \in \Theta(n) \ , \ a = 5 \ , \ b = 4$$

Vemos que $log_4(5) > 1$ y que corresponde al primer caso debido a:

$$4n \in O(n^{log_4(5) - \varepsilon}) \text{ porque } log_4(5) > 1 \text{ con } \varepsilon \approx 0.16 > 0$$

Lo comprobamos por teorema del límite:

$$\lim_{n \to \infty} \left(\frac{4n}{n^{log_4(5) - \varepsilon}} \right) => \lim_{n \to \infty} \left(\frac{4n}{n^1} \right) = 4$$

y como se cumple, podemos decir que:

$$T(n) \in O(n^{log_4(5)})$$

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

Nos damos cuenta que este ejercicio será muy difícil de resolver sin utilizar una sustitución de variable. Utilizamos el reemplazo m = $log_2(n)$.

Como $m = log_2(n)$ entonces $2^m = 2^{log_2 n} = n$

Reemplazando en la ecuación

$$T(2^m) = 4T(2^{m/2}) + log_2{}^5(2^m) = 4T(2^{m/2}) + m^5$$

También podemos decir que S(m) = T(2^m)

Entonces,

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

Con esta recurrencia, podemos ver fácilmente que podemos utilizar el Teorema del Maestro con:

$$f(m) = m^5 \text{ , } a = 4, \ b = 2$$

Lo cual sería el tercer caso del teorema del maestro. Debemos comprobar las condiciones de este.

Queremos probar que f(m) = $\Omega(m^{log_b a+\varepsilon})$, con $\varepsilon > 0$

$$log_b a = log_2 4 = 2$$

Con $2 + \varepsilon = 5$ con $\varepsilon > 0$ podemos usar el teorema del límite para demostrar que es $\Omega(m^{log_b a+\varepsilon})$

$$\lim_{n\to\infty}\left(\frac{n^5}{n^5}\right) = 1$$

Como es 1, entonces f(n) = $\Theta(m^{2+\varepsilon}) \rightarrow \Omega(m^{2+\varepsilon})$

Necesitamos probar que f(n) satisface la condición de regularidad. La condición de regularidad está definida como

$$af(m/b) \leq cf(m)$$

Para alguna constante $c < 1$ y todo m suficientemente grande. Reemplazando nuestros datos obtenemos:

$$4(m/2)^5 \leq cm^5$$
$$\frac{4}{32}m^5 \leq cm^5$$
$$\frac{1}{8} \leq c$$

Con esto, para $c \geq 1/8$ y m suficientemente grande se cumple la condición de regularidad.

Como ambas condiciones se cumplen, S(m) = $\Theta(m^5)$ y esto corresponde a $\Theta(log^5(n))$.

3) Determine si las siguientes afirmaciones son verdaderas o falsas. Justifique su respuesta:

a) $2^{n/2} es\ \Theta(2^n)$

Es falso, ya que $2^{n/2} es\ O(2^n)$.

Demostración:
Por teorema del límite:

$$\lim_{n\to\infty}\left(\frac{2^{\frac{n}{2}}}{2^n}\right) = \lim_{n\to\infty}\left(\frac{1}{2^{\frac{n}{2}}}\right) = 0$$

Como el límite es 0, nos indica que es $O(2^n)$, excluyendo ser $\Omega\ (2^n)$.

Otra justificación puede ser que el exponente n/2 siempre estará por debajo del exponente n cuando sus bases son iguales.

Por lo tanto la afirmación es falsa.

b) $n^{3/2}\ es\ O(n\ log^2(n))$

Es falso, lo demostramos por teorema del límite:

$$\lim_{n\to\infty}\left(\frac{n^{\frac{3}{2}}}{nlog^2(n)}\right) = \lim_{n\to\infty}\left(\frac{n^{\frac{1}{2}}}{log^2(n)}\right) = \infty$$

Lo que nos indica que $n^{3/2}\ es\ \Omega(nlog^2(n))$. Por lo tanto la afirmación es falsa.

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

Verdadero, dado que si $f(n) = O(g(n))$ entonces existe una constante c > 0 tal que

$$0 \leq f(n) \leq cg(n)$$

Para todo $n \geq n_0$ con n suficientemente grande y asumimos $f(n) \geq 1$

Como $cg(n)$ es siempre mayor que $f(n)$ dado un n suficientemente grande entonces podemos hacer log de ambos lados sin cambiar sentido de la desigualdad y obtenemos

$$log\, f(n) \geq log(c * g(n)) \Rightarrow log\, f(n) \leq log\, c + log\, g(n)$$

Pero, nosotros sabemos que $log\, c \leq log\, g(n)$ para un n suficientemente grande, reemplazando esto obtenemos

$$log\, f(n) \leq 2 * log\, g(n)$$

Si redefinimos 2 como una nueva constante d

$$log\, f(n) \leq d * log\, g(n)$$

Lo cual es justamente la definición de Big-O. Hemos demostrado que para n suficientemente grande, f(n) = O(g(n)) implica log(f(n)) = O(log(g(n)). Por esto, es verdadera.

d) $f(n) = 1.0000001^n$ *es* $O(n^2)$

Usamos el teorema del límite para determinar la veracidad de esta sentencia y aplicamos L'Hopital:

$$\lim_{n\to\infty}\left(\frac{1.0000001^n}{n^2}\right) = \lim_{n\to\infty}\frac{1.0000001^n \ln(1.0000001)}{2n}$$

Aplicamos L'Hopital una vez más obtenemos lo siguiente

$$\lim_{n\to\infty}\frac{1.0000001^n \ln(1.0000001)}{2n} = \lim_{n\to\infty}\frac{1.0000001^n \ln(1.0000001)\ln(1.0000001)}{2} = \infty$$

Como el límite tiende a infinito, podemos decir que $f(n) \in \Omega(g(n))$ y no a *O(g(n))*. Por esto, es falsa.

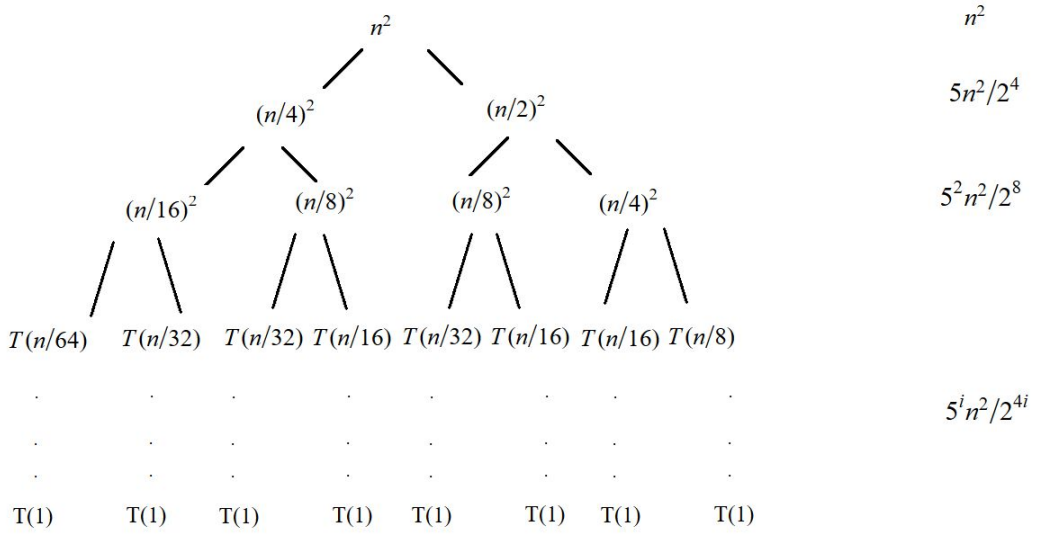
4) Construya los árboles recursivos para las siguientes recurrencias y úselo con el método de substitución para demostrar la solución de la recurrencia

a) $T(n) = T(n/4) + T(n/2) + n^2$
b) $T(n) = 3T(n/3) + nlog(n)$

a) $T(n) = T(n/4) + T(n/2) + n^2$

Para construir el árbol hacemos algunos reemplazos en la recurrencia

$T(n/2) = T(n/8) + T(n/4) + (n/2)^2$
$T(n/4) = T(n/16) + T(n/8) + (n/4)^2$
$T(n/8) = T(n/32) + T(n/16) + (n/8)^2$
$T(n/16) = T(n/64) + T(n/32) + (n/16)^2$
$T(n/32) = T(n/128) + T(n/64) + (n/32)^2$
…
Con estos valores calculados construimos nuestro árbol



Al construir el árbol nos damos cuenta que el costo de cada nivel puede ser expresado como

$$n^2 \left(\frac{5}{2^4}\right)^i$$

Ahora, para calcular la altura nosotros sabemos que el árbol llegara a su altura máxima en la rama que avanza más lento, es decir, el peor caso del algoritmo, la cual es T(n/2), esta llegara al caso base cuando

$$n/2^i = 1 \Rightarrow\ n = 2^i \Rightarrow log_2 n\ =\ i$$

Entonces, podemos estimar el costo total como la suma de todos los niveles hasta que se llegue al caso base

$$\sum_{i=0}^{log_2 n-1} n^2(\tfrac{5}{2^4})^i$$

Lo cual podemos resolver de la siguiente manera

$$\sum_{i=0}^{log_2 n-1} n^2(\tfrac{5}{2^4})^i\ =\ n^2\ \tfrac{1-(5/16)^{log_2 n}}{1-(5/16)} =\ \tfrac{16}{11}n^2(1-(\tfrac{5}{16})^{log_2 n})$$

Luego, para obtener la aproximación de la complejidad podemos decir que $(\frac{5}{16})^{log_2 n}$ disminuye cada vez más y tiende a 0, por lo tanto, podemos despreciar este término para nuestro análisis de peor caso, con lo cual obtenemos

$$\frac{16}{11}n^2$$

Lo cual trivialmente es $\Theta(n^2)$ y representa nuestra aproximación de peor caso.

Luego, por intentamos demostrar esta complejidad utilizando sustitución.
Asumimos T(1) = C

Como hipótesis, tenemos que nuestra complejidad es $\frac{16}{11}n^2$

Demostramos usando principio de inducción matemática.

Caso base: n = 1.

T(1) = (16/11)1^2 = 16/11 = C. Lo cual cumple.

Paso inductivo:

$T(n)\ =\ T(n/4)\ +T(n/2)\ +\ n^2$

$T(n)\ =\ 16/11\ *\ n^2/16\ +16/11 * n^2/4\ +\ n^2$ //Por HIP de inducción

$T(n)\ =\ n^2/11\ +\ 4n^2/11\ +\ 11/11 * n^2$

$T(n)\ =\ 16/11\ n^2$

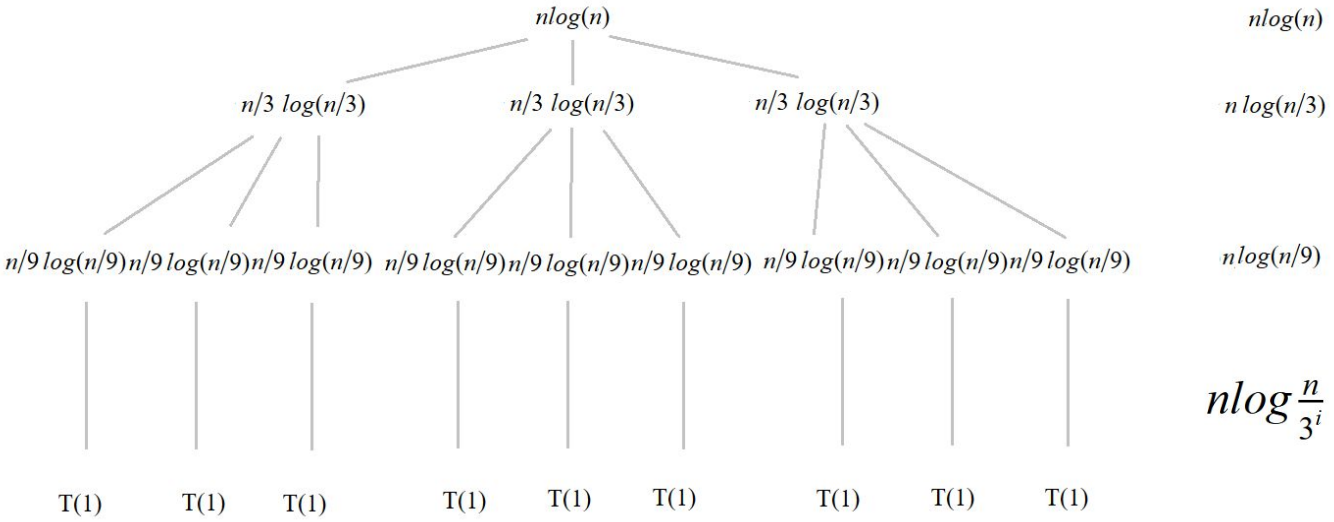
Por esto, queda demostrado por sustitución y la complejidad está dada por $\Theta(n^2)$

b) T(n) = 3T(n/3) + $nlog(n)$

Para construir el árbol hacemos algunos reemplazos en la recurrencia

$$T(n/3) = 3T(n/9) + n/3 * log(n/3)$$
$$T(n/9) = 3T(n/27) + n/9 * log(n/9)$$

Con estos valores podemos construir nuestro árbol



Al construir el árbol nos damos cuenta que el costo de cada nivel puede ser expresado como

$$nlog\frac{n}{3^i}$$

Ahora, para calcular la altura nosotros sabemos que el árbol llegara a su altura máxima en la rama que avanza más lento, es decir, el peor caso del algoritmo, la cual es T(n/3), esta llegara al caso base cuando

$$n/3^i = 1 \Rightarrow n = 3^i \Rightarrow log_3 n = i$$

Entonces, podemos estimar el costo total como la suma de todos los niveles hasta que se llegue al caso base

$$\sum_{i=0}^{log_3 n} nlog_2 \frac{n}{3^i}$$

Entonces, procedemos a resolver dicha sumatoria

$$\sum_{i=0}^{log_3 n} nlog_2 \frac{n}{3^i} = n \sum_{i=0}^{log_3 n} log_2 \frac{n}{3^i} = n \sum_{i=0}^{log_3 n} log_2 n - log 3^i$$

Luego, separando las sumatorias y resolviendo obtenemos

$$n(log_2 n * log_3 n - \sum_{i=0}^{log_3 n} i * log_2 3) = n(log_2 n * log_3 n - log_2 3 \sum_{i=0}^{log_3 n} i)$$

$$n(log_2 n * log_3 n - log_2 3((log_3 n(log_3 n + 1))/2))$$

Factorizando $log_3 n$

$$nlog_3 n(log_2 n - log_2 3((log_3 n + 1)/2))$$

Como estamos trabajando con notación asintótica, podemos aproximar que todos los logaritmos tienen la misma base

$$nlog\ n(log\ n - log\ 3((log\ n + 1)/2))$$

Expandiendo (log n + 1)

$$nlog\ n(log\ n - log\ 3 * log\ n * 1/2\ -\ log\ 3 * 1/2)$$

$$nlog\ n(log\ n(1 - log\ 3 * 1/2)\ -\ log\ 3 * 1/2)$$

Podemos omitir $log\ 3 * 1/2$ para nuestro análisis porque es una constante, así podemos factorizar juntar los log n para ver el resultado de manera más clara

$$nlog^2\ n(1 - log\ 3 * 1/2) \leq\ nlog^2(n) * \mathrm{c}$$

Nosotros sabemos que nuestra ecuación va a ser siempre menor a $nlog^2 n * c$ con c > 1. Luego, por lo anterior nuestra idea de complejidad está dada por

$$c * nlog^2\ n\ \in \Theta(nlog^2 n)$$

Ahora usaremos esta idea de complejidad para hacer la demostración por sustitución.
Luego, por intentamos demostrar esta complejidad utilizando sustitución.
Asumimos T(1) = C

Como hipótesis, tenemos que nuestra complejidad asintótica está dada por $nlog^2n$

Demostramos usando principio de inducción matemática.

Caso base: n = 1.

T(1) = 1* $log^2 1$ = 0 = C. Lo cual cumple.

Paso inductivo:

$$
\begin{aligned}
T(n) &= 3T(n/3) + nlog(n) \\
T(n) &= 3 * n/3 * log^2(n/3) + nlog(n) \qquad \text{//Por HIP de inducción} \\
T(n) &= n * (log\ n - log\ 3)^2 + nlog(n) \\
T(n) &= n * (log^2 n - 2\ log\ n\ log\ 3 + log^2 3) + nlog(n) \\
T(n) &= nlog^2 n - 2\ nlog\ n\ log\ 3 + nlog^2 3 + nlog(n)
\end{aligned}
$$

Como $nlog^2n$ domina la ecuación podemos decir que es de orden $nlog^2n$. Por esto, queda demostrado por sustitución y la complejidad está dada por $\Theta(nlog^2n)$

5) Resuelva las siguientes recurrencias usando el método de substitución:

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

En primer lugar, se asume T(1) = C con C ∈ R

Luego, proponemos que T(n) $\in \Theta(n^2)$ debido que nos da esta complejidad usando el Teorema del Maestro con f(n) = 100 n, a = 4 y b = 2 usando el primer caso con ε = 1.

Luego, por inducción:

Base: n = 1

$T(n) = 1^2 = 1$, lo cual cumple con ser una constante

Paso inductivo:

$T(n) = 4T(\frac{n}{2}) + 100n$

$=> T(n) = 4(\frac{n^2}{4}) + 100n$ // Sustituyendo la complejidad

$=> T(n) = n^2 + 100n$

Notamos que por lo anterior $T(n) \in \Theta(n^2)$, podemos afirmarlo con Teorema del límite:

$$\lim_{n\to\infty}\left(\frac{(n^2+100n)}{n^2}\right) = 1$$

Ahora, por ser $\Theta(n^2)$, demostraremos sus cotas superior e inferior:

Cota superior:

$T(n) \leq 4T(n/2) + 100n \leq 4T(n/2) + 100n^2$

$=> T(n) = 4(d\frac{n^2}{4}) + 100n^2$ // Sustituyendo T(n/2) por la complejidad postulada, d ∈ R

$=> T(n) = dn^2 + 100n^2$

Con cualquier d > 0, nos da que es cota superior. Por lo tanto queda demostrado.

Cota inferior:

$4T(n/2) + 100n \leq T(n)$

$4(d * \frac{n^2}{4}) + 100n \leq T(n)$

$dn^2 + 100n \leq T(n)$

Con 0 < d < 1, se cumple que es cota inferior.

Por lo que queda demostrado que $T(n) \in \Theta(n^2)$

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

Para T(n) asumimos que su complejidad es $\Theta(log^5(n))$ ya que es el mismo ejercicio hecho en 2.d resuelto por Teorema del Maestro.

Asumimos que T(1) = C.

Por substitución:

Como base: n = 1, $log^5(1) = 0 = C = T(1)$. Por lo que se cumple.

Paso inductivo:

$T(n) = 4T(\sqrt{n}) + log^5(n)$
$\Rightarrow T(n) = 4(log^5(\sqrt{n})) + log^5(n)$ // Sustituyendo $T(\sqrt{n})$ por la complejidad
$\Rightarrow T(n) = 4(\frac{log^5(n)}{2^5}) + log^5(n)$ // $log^5(\sqrt{n}) = log^5(n)/2^5$
$\Rightarrow T(n) = log^5(n)/8 + log^5(n)$ // 4/32 = 1/8
$\Rightarrow T(n) = (\frac{9}{8})log^5(n)$ // x/8 + 1 = 9x/8 (x = log^5(n))

Notamos que $T(n) = (\frac{9}{8})log^5(n) = d * log^5(n)$ siendo d una constante.

Por lo que se cumpliría nuestra hipótesis de que T(n) es $\Theta(log^5(n))$

Pero como T(n) es $\Theta(log^5(n))$ demostraremos sus cotas superior e inferior.

Cota superior:

$T(n) \leq 4T(\sqrt{n}) + log^5(n)$
$\Rightarrow T(n) \leq 4(d * log^5(\sqrt{n})) + log^5(n)$ // Reemplazando $T(\sqrt{n})$ por $dlog^5(\sqrt{n})$
$\Rightarrow T(n) \leq (d/8) * log^5(n) + log^5(n)$ // $log^5(\sqrt{n}) = log^5(n)/32$
$\Rightarrow T(n) \leq ((d+8)/8) * log^5(n)$
De aquí obtenemos que $(d+8)/8 > 4 \rightarrow$ d > 24
Con esto se demuestra la cota superior.
Cota inferior:

$4T(\sqrt{n}) + log^5(n) \leq T(n)$
$\Rightarrow 4(d * log^5(\sqrt{n})) + log^5(n) \leq T(n)$ // Reemplazando $T(\sqrt{n})$ por $dlog^5(\sqrt{n})$

$\Rightarrow (d/8) * log^5(n) + log^5(n) \leq T(n)$ // $log^5(\sqrt{n}) = log^5(n)/32$
$\Rightarrow ((d+8)/8) * log^5(n) \leq T(n)$
De aquí obtenemos que 4 > (d+8)/8 → d < 24
Con esto se demuestra la cota inferior.

Por lo tanto, queda demostrado que T(n) es $\Theta(log^5(n))$

6) Proporcione un análisis asintótico de peor caso en notación O() para el tiempo de ejecución de los siguientes fragmentos de programa.

a) Se adjunta imagen del código y análisis:

```
int F(int x, int y){
  int m = 1;          // 1

  if(y == 0)          // 1
    return 1;         // 1          (No siempre se entra)

  while(y > 0){       // log(y)     Mientras y > 0

    if(y % 2)         // 1          Si Y es impar
      m += x;         // 1

    x += x;           // 1          Duplicar x
    y /= 2;           // 1          Dividir y
  }//End While

  return m;           // 1

}//End F
```

Teniendo esto, si sumamos todas las operaciones sin contar el return 1 debido a que no es el peor caso, nos queda:

$$f(n) = 4\,log(n) + 3$$

Esto es porque estamos entrando en el ciclo while y debemos ir dividiendo y por 2 sucesivamente hasta que y > 0 no se cumpla. Como vamos dividiendo por 2, esto es logarítmico.

Podemos postular que f(n) es $O(\log(n))$, lo cual comprobaremos con Teorema del límite con g(n) = log(n).

$$\lim_{n\to\infty}\left(\frac{4log(n)+3}{log(n)}\right)=4$$

Como el resultado del límite es 4, nos dice que es $\Theta(\log(n))$, pero como es análisis de peor caso, decimos que es O(log(n)) ya que el mejor caso es constante cuando y = 0.

Por lo tanto, f(n) es O(log(n)).

b) Se adjunta una imagen del código y el análisis:

```
void procesar(int i);

int recurse(int n){
  for(int i = 0; i < n*n; i += 2)   // (n^2) / 2
    procesar(i);                    // O(n)= c*n
  // Hasta aqui es (c/2)n^3

  if(n <= 0)      // 1
    return 1;
  else
    return recurse(n - 3);    // T(n - 3)

}//End Recurse
```

Si sumamos todas las operaciones del análisis anterior tendríamos:

$$T(n) = T(n-3) + \tfrac{c}{2}n^3$$

Pero el término $\frac{c}{2}n^3$ podemos escribirlo como $O(n^3)$ lo cual es muy fácil de ver.
Por lo que hasta ahora tenemos:

$$T(n) = T(n-3) + O(n^3)$$

Luego hacemos unas cuantas iteraciones:

$T(n-3) = T(n-9) + O((n-3)^3)$ // $O((n-3)^3) = O(n^3)$ por notación asintótica.

$= T(n-9) + O(n^3)$

Reemplazando T(n-3) en T(n) nos queda:

$T(n) = T(n-9) + O(n^3) + O(n^3)$

$= T(n-9) + 2O(n^3)$

Luego desarrollamos T(n-9):

$T(n-9) = T(n-12) + O((n+3)^3)$ // $O((n-3)^3) = O(n^3)$

$= T(n-12) + O(n^3)$

Reemplazamos T(n-9) en T(n) y nos queda:

$T(n) = T(n-12) + O(n^3) + 2O(n^3)$

$= T(n-12) + 3O(n^3)$

Hacemos una última iteración:

$T(n-12) = T(n-15) + O((n-3)^3)$ // $O((n-3)^3) = O(n^3)$

$= T(n-15) + O(n^3)$

Reemplazamos T(n-12) en T(n) y nos queda:

$T(n) = T(n-15) + O(n^3) + 3O(n^3)$

$= T(n-15) + 4O(n^3)$

Hasta ahora tenemos:

$T(n) = T(n-3) + O(n^3)$

$T(n) = T(n-6) + 2O(n^3)$

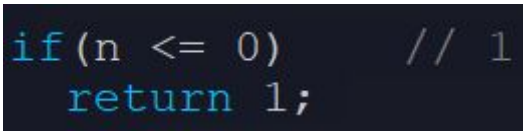
$T(n) = T(n-9) + 3O(n^3)$

$T(n) = T(n-12) + 4O(n^3)$

Con esto podemos apreciar que T(n) sigue la fórmula:

$$T(n) = T(n-3k) + kO(n^3)$$

Y con el caso base del código:



Tenemos que T(0) = 1. Igualamos lo siguiente:

$$T(n-3k) = T(0) = 1$$
$$\Rightarrow n - 3k = 0$$
$$\Rightarrow k = \tfrac{n}{3}$$

Reemplazamos k en la fórmula postulada:

$$T(n) = T(n-3n/3) + \tfrac{n}{3}O(n^3)$$
$$\Rightarrow T(n) = T(n-n) + \tfrac{n}{3}O(n^3)$$
$$\Rightarrow T(n) = T(0) + \tfrac{O(n^4)}{3} \;//\; nO(n^3) = O(n^4)$$
$$\Rightarrow T(n) = 1 + O(n^4)$$

Donde vemos claramente que $T(n) \in O(n^4)$.

Por lo tanto $T(n)$ *es* $O(n^4)$.

7) Para el problema postulado:

a) Escriba un pseudocódigo para un algoritmo O(n^2) que resuelva el problema:

```
Function: n2(VectorOfPairs vec)                    // Recibe un vector de pares

        count <- size.vec()                        // guardar en "count" el tamaño del vector

        IF count equals 0                          //caso en que el vector esté vacío
            RETURN 0

       // Ordenar puntos de menor a mayor
        FOR i <- 0 to (count - 1) DO        // SELECTION SORT O(n^2) para ordenar pares
          FOR j <- i (to count -1) DO

            IF vec(j).first <= vec(i).first and vec(j).second < vec(i).second  THEN
                aux <- vec(j)
                vec(j) <- vec(i)
                vec(i) <- aux
             END IF

            ELSE IF vec(j).first < vec(i).first THEN
                aux <- vec(j)
                vec(j) <- vec(i)
                vec(i) <- aux
            END ELSE IF

          END FOR
        END FOR

      // END SELECTION SORT

        max <- vec(0).second
        flicks <- 1

        FOR i <- 1 to (count - 1) DO                    // O(n)
            IF vec(i-1).second <= vec(i).first AND vec(i).first >= max THEN
              flicks <- flicks + 1
              max <- vec(i).second
```

```
            ELSE IF vec(i).second > max  THEN
                max <- vec(i).second

        END FOR

        RETURN flicks

      END N2
```

b) Diseñe un algoritmo que resuelva el problema O(nlog(n)) y escriba su pseudo código.

```
Function: nlogn(VectorOfPairs vec)          // Recibe un vector de pares con los puntos

        count <- vec.size()                 // guardar en "count" el tamaño del vector

        IF count equals 0                       //caso en que el vector este vacio
            RETURN 0

        // Ordenar puntos de menor a mayor

        sort(vec.begin(), vec.end())    // Sort de C++ es O(nlog(n))

        max <- vec(0).second
        flicks <- 1

        FOR i = 1 to (count -1) DO     // Esto es O(n)
          IF vec(i-1).second <= vec(i).first AND vec(i).first >= max   THEN
            flicks <- flicks + 1
              max <- vec(i).second

          ELSE IF vec(i).second > max  THEN
            max <-  vec(i).second

        END FOR
        RETURN  flicks

      END NLOGN
```

c) En nuestros códigos, seguramente es evidente darse cuenta que la complejidad temporal depende de el tipo de sort que sea usado. De hecho, los 2 códigos que resuelven el problema, me refiero al que es O(nlogn) y al que es O(n^2) son exactamente iguales excepto por el tipo de sort que se utiliza.

Por esto, nuestros códigos, nosotros asumimos la correctitud de los sort que utilizamos, que en el caso del código O(n^2) es el Selection Sort implementado por nosotros y en el caso del O($nlog\ n$) es el std::sort() de C++. Estos 2 algoritmos son bien conocidos ya y es claro que son correctos por lo que nos centraremos en ellos.

Por esto, solo nos interesa demostrar la correctitud de nuestro código en particular para resolver el problema. Aquí el pseudocódigo una vez más.

```
count <- vec.size()                  // guardar en "count" el tamaño del vector

IF count equals 0                    //caso en que el vector esté vacío
      RETURN 0

// Ordenar puntos de menor a mayor

sort(vec.begin(), vec.end())    // Sort de C++ es O(nlog(n))

max <- vec(0).second
flicks <- 1

FOR i = 1 to (count -1) DO     // Esto es O(n)
  IF vec(i-1).second <= vec(i).first AND vec(i).first >= max   THEN
    flicks <- flicks + 1
       max <- vec(i).second

  ELSE IF vec(i).second > max  THEN
    max <-  vec(i).second
```

Notamos que en el caso que el vector esté vacío, este va a retornar el valor entero 0, puesto que no va a ocurrir ninguna instancia en que se prenda la luz.

En caso que tenga elementos, hacemos un sort (ya sea std::sort() de C++ o nuestro Selection Sort) para ordenar los pares de nuestro vector.

El hecho que el vector esté ordenado es una precondición para ejecutar nuestro loop.

Notemos que nuestro algoritmo posee 2 variables, “max” y “flicks”. Max registra el minuto más tardío conocido hasta el momento en que una persona aún estará en la pieza. Por ejemplo, los tiempos de entrada y salida de una persona está definida por (2,6), entonces si se ha revisado el vector hasta ese punto, max corresponde a 6 sólo si ninguno de los asistentes ha marcado un minuto de salida mayor, siendo hasta ahora el minuto de salida conocido más tardío y el momento en que la sala debería quedar vacía.

Flicks corresponde a la cantidad de veces que se ha prendido la luz.

La primera persona (x_i, y_i) va a permanecer en la pieza hasta el minuto y_i. Entonces, la primera persona que entra en el instante x_o va a permanecer en la pieza hasta el tiempo y_o, por esto, registramos max = y_o, puesto que hasta el minuto y_0 no tenemos porque preocuparnos, ya que tenemos la certeza de que la luz va a estar encendida.

Entonces, cuando terminamos de revisar la persona (x_0, y_0) vamos a tener registrado que max = y_0. Vemos la persona (x_1, y_1), si el tiempo en que entra, es decir x_1 es mayor o igual a max, entonces la luz va a tener que ser prendida. En caso que x_1 haya sido efectivamente mayor o igual a max, tendremos que actualizar max y aumentar flicks en 1. En caso que x_1 no sea mayor o igual a max sabemos que no fue necesario prender la luz, puesto que aún había una persona en la pieza y en caso que y_1 sea mayor que max, lo actualizamos, ya que esto nos está diciendo que la persona (x_1, y_1) va a permanecer en la pieza aún cuando la persona (x_0, y_0) se vaya. Luego se ve la persona (x_2, y_2) y se vuelve a verificar sigue así hasta (x_{n-1}, y_{n-1}).

Ahora que tenemos clara la idea del algoritmo y la intuición de él, vamos a formalizar la correctitud usando un invariante de loop.

Como precondición tenemos que nuestro vector de pares está ordenado de menor a mayor. Proponemos como invariante que max va a representar el tiempo y_i más alto conocido hasta el momento y que si encontramos un x_{i+1} mayor que él sabemos que tendremos que prender la luz, sumando 1 a flicks y actualizar max con y_{i+1}.

Inicialización: Comenzamos en la posición $i = 1$ del vector, con $max = y_0$ y $flicks = 1$, puesto que esta va a ser la única persona que ha estado en la pieza hasta este momento y necesariamente va a ser max y necesariamente tuvo que prender la luz.

Mantención: Cuando estemos analizando un elemento i del vector nosotros tenemos la certeza que max va a haber sido asignado correctamente y tiene el tiempo más alto conocido hasta el momento y se sabe que el invariante es verdad.
Al revisar el elemento (x_i, y_i) vemos que si x_i es mayor o igual a que max, entonces se asigna y_i a max, puesto que este y_i va a representar el tiempo más alto conocido hasta este

momento lo cual se cumple debido a que $x_i < y_i$,luego también aumentamos flicks en 1. Si x_i no era mayor o igual a max, entonces verificamos si y_i es mayor que max. Si lo es, actualizamos max con y_i puesto que este será el tiempo más alto conocido hasta este momento. Con esto, hemos terminado la iteración i, y aun es verdad nuestro invariante, ya que max aun posee el valor de salida más alto conocido hasta el momento.

Término: El loop ha terminado puesto que revisamos todos los elementos de el y estamos en "count-1". En flicks tendremos la cantidad de veces que ha sido prendida la luz, mientras que en max estara valor y_i más alto conocido del vector, con lo cual se cumple el invariante.

d) Implemente sus algoritmos usando C++ definiendo una función para cada uno.

Código adjunto al pdf.

e) Realice análisis experimental para lo cual se pide que construya un gráfico que muestre cómo varían los tiempos de ejecución en nanosegundos variando el tamaño de la entrada (n).

En primer lugar, para obtener los inputs, se creó un código en C++ llamado "gen.cpp" el cual está adjunto a este pdf y que al ejecutarlo nos pedirá un "n" retornando n pares (x,y) con $1 \leq x \leq 9.999 \ \wedge x < y \leq 10.000$.

Los pares son generados aleatoriamente y no están en orden.

Ahora, con "gen.cpp" creamos varias entradas, por cada "n" creamos 5 inputs y promediamos sus tiempos de ejecución. Con "n" entre 1.000 y 10.000 entre espacios de 1.000 armamos la siguiente tabla:

n	n log n [ns]	n2 [ns]
1000	451942	10054083
2000	1402941	49294148
3000	1849132	118975362
4000	2132108	162859768
5000	2422575	252586620
6000	2894468	360879556
7000	2477477	492249944
8000	3750712	644817341
9000	4120321	804219730
10000	4598210	993170537

Tabla 1: Tiempos de ejecución en nanosegundos para los algoritmos "nlogn" y "n2" promedio para cada n.

Con estos datos, generamos los siguientes gráficos:

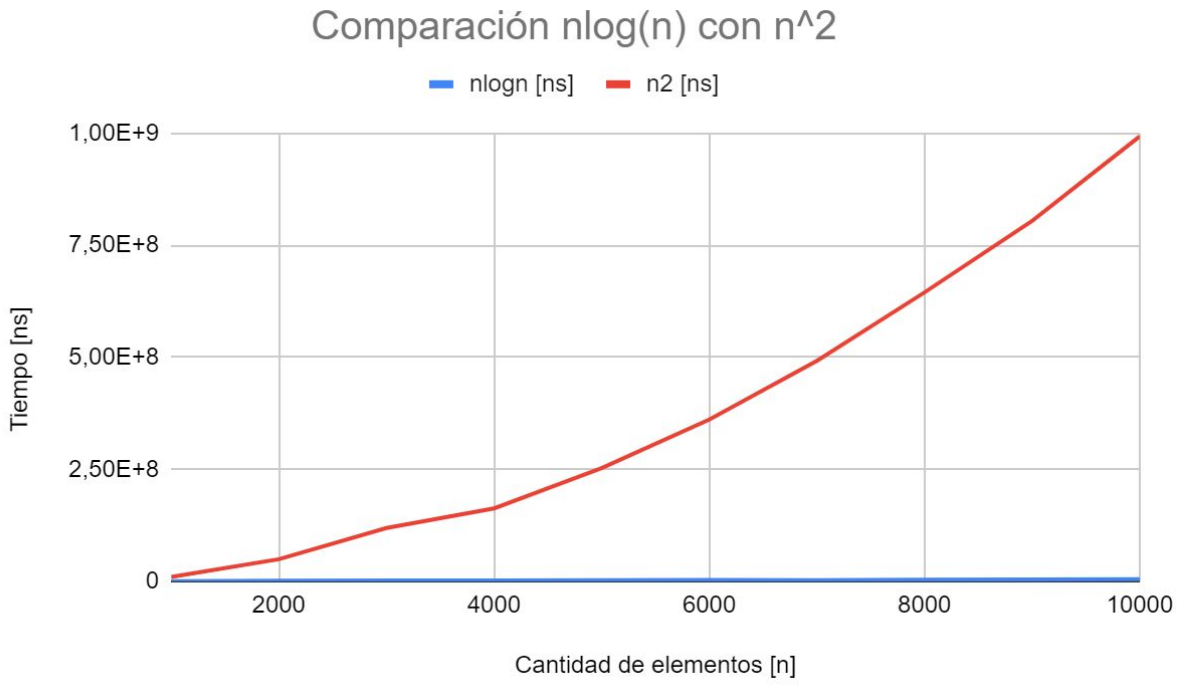


Gráfico 1: Comparación de tiempos de ejecución de los algoritmos presentados.

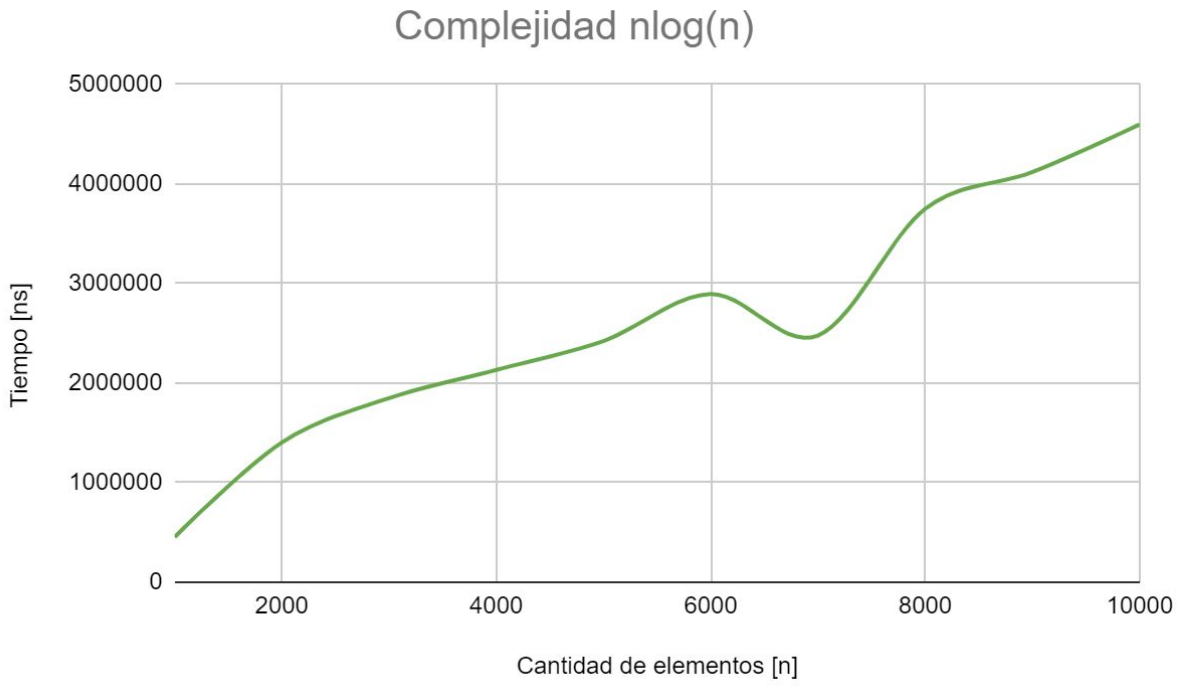


Gráfico 2: Complejidad del algoritmo “nlogn”

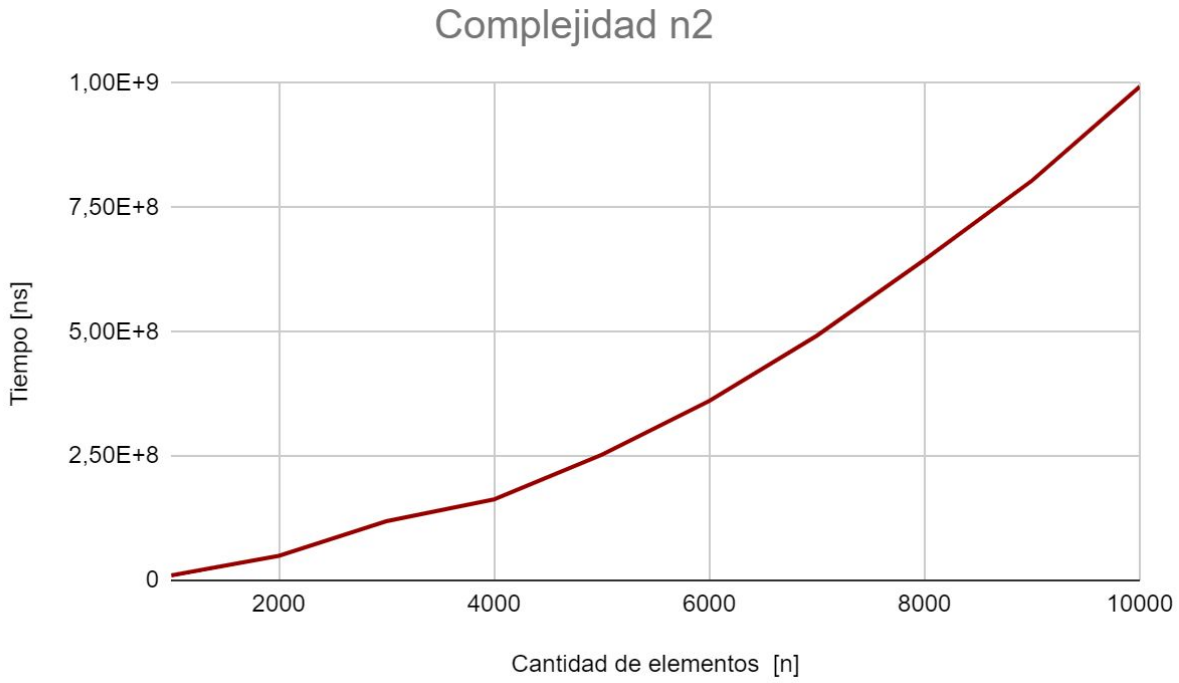


Gráfico 3: Complejidad del algoritmo “n2”

Nota: Ambas funciones fueron graficadas por separado debido a que la curva de “n2” era demasiado grande como para observar bien la curva de “nlogn”.

En Gráfico 1 se observa que el tiempo de ejecución de “nlogn” es mucho menor al de la función “n2”, tanto así que ni siquiera podemos ver la forma que tiene ya que la función roja tiene valores en el eje Y mucho mayores. Para analizarlos más detalladamente se graficaron los tiempos de ejecución de los algoritmos por separado.

Para el gráfico 2, podemos observar que es una curva creciente, más suave que una curva $f(n) = n^2$ pero mayor que una curva $g(n) = n$ y tiende a tener la forma de una curva $h(n) = nlog(n)$. Por lo que podemos concluir que el algoritmo “nlogn” tiene la complejidad postulada de O(nlog(n)).

Para el gráfico 3, se aprecia claramente que cumple con la forma de una función $f(n) = n^2$. Por lo que podemos concluir que el algoritmo “n2” tiene complejidad O(n^2) calzando con lo postulado en un inicio.