Proyecto 1:

Análisis de Algoritmos

Primer Semestre, Prof. Cecilia Hernández

Nicolas Araya y Martina Cádiz

PREGUNTA 1

a) $4^n \in w(4^{n/2})$

Consideramos g(n) $= 4^{n/2}$ y f(n) $= 4^n$

Esto es verdadero, ya que si $n \to \infty$ el valor de g(n) incrementa en menor medida que el de f(n). Por ende,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4^n}{4^{n/2}} = \lim_{n\to\infty} 4^{n-n/2} = \infty$$

Luego utilizando la definición:

$$w(g(n)) = \{f(n): para\ toda\ constante\ c > 0, existe\ n_o\ tal\ que\ 0 \leq c \cdot g(n) < f(n), \\ \forall\ n \geq n_o\}$$

Tenemos la desigualdad:

$$c \cdot 4^{n/2} < 4^n$$

$$\Leftrightarrow\ c < 4^{n-n/2}$$

$$\Leftrightarrow\ c < 4^{n/2}$$

Si asumimos $n_o = 1$, luego $c < 2$ por lo que podemos tomar $c = 1$ y $n_o = 1$ y así se cumple la desigualdad $0 \leq c \cdot 4^{n/2} < 4^n$.

b) $5\ \log_4(\log(\log(n^{100}))) \in O(\log\log(n))$

Consideramos g(n) $= O(\log\log(n))$ y f(n) $= 5\ \log_4(\log(\log(n^{100})))$

Utilizamos la definición dada por:

$$O(g(n)) = \{(f(n): existen\ las\ constantes\ positivas\ c\ y\ n_0\ tal\ que\ 0 \leq f(n) \\ \leq cg(n)\ para\ todo\ n \geq n_0\}$$

Desarrollamos la desigualdad:

$$5\log_4(\log(\log(n^{100}))) \leq c \cdot \log(\log(n))$$

$$\Leftrightarrow \log_4(\log(\log(n^{100}))) \leq \frac{c}{5} \cdot \log(\log(n))$$

$$\Leftrightarrow \log_4(\log(\log(n^{100}))) \leq \log(\log^{\frac{c}{5}}(n))$$

$$aplicando\ propiedad\ de\ cambio\ de\ base:$$

$$\Leftrightarrow \frac{\log(\log(\log(n^{100})))}{\log(4)} \leq \log(\log^{\frac{c}{5}}(n))$$

$$\Leftrightarrow \log(\log(\log(n^{100}))) \leq 2 \cdot \log(\log^{\frac{c}{5}}(n))$$

$$\Leftrightarrow \log(\log(\log(n^{100}))) \leq \log(\log^{\frac{2c}{5}}(n))$$

$$aplicando\ 2^{(\ )}:$$

$$\Leftrightarrow 2^{\log(\log(\log(n^{100})))} \leq 2^{\log(\log^{\frac{2c}{5}}(n))}$$

$$\Leftrightarrow \log(\log(n^{100}) \leq \log^{\frac{2c}{5}}(n)$$

Ahora considerando $c = \frac{5}{2}, nos\ queda$:

$$\log(\log(n^{100}) \leq \log(n)$$

$$aplicando\ 2^{(\ )}:$$

$$\log(n^{100}) \leq n$$

$$100\log(n) \leq n$$

Esto tiene solución ya que $O(\log(n)) \in O(n)$, lo que solo nos falta encontrar n que satisface la inecuación.

Esto se cumple con $n = 2^{10}$

$$100\log(2^{10}) \leq 2^{10}$$

$$100 * 10 \leq 1024$$

$$1000 \leq 1024$$

Luego queda demostrado que:

$$5\log_4(\log(\log(n^{100}))) \in O(\log\log(n))\ con\ c = \frac{5}{2}\ y\ n = 2^{10}$$

c) $4^{\log(n)} \in o(n^2)$

Consideramos $f(n) = 4^{\log(n)} = 2^{2\cdot\log(n)} = n^2$ y $g(n) = n^2$

Esta afirmación es falsa, ya que ambas funciones crecen (se incrementan) a igual medida. De hecho, si utilizamos el método del límite podemos ver que no cumple:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n^2} = 1 \quad \neq 0$$

Luego, para mostrar que esto no se cumple, basta encontrar un contraejemplo. Siguiendo la definición:

$$o(g(n)) = \{f(n): para\ toda\ constante\ c > 0, existe\ n_o\ tal\ que\ 0 \leq f(n) < c \cdot g(n), \\ \forall\ n \geq n_o\}$$

Probamos con un c = 1, por lo que nuestra desigualdad queda:

$$0 \leq n^2 < 1 \cdot n^2 \text{ /dividiendo por } n^2$$

$$\Leftrightarrow 0 \leq 1 < 1$$

Luego $1 < 1$ es falso, ya que ambos valores son iguales. Por lo que queda demostrado utilizando un contra ejemplo, con c=1 y con cualquier valor de n_0 (por ejemplo $n_0 = 4$), que la afirmación es falsa.

d) $2n - 2\sqrt{n} \in \theta(n)$

Consideramos $f(n) = 2n - 2\sqrt{n}$ y $g(n) = n$

Se puede observar que un n suficientemente grande en f(n), 2n domina la función. Por lo tanto, se debe hallar c_1 , c_2 y n_0 que satisfagan la definición:

$$0 < c_1 \cdot n \leq 2n - 2\sqrt{n} \leq c_2 \cdot n \text{ /multiplicando por } \frac{1}{n}$$

$$\Leftrightarrow 0 < c_1 \leq 2 - 2n^{\frac{1}{2}-1} \leq c_2$$

Analizando la desigualdad de la izquierda:

$$0 < c_1 \leq 2 - 2n^{-\frac{1}{2}}$$

$$\Leftrightarrow 0 < 2 - 2n^{-\frac{1}{2}}$$

$$\Leftrightarrow 0 < 2 - 2n^{-\frac{1}{2}} \text{ / dividiendo por 2}$$

$$\Leftrightarrow 1 > n^{-\frac{1}{2}} \text{ / elevando al cuadrado}$$

$$\Leftrightarrow n > 1$$

Como n > 1, podemos tomar $n_0 = 2$ y encontrar el valor de c_1 :

$$c_1 \leq 2 - \frac{2}{\sqrt{2}}$$

Por lo tanto, podemos tomar $c_1 = 2 - \frac{2}{\sqrt{2}}$, ahora analizando la desigualdad hacia la derecha:

$$2 - 2n^{-\frac{1}{2}} \leq c_2$$

En este caso sabemos que $n \to \infty$, por lo que $\frac{2}{\sqrt{n}} \to 0$ podemos tomar $c_2 = 2$.

Entonces podemos concluir que la afirmación es verdadera, ya que para $c_1 = 2 - \frac{2}{\sqrt{2}}$, $c_2 = 2$ y $n_0 = 2$ se cumple que $f(n) \in \theta(n)$

e) Si $f(n) \in \theta\big(g(n)\big)$ entonces $g(n) \in \theta\big(f(n)\big)$

Esta afirmación es verdadera, ya que se puede desprender de la propiedad de simetría.

Tenemos que:

$$f(n) \in O\big(g(n)\big) \Rightarrow c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad (1)$$

$$g(n) \in O\big(f(n)\big) \Rightarrow d_1 f(n) \leq g(n) \leq d_2 f(n) \qquad (2)$$

Dividiendo por $c_1$en (1), considerando que $c_1 \neq 0$

$$g(n) \leq \frac{f(n)}{c_1} \leq \frac{c_2}{c_1} g(n)$$

$$\Leftrightarrow d_1 f(n) \leq g(n) \leq \frac{f(n)}{c_1} \leq \frac{c_2}{c_1} g(n) \qquad (3)$$

Y ahora dividiendo por c_2 en (1), considerando $c_2 \neq 0$

$$\frac{c_1}{c_2} g(n) \leq \frac{f(n)}{c_2} \leq g(n)$$

$$\Leftrightarrow \frac{c_1}{c_2} g(n) \leq \frac{f(n)}{c_2} \leq g(n) \leq d_2 f(n) \qquad (4)$$

De (3) y (4) tenemos:

$$d_1 f(n) \leq g(n) \leq \frac{f(n)}{c_1}$$

y

$$\frac{f(n)}{c_2} \leq g(n) \leq d_2 f(n)$$

De aquí podemos asumir que:

$$d_1 = \frac{1}{c_2} \quad y \quad d_2 = \frac{1}{c_1}$$

$$d_1 \cdot c_2 = 1 \quad \Rightarrow \quad d_1 = 1, \qquad c_2 = 1$$

$$d_2 \cdot c_1 = 1 \quad \Rightarrow \quad d_2 = 1\,, \qquad c_1 = 1$$

Luego tomando $c_1 = 1,\ c_2 = 1\,,\ d_1 = 1\,,\ d_2 = 1$ y reemplazando en (1) y (2)

$$g(n) \leq f(n) \leq g(n)$$

$$f(n) \leq g(n) \leq f(n)$$

Lo cual se cumple si y solo si $f(n) = g(n)$

Luego por reflexividad:

$$f(n) = O\big(f(n)\big)$$

Y como $f(n) = g(n)$

$$f(n) = O\big(g(n)\big), \qquad con\, f(n) = g(n)$$

Con lo que queda demostrado.

PREGUNTA 2

Para ordenar las funciones, primero las desarrollamos un poco:

a) $n \cdot \sqrt[3]{n}\ :\ n^{\frac{1}{3}+1}\ = n^{\frac{4}{3}} \in O(n^{\frac{4}{3}})$

b) $4000^{5^{1000000}}$: Al no depender de una variable es O(c), con c una cte

c) $3^{n \cdot 0{,}001}$: Como n se encuentra en el exponente de una constante, se tiene una complejidad exponencial $\in O(3^n)$

d) $4^{\log(n)}\ = 2^{2 \cdot \log(n)} = n^2\ \in O(n^2)$

De esta manera, tenemos ya el análisis asintótico de cada uno, solo queda ordenarlo. Partiendo desde el mayor tenemos la complejidad exponencial, es la mayor del grupo. Luego comparando entre $O(n^2)$ y $O(n^{\frac{4}{3}})$ tenemos que $O(n^2)$ domina a la otra ya que posee un exponente mayor. Finalmente nos restaría la complejidad constante

Ordenando de menor a mayor nos quedaría:

$$O(c) < O\left(n^{\frac{4}{3}}\right) <\ O(n^2) < O(3^n)$$

$$b) < a) < d) < c)$$

PREGUNTA 3

a) $T(n) = 3T(n/2) + cn$

Utilizaremos el teorema del maestro, ya que $a \geq 1$, $b > 1$ y consideramos que c es una constante entonces $f(n) = c \cdot n$ es una función asintóticamente positiva.

Luego tenemos que $a = 3$, $b = 2$ y $d = \log_2 3$.

Comparamos f(n) con d y analizando el siguiente caso, donde buscamos que $f(n) = O(n^{\log_2 3 - \varepsilon})$ para una constante $\varepsilon > 0$. Es decir, debemos encontrar un caso donde se cumpla $\log_2 3 - \varepsilon = 1$.

Podemos ver que si existe un valor para ε, ya que el valor de $d \approx 1.58$, por lo que con un $\varepsilon > 0$ es posible obtener el valor 1.

Finalmente, gracias al teorema del maestro, tenemos que $T(n) = O(n^{\log_2 3})$

b) $T(n) = 7T(n/2) + cn^2$

Utilizaremos el teorema del maestro, ya que $a \geq 1$, $b > 1$ y consideramos que c es una constante, entonces $f(n) = c \cdot n^2$ es una función asintóticamente positiva.

Luego tenemos que $a = 7$, $b = 2$ y $d = \log_2 7$.

Comparamos f(n) con d y analizando el siguiente caso, donde buscamos que $f(n) = O(n^{\log_2 7 - \varepsilon})$ para una constante $\varepsilon > 0$. Es decir, debemos encontrar un caso donde se cumpla $\log_2 7 - \varepsilon = 2$.

Podemos ver que si existe un valor para ε, ya que el valor de $d \approx 2.8$, por lo que con un $\varepsilon > 0$ es posible obtener el valor 2.

Finalmente, gracias al teorema del maestro, tenemos que $T(n) = O(n^{\log_2 7})$

c) $T(n) = 3T(2\frac{n}{3}) + cn$

Utilizaremos el teorema del maestro, ya que $a \geq 1$, $b > 1$ y consideramos que c es una constante, entonces $f(n) = c \cdot n$ es una función asintóticamente positiva.

Luego tenemos que $a = 3$, $b = \frac{3}{2}$ y $d = \log_{\frac{3}{2}} 3$.

Comparamos f(n) con d y analizando el siguiente caso, donde buscamos que $f(n) = O(n^{\log_{\frac{3}{2}} 3 - \varepsilon})$ para una constante $\varepsilon > 0$. Es decir, debemos encontrar un caso donde se cumpla $\log_{\frac{3}{2}} 3 - \varepsilon = 1$.

Podemos ver que si existe un valor para ε, ya que el valor de $d \approx 2.7$, por lo que con un $\varepsilon > 0$ es posible obtener el valor 1.

Finalmente, gracias al teorema del maestro, tenemos que $T(n) = O(n^{\log_{\frac{3}{2}} 3})$

d) $T(n) = 16T(\sqrt{n}) + \log_5(n)$

En este caso vamos a hacer cambio de variables, donde tomaremos $m = \log(n)$, por lo que $n=2^m$. Luego, reemplazando en la ecuación: $T(2^m) = 16T(2^{\frac{m}{2}}) + \log_5(2^m)$.

Consideraremos $S(m) = T(2^m)$. Luego $S(m) = 16S(m/2) + \frac{\log_2(2^m)}{\log_2(5)}$.

$$S(m) = 16S(m/2) + \frac{m}{\log_2(5)}$$

Ahora que tenemos una recurrencia que puede ser resuelta mediante el teorema del maestro, ya que $a \geq 1$, $b > 1$ y teniendo en cuenta que $\log_2(5) \approx 2.3$ (siendo este un valor constante), la función $f(m) = 2.3m$ es una función asintóticamente positiva.

Luego tenemos que $a = 16$, $b = 2$ y $d = \log_2 16$.

Comparamos f(m) con d y analizando el siguiente caso, donde buscamos que $f(m) = O(m^{\log_2 16 - \varepsilon})$ para una constante $\varepsilon > 0$. Es decir, debemos encontrar un caso donde se cumpla $\log_2 16 - \varepsilon = 1$.

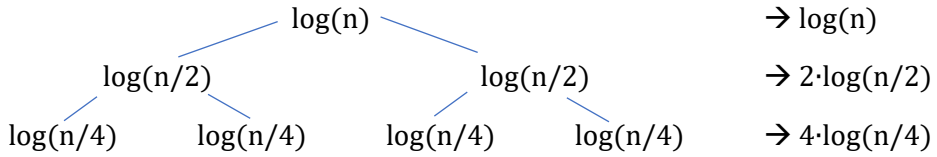
Podemos ver que si existe un valor para ε, ya que el valor de $d = 4$, por lo que con un $\varepsilon > 0$ es posible obtener el valor 1.

Luego, gracias al teorema del maestro, tenemos que $S(m) = O(m^4)$. De aquí podemos obtener el valor de $T(n) = T(2^m) = S(m) = O(log^4(n))$.

PREGUNTA 4

a) $T(n) = 2T(n/2) + \log(n)$

Dibujando el árbol de recurrencia tenemos lo siguiente:



costo

→ log(n)

→ 2·log(n/2)

→ 4·log(n/4)

Y así se expanden las ramas hasta llegar al caso base. Al utilizar el árbol recursivo consideramos:

1. El tamaño del problema de la entrada de la recurrencia en el nivel i es $\frac{n}{2^i}$
2. Si calculamos el momento en que se llega al caso base (asumiendo T=1), podemos calcular la altura: $\frac{n}{2^i} = 1$. Luego la altura dado por el nivel i es log(n)
3. El número de hojas en cualquier nivel es 2^i
4. En el nivel i hay 2^i hojas con costo de $\log(\frac{n}{2^i})$. Por lo tanto la suma de los costos al nivel i es $2^i \cdot \log(\frac{n}{2^i})$.

Con los resultados anteriores podemos calcular el costo total:

$T(n) = \log(n) + 2 \cdot \log(n/2) + 4 \cdot \log(n/4) + 8 \cdot \log(n/8) + 16 \cdot \log(n/16) + \ldots$

$$T(n) = \sum_{i=0}^{\log(n)-1} 2^i \cdot \log(\frac{n}{2^i}) + 2^{\log(n)} T(1)$$

Aplicamos propiedad de logaritmo:

$$T(n) = \sum_{i=0}^{\log(n)-1} 2^i \cdot [\log(n) - \log(2^i)] + 2^{\log(n)} T(1)$$

$$T(n) = \sum_{i=0}^{\log(n)-1} 2^i \cdot [\log(n) - i] + nT(1)$$

Aplicamos propiedad de sumatorias:

$$T(n) = \sum_{i=0}^{\log(n)-1} 2^i \cdot \log(n) - \sum_{i=0}^{\log(n)-1} 2^i \cdot i + nT(1)$$

$$T(n) = \log(n) \cdot \sum_{i=0}^{\log(n)-1} 2^i - \sum_{i=0}^{\log(n)-1} 2^i \cdot i + nT(1)$$

Resolvemos la primera sumatoria:

$$T(n) = \log(n) \cdot (-1 + 2^{\log(n)}) - \sum_{i=0}^{\log(n)-1} 2^i \cdot i + n \cdot T(1)$$

Resolvemos la segunda sumatoria:

$$T(n) = -\log(n) + n \cdot \log(n) - 2^{\log(n)} \cdot log(n) + 2(2^{\log(n)} - 1) + n \cdot T(1)$$

$$T(n) = -\log(n) + n \cdot \log(n) - n \cdot log(n) + 2 \cdot n - 2 + n \cdot T(1)$$

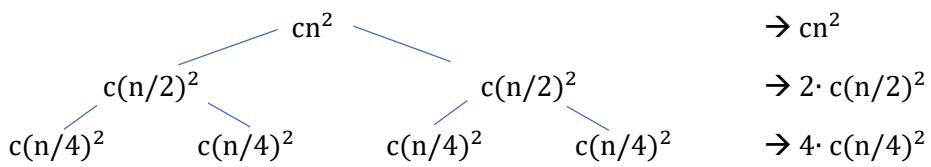
Reduciendo así T(n) a la siguiente expresión y asumiendo que $T(1) = c$, ya que es el caso base:

$$T(n) = -\log(n) - 2 + 2 \cdot n + n \cdot c$$

Finalmente podemos concluir que $2 \cdot n$ domina la función, por lo que $T(n) \in O(n)$

b) $T(n) = 2T(n/2) + cn^2$

Dibujando el árbol de recurrencia tenemos lo siguiente: costo



Y así se expanden las ramas hasta llegar al caso base. Al utilizar el árbol recursivo consideramos:

1. El tamaño del problema de la entrada de la recurrencia en el nivel i es $\frac{n}{2^i}$
2. Si calculamos el momento en que se llega al caso base (asumiendo T=1), podemos calcular la altura: $\frac{n}{2^i} = 1$. Luego la altura dado por el nivel i es log(n)
3. El número de hojas en cualquier nivel es 2^i
4. En el nivel i hay 2^i hojas con costo de $c \cdot (\frac{n}{2^i})^2$. Por lo tanto, la suma de los costos al nivel i es $2^i \cdot c \cdot (\frac{n}{2^i})^2$.

Con los resultados anteriores podemos calcular el costo total:

$T(n) = cn^2 + 2 \cdot c(n/2)^2 + 4 \cdot c(n/4)^2 + 8 \cdot c(n/8)^2 + 16 \cdot c(n/16)^2 + \ldots$

$$T(n) = \sum_{i=0}^{\log(n)-1} c \cdot (\frac{n^2}{2^{2 \cdot i}}) \cdot 2^i + 2^{\log(n)} \cdot c \cdot T(1)$$

Aplicando propiedad de sumatoria y logaritmo, además asumiendo que T(1) toma valor constante, al igual que la constante c. Podemos considerar lo siguiente:

$$T(n) = c \cdot n^2 \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2^i}\right) + n \cdot c$$

Resolviendo la sumatoria nos queda:

$$T(n) = c \cdot n^2 (2 - 2^{1-log(n)}) + n \cdot c$$

Reduciendo así T(n) a la siguiente expresión:

$$T(n) = 2 \cdot c \cdot n^2 - c \cdot n$$

Finalmente podemos concluir que $2cn^2$ domina la función, por lo que $T(n) \in O(n^2)$.

PREGUNTA 5

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

Primero para poder tener una idea de la complejidad de T(n), utilizaremos el método de expansión:

En la primera iteración (k=1) tendremos:

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

Para la segunda iteración (k=2) consideraremos que $T(n/2) = 4T(n/4) + n/2$, por lo que nos queda:

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

$T(n) = 16T(n/4) + 2n + n$

Para la tercera iteración (k=3) consideraremos que $T(n/4) = 4T(n/8) + n/4$, por lo que nos queda:

$T(n) = 16(4T(n/8) + n/4) + 2n + n$

$T(n) = 64T(n/8) + 4n + 2n + n$

Para la cuarta iteración (k=4) consideraremos que $T(n/8) = 4T(n/16) + n/8$, por lo que nos queda:

$T(n) = 64(4T(n/16) + n/8) + 4n + 2n + n$

$T(n) = 256T(n/16) + 8n + 4n + 2n + n$

Con las iteraciones realizadas, ya es posible dilucidar un patrón. En resumen, vemos lo siguiente:

$k=1, T(n) = 4T(n/2) + n = 4^1T(n/2^1) + (2^0)n$

$k=2, T(n) = 16T(n/4) + 2n + n = 4^2T(n/2^2) + (2^1 + 2^0)n$

$k=3, T(n) = 64T(n/8) + 4n + 2n + n = 4^3T(n/2^3) + (2^2 + 2^1 + 2^0)n$

$k=4, T(n) = 256T(n/16) + 8n + 4n + 2n + n = 4^4T(n/2^4) + (2^3 + 2^2 + 2^1 + 2^0)n$

El patrón que sigue sería:

$$T(n) = 4^kT(n/2^k) + \sum_{i=0}^{k-1} 2^i n$$

Cuando $T(n/2^k)$ llegue al caso base, es cuando termina la recursión. El caso base siempre tiene costo constante por lo que asumimos $T(n=1) = C$.

Si desarrollamos la ecuación, tenemos $T((n/2^k) = 1) = C$, se alcanza el caso base cuando $k = \log(n)$.

Para poder tener una idea de la complejidad debemos reemplazar k en $T(n) = 4^kT(n/2^k) + \sum_{i=0}^{k-1} 2^i n$

Quedando así:

$$T(n) = 4^{\log(n)}T(n/2^{\log(n)}) + \sum_{i=0}^{\log(n)-1} 2^i n$$

Aplicando propiedad de sumatoria y logaritmo:

$$T(n) = n^2T(1) + n \cdot \sum_{i=0}^{\log(n)-1} 2^i$$

Resolviendo la sumatoria:

$$\mathrm{T(n)} = n^2\mathrm{T(1)} + \mathrm{n} \cdot (\frac{2^{log(n)} - 1}{2 - 1}) = n^2\mathrm{T(1)} + \mathrm{n} \cdot (n - 1)$$

Finalmente reemplazamos $\mathrm{T(1)} = \mathrm{C}$ y nos queda que $\mathrm{T(n)} = n^2\mathrm{C} + n^2 - n, es\ decir\ T(n) \in O(n^2)$

Ahora que tenemos una idea de la complejidad de la recursión, podemos aplicar substitución para realizar la demostración exacta.

Idea de complejidad: $\mathrm{T(n)} = n^2\mathrm{C} + n^2 - n$ y asumimos que $\mathrm{T(1)} = \mathrm{C}$

- Usando inducción
  Caso base: n =1, $n^2\mathrm{C} + n^2 - n = C + 1 - 1 = C = T(n). Se\ cumple$
  Paso inductivo:
  $\mathrm{T(n)} = 4T(n/2) + n$

  $= 4\left(\left(\frac{n}{2}\right)^2 \mathrm{C} + \left(\frac{n}{2}\right)^2 - \frac{n}{2}\right) + n$ (Por HI)

  $= 4\frac{n^2}{4}\mathrm{C} + 4\frac{n^2}{4} - 4\frac{n}{2} + n = n^2\mathrm{C} + n^2 - n$

Luego, queda demostrado por inducción que la solución de $\mathrm{T(n)} = 4\mathrm{T(n/2)} + \mathrm{n}$ es $n^2\mathrm{C} + n^2 - n$

b) $\mathrm{T(n)} = \mathrm{T(n/2)} + \mathrm{c}$

Primero para poder tener una idea de la complejidad de T(n), utilizaremos el método de expansión:

En la primera iteración (k=1) tendremos:

$\mathrm{T(n)} = \mathrm{T(n/2)} + \mathrm{c}$

Para la segunda iteración (k=2) consideraremos que $\mathrm{T(n/2)} = \mathrm{T(n/4)} + \mathrm{c}$, por lo que nos queda:

$\mathrm{T(n)} = \mathrm{T(n/4)} + \mathrm{c} + \mathrm{c}$

Para la tercera iteración (k=3) consideraremos que $\mathrm{T(n/4)} = \mathrm{T(n/8)} + \mathrm{c}$, por lo que nos queda:

$\mathrm{T(n)} = \mathrm{T(n/8)} + \mathrm{c} + \mathrm{c} + \mathrm{c}$

Para la cuarta iteración (k=4) consideraremos que $\mathrm{T(n/8)} = \mathrm{T(n/16)} + \mathrm{c}$, por lo que nos queda:

$\mathrm{T(n)} = \mathrm{T(n/16)} + \mathrm{c} + \mathrm{c} + \mathrm{c} + \mathrm{c}$

Con las iteraciones realizadas, ya es posible dilucidar un patrón. En resumen, vemos lo siguiente:

$\mathrm{k=1, T(n)} = \mathrm{T(n/2)} + \mathrm{c} = \mathrm{T(n/2^1)} + (1)\mathrm{c}$

$\mathrm{k=2, T(n)} = \mathrm{T(n/4)} + \mathrm{c} + \mathrm{c} = \mathrm{T(n/2^2)} + (2)\mathrm{c}$

$\mathrm{k=3, T(n)} = \mathrm{T(n/8)} + \mathrm{c} + \mathrm{c} + \mathrm{c} = \mathrm{T(n/2^3)} + (3)\mathrm{c}$

k=4,$T(n) = T(n/16) + c + c + c + c = T(n/2^4) + (4)c$

El patrón que sigue sería:

$$T(n) = T(n/2^k) + kc$$

Cuando $T(n/2^k)$ llegue al caso base, es cuando termina la recursión. El caso base siempre tiene costo constante por lo que asumimos $T(n=1) = C$.

Si desarrollamos la ecuación, tenemos $T((n/2^k) = 1) = C$, se alcanza el caso base cuando $k = \log(n)$.

Para poder tener una idea de la complejidad debemos reemplazar k en $T(n) = T(n/2^k) + kc$

Quedando así:

$$T(n) = T(n/2^{log(n)}) + \log(n)c$$

Aplicando propiedad de logaritmo:

$$T(n) = T(1) + \log(n)c$$

Finalmente reemplazamos $T(1) = C$ y nos queda que $T(n) = C + \log(n)c$, *es decir* $T(n) \in O(\log(n))$

Ahora que tenemos una idea de la complejidad de la recursión, podemos aplicar substitución para realizar la demostración exacta.

Idea de complejidad: $T(n) = C + \log(n)c$ y asumimos que $T(1) = C$

- Usando inducción
  Caso base: n =1, $C + \log(n)c = C + \log(1)c = C = T(n)$. *Se cumple*
  Paso inductivo:
  $T(n) = T(n/2) + c$

  $= C + log(n/2)c + c$ (Por HI)

  $= C + log(n)c - log(2)c + c$

  $= C + log(n)c$

Luego, queda demostrado por inducción que la solución de $T(n) = T(n/2) + c$ es $C + log(n)c$.

PREGUNTA 6

a)

```
for( int i = 1; i <= n; i *= 2 ) { //O(log(n))
       for( int j = 0; j < i; j++ ) {   // O(1)+O(2)+O(4)+O(8)+…+O(n-1)
              for( int k = 0; k < n; k += 2 ) { //O(n)
                     f(); // O(1)
              }
              for( int l = 1; l < n; l *= 2 ) {  //O(log(n))
                     g() // O(1)
              }
       }
}
```

Dentro de for(int j) tenemos O(n) y O(log(n)) como O(n) domina a la otra, solo consideraremos esa.
Luego tenemos que la complejidad de for(int j) depende de los valores que tome i, por lo que no es posible evaluarlo independientemente

$$\sum_{i=1}^{\log(n)} \sum_{j=0}^{i-1} O(n)$$

Sin embargo, como i toma valores 2^i, por lo que nos queda de la siguiente manera.

$$\sum_{i=1}^{\log(n)} \sum_{j=0}^{2^i-1} O(n)$$

$$\Leftrightarrow O(n) \sum_{i=1}^{\log(n)} \sum_{j=0}^{2^i-1} 1$$

$$\Leftrightarrow O(n) \sum_{i=1}^{\log(n)} \left(2^i - 1\right)$$

$$\Leftrightarrow O(n) \left[\sum_{i=1}^{\log(n)} 2^i - \sum_{i=1}^{\log(n)} 1 \right]$$

$$\Leftrightarrow O(n)\left[2(2^{\log(n)} - 1) - \log(n)\right]$$

$$\Leftrightarrow O(n)[2(n-1) - \log(n)]$$

$$\Leftrightarrow O(n)(2\mathrm{n} - 2) - \mathrm{O(n)log\ (n)}]$$

De aquí podemos ver que el término que domina es $O(n)(2\mathrm{n} - 2)$

$$\therefore O(n^2)$$

b)

```
for ( i=1; i < n; i *= 2 ) { //O(log(n))
       for ( j = n; j > 0; j /= 2 ) { //j={n, n/2, n/4, n/8, ... } O(log(n))
              for ( k = j; k < n; k += 2 ) {
                     sum += (i + j * k );  //O(1)
              }
       }
}
```

Como for(k) depende de for(j), no podemos expresarlo independiente

$$\sum_{j=0}^{\log(n)} \sum_{k=\frac{n}{2^j}}^{n} O(1)$$

$$\Leftrightarrow \sum_{j=0}^{\log(n)} \sum_{k=0}^{n-\frac{n}{2^j}} O(1)$$

$$\Leftrightarrow \sum_{j=0}^{\log(n)} (n - \frac{n}{2^j} + 1)$$

$$\Leftrightarrow \sum_{j=0}^{\log(n)} n + n \cdot \sum_{j=0}^{\log(n)} 2^{-j} + \sum_{j=0}^{\log(n)} 1$$

$$\Leftrightarrow nlog(n) + n\left(1 - 2^{-\log(n)}\right) + \log(n)$$

$$\Leftrightarrow nlog(n) + n(1+n) + \log(n)$$

Luego nos faltaría multiplicar por el costo de for(i):

$$\log(n) * [nlog(n) + n(1+n) + \log(n)]$$

$$nlog^2(n) + n(1+n)\log(n) + log^2(n)$$

De aquí se ve que el término que domina es $nlog^2(n)$

$$\therefore O(nlog^2(n))$$

c)

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

Del for(i=0) tenemos dentro una instrucción O(n), por lo que nos quedaría un for de O(n) * $O(n^2) = O(n^3)$

Nos quedaría que F, para n <=0 nos resultaría una complejidad O(1), mientras que para n>0 $O(n^3) + T(n-3)$

Desarrollamos la recurrencia mediante iteración:

$$F(n) = \begin{cases} 1, & n \leq 0 \\ F(n-3) + O(n^3), & n > 0 \end{cases}$$

$$F(n-3) = F(n-6) + (n-3)^3$$

$$para\ k = 2, \qquad F(n) = F(n-6) + (n-3)^3 + n^3$$
$$F(n-6) = F(n-9) + (n-6)^3$$

$$para\ k = 3, \qquad F(n) = F(n-9) + (n-6)^3 + (n-3)^3 + n^3$$
$$F(n-9) = F(n-12) + (n-9)^3$$

$$para\ k = 4, \qquad F(n) = F(n-12) + (n-9)^3 + (n-6)^3 + (n-3)^3 + n^3$$
$$para\ simplificar, \qquad F(n) = F(n-3k) + \sum_{i=0}^{k-1} (n-3i)^3$$

$$tomando\ F(0) = 1$$
$$F\big((n-3\cdot k) = 0\big) = 1$$
$$n - 3\cdot k = 0$$
$$k = {}^{n}/_{3}$$

$$Luego\ reemplazando\ k, \qquad F(n) = F\big(n - 3\cdot {}^{n}/_{3}\big) + \sum_{i=0}^{{}^{n}/_{3}-1} (n-3i)^3$$

$$F(n) = F(0) + \sum_{i=1}^{{}^{n}/_{3}-1} (n-3i)^3$$

$$F(n) = F(0) + \frac{n^2(n+3)^2}{12}$$

$$F(n) = 1 + \frac{n^2(n+3)^2}{12}$$

$$Luego, F(0) = 1 + \frac{0^2(0+3)^2}{12} = 0, por\ lo\ que\ satisface\ para\ F(0)$$

Ahora podemos usar la expresión actual con el método de substitución para obtener la complejidad de la recurrencia.

Caso Base:

$$F(0) = 1 + \frac{0^2(0+3)^2}{12} = 0$$

Hipótesis:

$$F(n-3) = 1 + \frac{(n-3)^2(n-3+3)^2}{12} = 0$$

$$F(n-3) = 1 + \frac{(n-3)^2 n^2}{12} = 0$$

Paso inductivo:

$$F(n) = F(n-3) + n^3$$

$$F(n) = 1 + \frac{(n-3)^2 n^2}{12} + n^3$$

$$F(n) = 1 + \frac{(n^2 - 6n + 9)n^2}{12} + n^3$$

$$F(n) = 1 + \frac{(n^4 - 6n^3 + 9n^2)}{12} + n^3$$
$$F(n) = 1 + \frac{(n^4 - 6n^3 + 9n^2) + 12n^3}{12}$$
$$F(n) = 1 + \frac{(n^4 + 6n^3 + 9n^2)}{12}$$
$$F(n) = 1 + \frac{n^2(n^2 + 6n + 9)}{12}$$
$$F(n) = 1 + \frac{n^2(n+3)^2}{12}$$

Con lo que llegamos a la expresión obtenida anteriormente, por lo que queda demostrada y además se demuestra que:

$$F(n) \in O(n^4)$$

d)

```
void G(a[], n){
	if( n <= 1) //O(1)
		return;
	else  //O(n)
		for(i=1; i<= n-1; i++){ //O(n)
			if (a[i] > a[n] ) //O(1)
				intercambiar(a[i], a[n]) //O(1)
		}
		G(a[], n-1) //G(n-1) + cn
}
```

Acá se tiene que cuando n>1 la complejidad es de O(n) mientras que en caso contrario es $O(1)$.

$$G(n) = \begin{cases} 1, & n \leq 1 \\ G(n-1) + O(n), & n > 1 \end{cases}$$

$$G(n-1) = G(n-2) + (n-1)$$

Utilizando método de iteraciones:

$$para\ k = 1, \qquad G(n) = G(n-2) + (n-1) + n$$
$$G(n-2) = G(n-3) + (n-2)$$

$$para\ k = 2, \qquad G(n) = G(n-3) + (n-2) + (n-1) + n$$
$$G(n-3) = G(n-4) + (n-3)$$

$$para\ k = 3, \qquad G(n) = G(n-4) + (n-3) + (n-2) + (n-1) + n$$

$$G(n) = G(n-i-1) + \sum_{k=0}^{i} (n-k)$$

$$Como\ se\ tiene\ que\ G(1) = 1$$
$$G\big((n-i-1) = 1\big) = 1$$

$$n - i - 1 = 1$$
$$n - 2 = i$$
$$G(n) = G(1) + \sum_{k=0}^{n-2} (n - k)$$
$$G(n) = 1 + n \sum_{k=0}^{n-2} 1 - \sum_{k=0}^{n-2} k$$
$$G(n) = 1 + n(n-1) - \left(\frac{(n-1)(n-2)}{2}\right)$$
$$G(n) = 1 + n^2 - n - \left(\frac{n^2 - 3n + 2}{2}\right)$$
$$G(n) = \frac{n^2}{2} + \frac{n}{2}$$
$$G(n) = \frac{n(n+1)}{2}$$

$$Luego, G(1) = \frac{1(1+1)}{2} = 1$$

Ahora podemos usar la expresión actual con el método de substitución para obtener la complejidad de la recurrencia.
Caso Base:

$$G(1) = \frac{1(1+1)}{2} = 1$$

Paso Inductivo:

$$Hipotesis: G(n-1) = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$
$$G(n) = G(n-1) + n$$
$$G(n) = \frac{n(n-1)}{2} + n$$
$$G(n) = \frac{n^2 - n}{2} + n$$
$$G(n) = \frac{n^2 + n}{2}$$

Con lo que queda demostrado que:

$$G(n) = \frac{n(n+1)}{2}$$

Luego, como se ve que el término que predomina es n^2, por lo tanto:

$$G(n) \in O(n^2)$$

PREGUNTA 7

a)

```
int F(a,b)   // b * a;   O(n)
       x=a, y=b, z = 0
       while( x > 0)  //O(n)
              z = z + y  //O(1)
              x = x - 1  //O(1)
       return z       //O(1)
```

Se tiene un único while que realiza operaciones O(1), por lo que la función tiene complejidad O(n).

El algoritmo debería retornar la multiplicación entre a y b. Esto se realiza sumando **y** a **z**, **x** veces, y decrementando **x** en una unidad para llegar a la condición de termino y no se sume más de **x** veces. Si probamos con diferentes entradas los podemos verificar que cumple en esos casos al menos

$$a = 1, b = 1 \rightarrow return\ 1$$
$$a = 2, b = 2 \rightarrow return\ 4$$
$$a = 3, b = 3 \rightarrow return\ 9$$
$$a = 5, b = 4 \rightarrow return\ 20$$
$$a = 4, b = 5 \rightarrow return\ 20$$
$$a = 7, b = 9 \rightarrow return\ 63$$

Sin embargo, necesitamos analizar su correctitud, para ello debemos identificar el loop invariante.
La invariante en este caso podría ser $z + (x \cdot y) = a * b$

Tenemos el caso base $z = 0, x = a, y = b$

$$0 + (a * b) = (a * b)$$

Por lo que el caso base se cumple.

Consideremos ahora **z1, x1 y y1** como los valores finales de la iteración k, o los iniciales de la iteración **k+1** y **z2, x2, y2** como los valores finales de **k+1**.

Tenemos por hipótesis entonces que se cumple en la iteración **k**

$$z1 + (x1 * y1) = (a * b)$$

Visto de otra forma

$$z2 + (x2 * y2) = z1 + (x1 * y1)$$

En la iteración **k+1**, se cumple que z1+(x1* y1) = (a*b), luego **z1** se incrementa en **y1** unidades, **x1** se decrementa en una unidad e **y1** permanece invariante

$$z2 = z1 + y1$$

$$x2 = x1 - 1$$

$$y2 = y1$$

Entonces reemplazando nos queda:

$$(z1 + y1) + \big((x1 - 1) * y2\big) = z1 + (x1 * y1)$$

$$(z1 + y1) + (x1 * y2 - y2) = z1 + (x1 * y1)$$

$$(z1 + y1) + x1 * y2 - y2 = z1 + (x1 * y1)$$

Como y1 = y2, se simplifica

$$z1 + x1 * y2 = z1 + (x1 * y2) = z2 + (x2 * y2) = a * b$$

Luego por inducción la invariante es verdadera mientras dure el ciclo.

Finalmente dado que la condición de termino es (x>0) en la última iteración x toma el valor 0, por lo que al final termina el ciclo.

b)

```
int G(a,b)  //O(n)
      x=a, y=b, z = 0; // O(1)
      while( x > 0)  { //O(n)
            if (x % 2 == 1)  //O(1)  cuando x es impar
                  z = z + y;    //O(1)    z = se incrementa en y;
            x = x>>1 ;        //O(1)   = /2  x se decrementa a la mitad
            y = y<<1;         //O(1)  << = *2 y se duplica
      }
      return z
```

Se tiene un único while que realiza operaciones O(1), por lo que la función tiene complejidad O(n).

El algoritmo debería retornar la multiplicación entre `a` y `b`. Esto se realiza incrementando en **y** a **z**, cada vez que el residuo de la división entera entre **x** y 2 sea igual a 1. En cada iteración del while, **x** es dividido por 2, e **y** es duplicado. Llegando finalmente a la condición de termino cuando x alcanza el valor 0

$$a = 1, b = 1 \rightarrow return\ 1$$
$$a = 2, b = 2 \rightarrow return\ 4$$
$$a = 3, b = 3 \rightarrow return\ 9$$
$$a = 5, b = 4 \rightarrow return\ 20$$
$$a = 4, b = 5 \rightarrow return\ 20$$
$$a = 7, b = 9 \rightarrow return\ 63$$

Tenemos el caso base $z = 0, x = a, y = b$

$$0 + (a * b) = (a * b)$$

Por lo que el caso base se cumple.

Visto de otra forma

$$z2 + (x2 * y2) = z1 + (x1 * y1)$$

En la iteración **k+1**, se cumple que z1+(x1* y1) = (a*b), luego **z1** se incrementa en **y1** si y solo si el resto de la división entera entre **x1** y 2 es 1, asumiremos en este caso que es así. Luego **x** se decrementa a la mitad, mientras que **y** se duplica

$$z2 = z1 + y1$$
$$x2 = x1 \gg 1 = \frac{x1 - 1}{2}$$
$$y2 = y1 \ll 1 = 2 * y1$$

Como en este caso, la división entera deja un resto, tomamos $x2 = \frac{x1-1}{2}$

Reemplazando nos queda:

$$z2 + (x2 * y2) = (z1 + y1) + \left(\frac{(x1 - 1)}{2} * 2y1\right) = z1 + (x1 * y1)$$

$$z1 + y1 + \big((x1 - 1) * y1\big) = z1 + (x1 * y1)$$

$$z1 + y1 + (x1 * y1 - y1) = z1 + (x1 * y1)$$

Luego se cumple que:

$$z2 + (x2 * y2) = z1 + (x1 * y1) = a * b, \qquad cuando\ x\%2 = 1$$

Para el caso en el que **x** % 2 = 0 tenemos lo siguiente:

$$z2 = z1$$

$$x2 = x1 \gg 1 = \frac{x1}{2}$$
$$y2 = y1 \ll 1 = 2 * y1$$

Como **x** % 2 = 0, en este caso no se entra al `if` y por ende **z** no varía su valor y además como el resto de la división entera de **x** ahora si podemos tomar $x2 = \frac{x1}{2}$

Reemplazando nos queda:

$$z2 + (x2 * y2) = z1 + (\frac{x1}{2} * 2y1) = z1 + (x1 * y1)$$

$$z1 + (x1 * y1) = z1 + (x1 * y1)$$

Luego se cumple que:

$$z2 + (x2 * y2) = z1 + (x1 * y1) = a * b, \qquad cuando\ x\%2 = 0$$

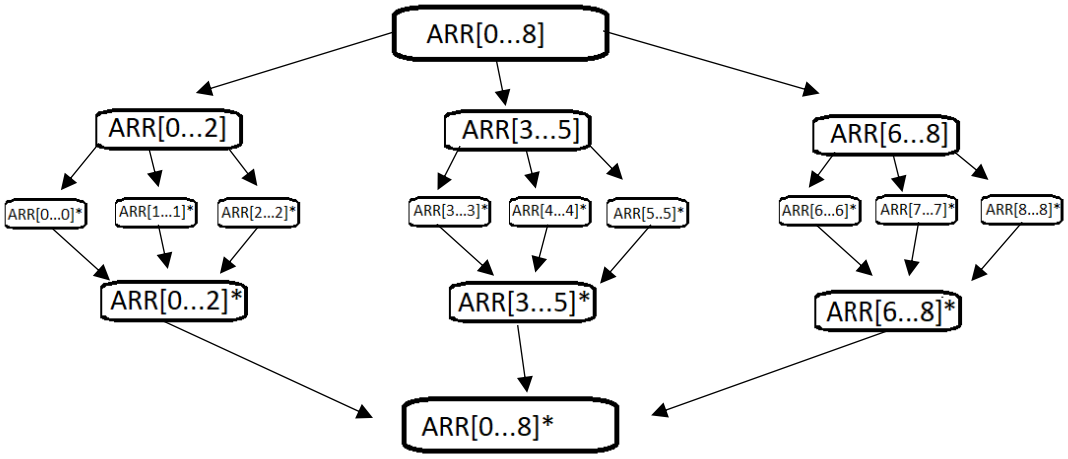
Entonces queda demostrado que la invariante se cumple cuando se entra al `if` y cuando no se entra, luego queda demostrado que esto aplica para cualquier valor de **k**

Finalmente, el algoritmo termina ya que en la penúltima iteración se tiene que $x = 1$ y esto al aplicarle " $\ll$ " resulta en 0 por lo que el ciclo **`while`** termina

PREGUNTA 8

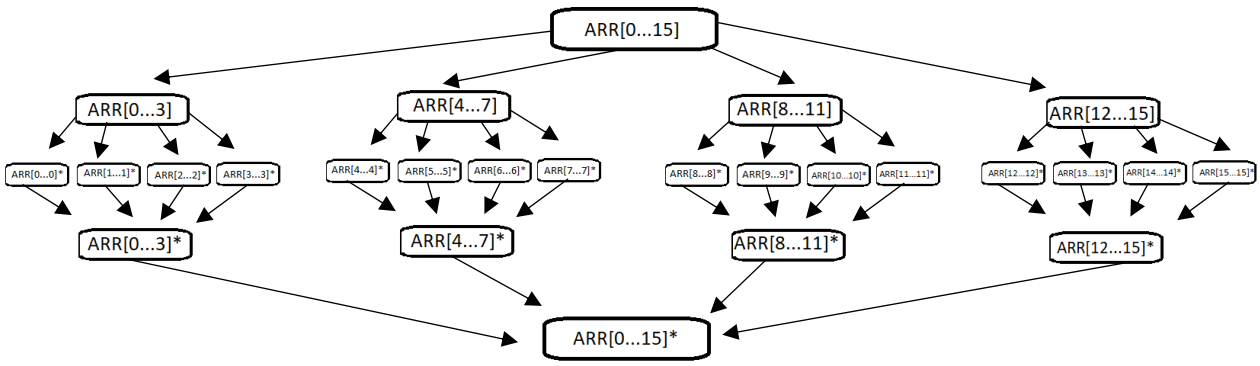
a)

En el siguiente ejemplo utilizamos un arreglo con 9 números, los dígitos que se muestran en el bosquejo corresponden a los índices del arreglo. Los subarreglos se dividen hasta llegar al caso base. Los arreglos con * son los modificados después de realizar el merge.



b)

En el siguiente ejemplo utilizamos un arreglo con 16 números, los dígitos que se muestran en el bosquejo corresponden a los índices del arreglo. Los subarreglos se dividen en 4 hasta llegar al caso base. Los arreglos con * son los modificados después de realizar el Merge.



c)

Para analizar la recurrencia del MergeSort que se divide en 3, consideramos que trabaja 3 arreglos por separado y analiza cada uno de estos. Por ende, es un algoritmo "dividir para conquistar". Por lo que siguiendo la forma de una recurrencia general la ecuación T(n) queda:

$$\mathrm{T(n)} = 3\mathrm{T(n/3)} + \mathrm{cn}$$

Para trabajar esta recurrencia utilizaremos el árbol de recurrencia.

Guiándonos de lo realizado en la parte 1, podemos tener en cuenta que al utilizar el árbol recursivo consideramos:

1. El tamaño del problema de la entrada de la recurrencia en el nivel i es $\frac{n}{3^i}$
2. Si calculamos el momento en que se llega al caso base (asumiendo T=1), podemos calcular la altura: $\frac{n}{3^i} = 1$. Luego la altura dado por el nivel i es $\log_3(n)$
3. El número de hojas en cualquier nivel es 3^i
4. En el nivel i hay 3^i hojas con costo de $\mathrm{c} \cdot (\frac{n}{3^i})$. Por lo tanto, la suma de los costos al nivel i es $3^i \cdot \mathrm{c} \cdot (\frac{n}{3^i})$.

Con los resultados anteriores podemos calcular el costo total:

$\mathrm{T(n)} = \mathrm{cn} + 3 \cdot \mathrm{c(n/3)} + 9 \cdot \mathrm{c(n/9)} + 27 \cdot \mathrm{c(n/27)} + + 81 \cdot \mathrm{c(n/81)} + \ldots$

$$\mathrm{T(n)} = \sum_{i=0}^{\log_3(n)-1} c \cdot (\frac{n}{3^i}) \cdot 3^i + 3^{\log_3(n)} \cdot c \cdot T(1)$$

Aplicando simplificación de términos, propiedades de sumatoria y logaritmo, además asumiendo que T(1) toma valor constante, al igual que la constante c. Podemos considerar lo siguiente:

$$\mathrm{T(n)} = cn \sum_{i=0}^{\log_3(n)-1} (1) + nc$$

Resolviendo la sumatoria nos queda:

$$\mathrm{T(n)} = cn(\log_3(n) - 1) + nc$$

Reduciendo así T(n) a la siguiente expresión:

$$T(n) = cn \log_3(n)$$

Finalmente, tomando en cuenta que c es una constante es posible asumir que $\mathrm{T(n)} \in \mathrm{O}(n \log_3(n))$.

Ahora de manera análoga trabajamos el Merge Sort que divide en 4, que al igual que el otro algoritmo se rige por el "dividir para conquistar". Por lo que siguiendo la forma de una recurrencia general la ecuación T(n) queda:

$$T(n) = 4T(n/4) + cn$$

Guiándonos de lo realizado en la parte 2, podemos tener en cuenta que al utilizar el árbol recursivo consideramos:

1. El tamaño del problema de la entrada de la recurrencia en el nivel i es $\frac{n}{4^i}$
2. Si calculamos el momento en que se llega al caso base (asumiendo T=1), podemos calcular la altura: $\frac{n}{4^i} = 1$. Luego la altura dado por el nivel i es $\log_4(n)$
3. El número de hojas en cualquier nivel es 4^i
4. En el nivel i hay 4^i hojas con costo de $c \cdot (\frac{n}{4^i})$. Por lo tanto, la suma de los costos al nivel i es $4^i \cdot c \cdot (\frac{n}{4^i})$.

Con los resultados anteriores podemos calcular el costo total:

$$T(n) = cn + 4 \cdot c(n/4) + 16 \cdot c(n/16) + 64 \cdot c(n/64) + + 256 \cdot c(n/256) + \ldots$$

$$T(n) = \sum_{i=0}^{\log_4(n)-1} c \cdot (\frac{n}{4^i}) \cdot 4^i + 4^{\log_4(n)} \cdot c \cdot T(1)$$

Realizando el mismo procedimiento que se trabajó anteriormente, tenemos:

$$T(n) = cn \sum_{i=0}^{\log_3(n)-1} (1) + nc$$

$$T(n) = cn(\log_4(n) - 1) + nc$$

$$T(n) = cn \log_4(n)$$

Finalmente, tomando en cuenta que c es una constante es posible asumir que $T(n) \in O(n \log_4(n))$.

Análisis teórico

Realizando un análisis de los resultados, es posible visualizar que en términos de tiempo de ejecución es lo siguiente:

- El Merge Sort 4 $\in O(n \log_4(n))$, por lo que es posible decir que cuando se utilizan grandes volúmenes de datos su comportamiento debería ser más rápido que los demás algoritmos.
- El Merge Sort 3$\in O(n \log_3(n))$, es el que le sigue al Merge Sort 4.
- El Merge Sort 2$\in O(n \log_2(n))$, es el menos eficiente cuando se trabaja con grandes datos.

Sin embargo, cuando son bajos datos los que se procesan en los algoritmos, ocurre lo contrario. Es decir, el Merge Sort 2 es más rápido, le sigue el Merge Sort 3 y luego el

Merge Sort 4. Lo anterior se debe a que en los últimos dos algoritmos mencionados se realizan más particiones y esto cuando hay pocos datos, puede resultar costo. Debido a que con pocos datos es posible ordenar la secuencia de manera más directa con menos particiones/separaciones en el arreglo.

d)

Algoritmos Merge:

```
void merge_2(vector<int>* arr, long long int l, long long int m, long long int r){
     /*
          Se separan los elementos del arreglo desde el elemento 'l'
          hasta el 'r', en dos vectores distintos,
          para luego ir comparando el extremo izquierdo de cada uno
          y asignarlos en la posicion correspondiente del vector
          original 'arr'
    */
    long long int n1 = m - l + 1;
    long long int n2 = r - m;
    vector<int> L;
    vector<int> R;

    for (long long int i = 0; i < n1; i++)L.push_back(arr->at(l + i));
    for (long long int j = 0; j < n2; j++)R.push_back(arr->at(m + 1 + j));

    long long int i = 0;
    long long int j = 0;
    long long int k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr->at(k) = L[i];
            i++;
        }
        else {
            arr->at(k) = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr->at(k) = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr->at(k) = R[j];
        j++;
        k++;
    }
}
```

```cpp
void merge_3(vector<int>* arr,long long int l,long long int m1, long long int m2, long long int r){
     /*
          Se crean tres vectores, 'L' contendra los elementos desde 'l' hasta 'm1'
          El vector 'M', contendra los elementos desde 'm1+1' hasta 'm2'
          y el vector 'R' desde 'm2+1' hasta 'r'
          Luego se va comparando los extremos izquierdos de los tres vectores y al menor
          de ellos se ubica en la posicion correspondiente del arreglo 'arr' original
    */
    long long int n1 = m1 - l + 1;
    long long int n2 = m2 - m1;
    long long int n3 = r - m2;
    vector<int> L;
    vector<int> M;
    vector<int> R;

    for (long long int i = 0; i < n1; i++)L.push_back(arr->at(l + i));
    for (long long int j = 0; j < n2; j++)M.push_back(arr->at(m1 + 1 + j));
    for (long long int j = 0; j < n3; j++)R.push_back(arr->at(m2 + 1 + j));

    long long int i = 0;
    long long int j = 0;
    long long int z = 0;
    long long int k = l;

    while (i < n1 && j < n2 && z <n3) {
        if (L[i] <= M[j]) {
            if(L[i] <= R[z]){
                arr->at(k) = L[i];
                i++;
            }
            else{
                arr->at(k) = R[z];
                z++;
            }
        }
        else if (M[j]<= R[z]) {
            arr->at(k) = M[j];
            j++;
        }
        else{
            arr->at(k) = R[z];
            z++;
        }
        k++;
    }
    while ((i < n1) && (j < n2)){
        if (L[i] <= M[j]) {
            arr->at(k) = L[i];
            i++;
            k++;
        }else{
            arr->at(k) = M[j];
            j++;
            k++;
        }
```

```
    }
    while ((j < n2) && (z < n3)){
        if (M[j] <= R[z]) {
            arr->at(k) = M[j];
            j++;
            k++;
        }else{
            arr->at(k) = R[z];
            z++;
            k++;
        }
    }
    while ((i < n1) && (z < n3)){
        if (L[i] <= R[z]) {
            arr->at(k) = L[i];
            i++;
            k++;
        }else{
            arr->at(k) = R[z];
            z++;
            k++;
        }
    }
    while (i < n1) {
        arr->at(k) = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr->at(k) = M[j];
        j++;
        k++;
    }
    while (z < n3) {
        arr->at(k) = R[z];
        z++;
        k++;
    }
}
```

```
void merge_4(vector<int>* arr, long long int l, long long int m1, long long int m2, long long int
m3, long long int r){
/*
          Se aprovecha la funcion "merge_2" definida anteriormente,  en ella se hace merge_2
primero con los elementos
          desde 'l' hasta 'm2'. Luego se hace merge_2 desde 'm2+1' hasta 'r'
          y finalmente a lo obtenido de los dos anteriores se hace merge_2, es decir desde
'l' hasta 'r'
          */
    merge_2(arr,l, m1, m2);
    merge_2(arr,m2+1, m3, r);
    merge_2(arr,l, m2, r);

}
```

Algoritmo MergeSort:

```
void mergeSort_3(vector<int>* arr,long long int l,long long int r){
          /*
          Algoritmo base de MergeSort 3, hace una division de 3 equidistante entre 'l' y 'r', para
luego
          llamar recursivamente al metodo, cuando l>=r deja de llamarse recursivamente y empiza a
llamarse
          el metodo merge_3 para unir el arreglo ordenado.
          */
    if(l>=r){
        return;
    }
    long long int m1 =l+ (r-l)/3;
    long long int m2 =l+ 2*((r-l)/3) +1;

    mergeSort_3(arr,l,m1);
    mergeSort_3(arr,m1+1,m2);
    mergeSort_3(arr,m2+1,r);
    merge_3(arr,l,m1,m2,r);
}
```

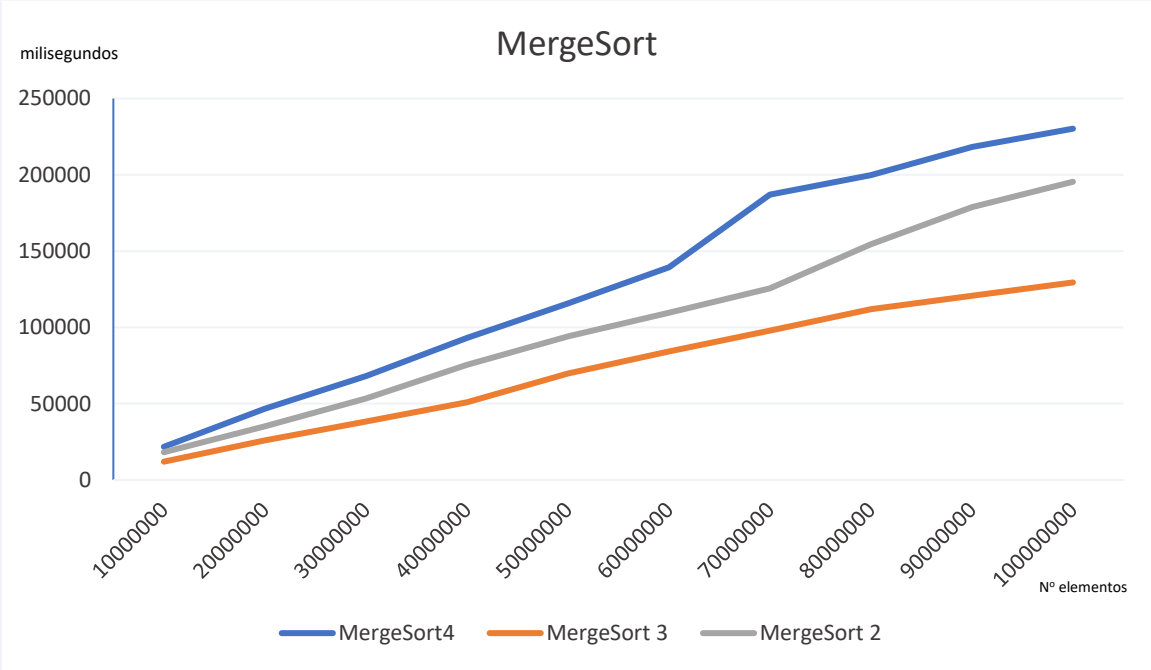
```
void mergeSort_4(vector<int>* arr,long long int l,long long int r){
          /*
          Algoritmo base de MergeSort 4, hace una division de 4 equidistante entre 'l' y 'r', para
luego
          llamar recursivamente al metodo, cuando l>=r deja de llamarse recursivamente y empiza a
llamarse
          el metodo merge_4 para unir el arreglo ordenado.

          la condicion del if ((r-l)%4) cumple la funcion de no excederse en los limites de los
indices del arreglo
          */
    if(l>=r){
        return;
    }

    long long int m1 =l+ (r-l)/4;
    long long int m2 =l+ 2*((r-l)/4) +1;
    long long int m3 =l+ 3*((r-l)/4) +2;
    if(((r-l)%4) != 0){
          m2--;
          m3--;
          m1--;
    }
    mergeSort_4(arr,l,m1);
    mergeSort_4(arr,m1+1,m2);
    mergeSort_4(arr,m2+1,m3);
    mergeSort_4(arr,m3+1,r);
    merge_4(arr,l,m1,m2,m3,r);
}
```

```
void mergeSort_2(vector<int>* arr,long long int l,long long int r){

          /*
          Algoritmo base de MergeSort 2, hace una division equidistante entre 'l' y 'r', para luego
          llamar recursivamente al metodo, cuando l>=r deja de llamarse recursivamente y empieza a
llamarse
          el metodo merge_2 para unir el arreglo ordenado.
          */
    if(l>=r){
        return;
    }
    long long int m =l+ (r-l)/2;
    mergeSort_2(arr,l,m);
    mergeSort_2(arr,m+1,r);
```



Nº de Elementos	MergeSort4	MergeSort 3	MergeSort 2
10000000	21743 [ms]	12001 [ms]	18176 [ms]
20000000	46697 [ms]	25977 [ms]	35088 [ms]
30000000	68141 [ms]	38217 [ms]	53377 [ms]
40000000	93143 [ms]	50831 [ms]	75424 [ms]
50000000	115645 [ms]	69796 [ms]	94203 [ms]
60000000	139435 [ms]	84232 [ms]	109660 [ms]
70000000	186980 [ms]	97979 [ms]	125574 [ms]
80000000	199883 [ms]	111925 [ms]	154444 [ms]
90000000	218255 [ms]	120712 [ms]	178897 [ms]
100000000	230252 [ms]	129460 [ms]	195472 [ms]

Análisis experimental

Como se puede apreciar del grafico y de la tabla, el algoritmo que obtuvo mejores resultados fue MergeSort 3, seguido del MergeSort 2 y por último el MergeSort 4. Sin embargo, si contrastamos con el análisis teórico obtenido; se pudo observar que al resolver las recurrencias era más eficiente MergeSort 4, luego el MergeSort 3 y por último el MergeSort 2. Se considera que esta situación puede explicarse debido a las características de la maquina en la que se hicieron las pruebas. Lo anterior se deduce a raíz de que a pesar de que se realizaron múltiples pruebas, con una cantidad muy alta de elementos, no existe una diferencia muy significativa en cuanto a los tiempos obtenidos entre todos los algoritmos.

En conclusión, al realizar análisis teórico versus experimental es posible obtener resultados variados. Esto se debe a que el ambiente de prueba influye añadiendo márgenes de error en los resultados.