

Práctica 8

1. Determinar para cada función $t(n)$ en la siguiente tabla, cuál es el mayor tamaño n de una instancia de un problema que puede ser resuelto en cada uno de los tiempos indicados en las columnas de la tabla, suponiendo que el algoritmo para resolverlo utiliza $t(n)$ microsegundos.

| $t(n)$ | 1 seg. | 1 min. | 1 hora | 1 día | 1 mes | 1 año | 1 siglo |
|----------------------|---------------------|----------------------|------------------------|--------------------------|---------------------------|-----------------------------|-----------------------------|
| $\log_2(n)$ | 2^{10^6} | $2^{6 \times 10^7}$ | $2^{36 \times 10^8}$ | $2^{864 \times 10^8}$ | $2^{25920 \times 10^8}$ | $2^{311040 \times 10^8}$ | $2^{31104000 \times 10^8}$ |
| \sqrt{n} | 10^{12} | 3.6×10^{15} | 1.296×10^{19} | 7.46496×10^{21} | 6.718464×10^{24} | $9.67458816 \times 10^{26}$ | $9.67458816 \times 10^{30}$ |
| n | 10^6 | 6×10^7 | 3.6×10^9 | 8.64×10^{10} | 2.592×10^{12} | 3.1104×10^{13} | 3.1104×10^{15} |
| $n \times \log_2(n)$ | 6.275×10^4 | 2.801×10^6 | 1.334×10^8 | 2.755×10^9 | 7.187×10^{10} | 7.871×10^{11} | 6.77×10^{13} |
| n^2 | 10^3 | 7.746×10^3 | 6×10^4 | 2.9393×10^5 | 1.61046×10^6 | 5.5757×10^6 | 5.5757×10^7 |
| 2^n | 19.93 | 25.84 | 31.75 | 36.33 | 41.24 | 44.82 | 51.46 |
| $n!$ | ≈ 9 | ≈ 11 | ≈ 12 | ≈ 13 | ≈ 15 | ≈ 16 | ≈ 17 |

- **Referencias de tiempo en microsegundos (μs):**

- $1 \text{ seg} = 10^6 \mu s$
- $1 \text{ min} = 60 \times 10^6 \mu s$
- $1 \text{ hora} = 3600 \times 10^6 \mu s$
- $1 \text{ día} = 24 \times 3600 \times 10^6 \mu s = 86400 \times 10^6 \mu s$
- $1 \text{ mes} = 30 \times 24 \times 3600 \times 10^6 \mu s = 2592000 \times 10^6 \mu s$
- $1 \text{ año} = 12 \times 30 \times 24 \times 3600 \times 10^6 \mu s = 31104000 \times 10^6 \mu s$
- $1 \text{ siglo} = 100 \times 12 \times 30 \times 24 \times 3600 \times 10^6 \mu s = 3110400000 \times 10^6 \mu s$

1. $\log_2(n)$:

i. **1 seg:**

- $\log_2(n) = 10^6$
- $2^{10^6} = n$
- $n = 2^{10^6}$

ii. **1 min:**

- $\log_2(n) = 60 \times 10^6$
- $2^{60 \times 10^6} = n$
- $n = 2^{60 \times 10^6}$

d. $n = 2^{6 \times 10^7}$

iii. **1 hora:**

a. $\log_2(n) = 3600 \times 10^6$

b. $2^{3600 \times 10^6} = n$

c. $n = 2^{3600 \times 10^6}$

d. $n = 2^{36 \times 10^8}$

iv. **1 día:**

a. $\log_2(n) = 24 \times 3600 \times 10^6$

b. $2^{24 \times 3600 \times 10^6} = n$

c. $n = 2^{24 \times 3600 \times 10^6}$

d. $n = 2^{24 \times 36 \times 10^8}$

e. $n = 2^{864 \times 10^8}$

v. **1 mes:**

a. $\log_2(n) = 30 \times 24 \times 3600 \times 10^6$

b. $2^{30 \times 24 \times 3600 \times 10^6} = n$

c. $n = 2^{30 \times 24 \times 3600 \times 10^6}$

d. $n = 2^{30 \times 24 \times 36 \times 10^8}$

e. $n = 2^{25920 \times 10^8}$

vi. **1 año:**

a. $\log_2(n) = 12 \times 30 \times 24 \times 3600 \times 10^6$

b. $2^{12 \times 30 \times 24 \times 3600 \times 10^6} = n$

c. $n = 2^{12 \times 30 \times 24 \times 3600 \times 10^6}$

d. $n = 2^{12 \times 30 \times 24 \times 36 \times 10^8}$

e. $n = 2^{311040 \times 10^8}$

vii. **1 siglo:**

a. $\log_2(n) = 100 \times 12 \times 30 \times 24 \times 3600 \times 10^6$

b. $2^{100 \times 12 \times 30 \times 24 \times 3600 \times 10^6} = n$

c. $n = 2^{100 \times 12 \times 30 \times 24 \times 3600 \times 10^6}$

d. $n = 2^{100 \times 12 \times 30 \times 24 \times 36 \times 10^8}$

e. $n = 2^{31104000 \times 10^8}$

2. \sqrt{n} :

i. **1 seg:**

a. $\sqrt{n} = 10^6$

b. $n = (10^6)^2$

c. $n = 10^{12}$

ii. **1 min:**

a. $n = 60^2 \times 10^{12}$

b. $n = 3600 \times 10^{12}$

c. $n = 3.6 \times 10^{15}$

iii. **1 hora:**

a. $n = 60^2 \times (3.6 \times 10^{15})$

b. $n = 3600 \times (3.6 \times 10^{15})$

c. $n = 12960 \times 10^{15}$

d. $n = 1.296 \times 10^{19}$

iv. **1 día:**

- a. $n = 24^2 \times (1.296 \times 10^{19})$
- b. $n = 576 \times (1.296 \times 10^{19})$
- c. $n = 746.496 \times 10^{19}$
- d. $n = 7.46496 \times 10^{21}$

v. **1 mes:**

- a. $n = 30^2 \times (7.46496 \times 10^{21})$
- b. $n = 900 \times (7.46496 \times 10^{21})$
- c. $n = 6718.464 \times 10^{21}$
- d. $n = 6.718464 \times 10^{24}$

vi. **1 año:**

- a. $n = 12^2 \times (6.718464 \times 10^{24})$
- b. $n = 144 \times (6.718464 \times 10^{24})$
- c. $n = 967.458816 \times 10^{24}$
- d. $n = 9.67458816 \times 10^{26}$

vii. **1 siglo:**

- a. $n = 100^2 \times (9.67458816 \times 10^{26})$
- b. $n = 10000 \times (9.67458816 \times 10^{26})$
- c. $n = 96745.8816 \times 10^{26}$
- d. $n = 9.67458816 \times 10^{30}$

3. **n :**

i. **1 seg:**

a. $n = 10^6$

ii. **1 min:**

a. $n = 60 \times 10^6$
 b. $n = 6 \times 10^7$

iii. **1 hora:**

a. $n = 3600 \times 10^6$
 b. $n = 3.6 \times 10^9$

iv. **1 día:**

a. $n = 24 \times 3600 \times 10^6$
 b. $n = 86400 \times 10^6$
 c. $n = 8.64 \times 10^{10}$

v. **1 mes:**

a. $n = 30 \times 24 \times 3600 \times 10^6$
 b. $n = 2592000 \times 10^6$
 c. $n = 2.592 \times 10^{12}$

vi. **1 año:**

a. $n = 12 \times 30 \times 24 \times 3600 \times 10^6$
 b. $n = 31104000 \times 10^6$
 c. $n = 3.1104 \times 10^{13}$

vii. **1 siglo:**

a. $n = 100 \times 12 \times 30 \times 24 \times 3600 \times 10^6$
 b. $n = 3110400000 \times 10^6$
 c. $n = 3.1104 \times 10^{15}$

4. $n \times \log_2(n)$:

i. **1 seg:**

- a. $n \times \log_2(n) = 10^6$
- b. Esta ecuación no tiene una solución algebraica directa, por lo que se debe resolver numéricamente o mediante aproximaciones.
- c. Se sabe que $n \leq n \times \log_2(n) \leq n^2$, por lo que n estará entre 10^3 y 10^6 .
- d. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 6.275 \times 10^4$.

ii. 1 min:

- a. $n \times \log_2(n) = 60 \times 10^6$
- b. n estará entre 7.746×10^3 y 6×10^7 .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 2.801 \times 10^6$.

iii. 1 hora:

- a. $n \times \log_2(n) = 3600 \times 10^6$
- b. n estará entre 6×10^4 y 3.6×10^9 .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 1.334 \times 10^8$.

iv. 1 día:

- a. $n \times \log_2(n) = 24 \times 3600 \times 10^6$
- b. n estará entre 2.939×10^5 y 8.64×10^{10} .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 2.755 \times 10^9$.

v. 1 mes:

- a. $n \times \log_2(n) = 30 \times 24 \times 3600 \times 10^6$
- b. n estará entre 1.61×10^6 y 2.592×10^{12} .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 7.187 \times 10^{10}$.

vi. 1 año:

- a. $n \times \log_2(n) = 12 \times 30 \times 24 \times 3600 \times 10^6$
- b. n estará entre 5.576×10^6 y 3.11×10^{13} .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 7.871 \times 10^{11}$.

vii. 1 siglo:

- a. $n \times \log_2(n) = 100 \times 12 \times 30 \times 24 \times 3600 \times 10^6$
- b. n estará entre 5.576×10^7 y 3.11×10^{15} .
- c. Usando el programa `metodo_numerico.py`, se obtiene que $n \approx 6.77 \times 10^{13}$.

5. n^2 :

i. 1 seg:

- a. $n^2 = 10^6$
- b. $n = \sqrt{10^6}$
- c. $n = 10^3$

ii. 1 min:

- a. $n = \sqrt{60} \times 10^3$
- b. $n = 7.746 \times 10^3$

iii. 1 hora:

- a. $n = \sqrt{60} \times 7.746 \times 10^3$
- b. $n = 7.746 \times 7.746 \times 10^3$
- c. $n = 60 \times 10^3$
- d. $n = 6 \times 10^4$

iv. 1 día:

- a. $n = \sqrt{24} \times 6 \times 10^4$
- b. $n = 4.898979486 \times 6 \times 10^4$
- c. $n = 29.393 \times 10^4$

d. $n = 2.9393 \times 10^5$

v. **1 mes:**

- a. $n = \sqrt{30} \times 2.9393 \times 10^5$
- b. $n = 5.477 \times 2.9393 \times 10^5$
- c. $n = 16.1046 \times 10^5$
- d. $n = 1.61046 \times 10^6$

vi. **1 año:**

- a. $n = \sqrt{12} \times 1.61046 \times 10^6$
- b. $n = 3.4641 \times 1.61046 \times 10^6$
- c. $n = 5.5757 \times 10^6$

vii. **1 siglo:**

- a. $n = \sqrt{100} \times 5.5757 \times 10^6$
- b. $n = 10 \times 5.5757 \times 10^6$
- c. $n = 55.757 \times 10^6$
- d. $n = 5.5757 \times 10^7$

6. 2^n :

i. **1 seg:**

- a. $2^n = 10^6$
- b. $\log(2^n) = \log(10^6)$
- c. $n \times \log(2) = \log(10^6)$
- d. $n \times \log(2) = 6$
- e. $n = \frac{6}{\log(2)}$
- f. $n = \frac{6}{0.301}$
- g. $n = 19.93$

ii. **1 min:**

- a. $n = \log_2(60) + 19.93$
- b. $n = 5.9069 + 19.93$
- c. $n = 25.84$

iii. **1 hora:**

- a. $n = \log_2(60) + 25.84$
- b. $n = 5.9069 + 25.84$
- c. $n = 31.75$

iv. **1 día:**

- a. $n = \log_2(24) + 31.75$
- b. $n = 4.5849 + 31.75$
- c. $n = 36.33$

v. **1 mes:**

- a. $n = \log_2(30) + 36.33$
- b. $n = 4.9069 + 36.33$
- c. $n = 41.24$

vi. **1 año:**

- a. $n = \log_2(12) + 41.2369$
- b. $n = 3.5849 + 41.2369$
- c. $n = 44.82$

vii. **1 siglo:**

- a. $n = \log_2(100) + 44.8218$

b. $n = 6.6439 + 44.8218$

c. $n = 51.46$

7. $n!:$

i. **1 seg:**

a. $n! = 10^6$

b. $n \approx 9$

ii. **1 min:**

a. $n! = 60 \times 10^6$

b. $n \approx 11$

iii. **1 hora:**

a. $n! = 3600 \times 10^6$

b. $n \approx 12$

iv. **1 día:**

a. $n! = 24 \times 3600 \times 10^6$

b. $n \approx 13$

v. **1 mes:**

a. $n! = 30 \times 24 \times 3600 \times 10^6$

b. $n \approx 15$

vi. **1 año:**

a. $n! = 365 \times 24 \times 3600 \times 10^6$

b. $n \approx 16$

vii. **1 siglo:**

a. $n! = 100 \times 365 \times 24 \times 3600 \times 10^6$

b. $n \approx 17$

2. Si el tiempo de ejecución en el mejor caso de un algoritmo, $t_m(n)$, es tal que $t_m(n) \in \Omega(f(n))$ y el tiempo de ejecución en el peor caso de un algoritmo, $t_p(n)$, es tal que $t_p(n) \in O(f(n))$, ¿Se puede afirmar que el tiempo de ejecución del algoritmo es $\Theta(f(n))$?

- Tiempo de ejecución del algoritmo en el mejor caso: $t_m(n) \in \Omega(f(n))$
- Tiempo de ejecución del algoritmo en el peor caso: $t_p(n) \in O(f(n))$

Sí, se puede afirmar que el tiempo de ejecución del algoritmo es $\Theta(f(n))$, porque por definición de la notación Θ , si una función está acotada inferiormente por $f(n)$ (es decir, pertenece a $\Omega(f(n))$) y acotada superiormente por $f(n)$ (es decir, pertenece a $O(f(n))$), entonces pertenece a $\Theta(f(n))$.

3. Un algoritmo tarda 1 segundo en procesar 1000 items en una máquina determinada. ¿Cuánto tiempo tomará procesar 10000 items si se sabe que

el tiempo de ejecución del algoritmo es n^2 ? ¿y si se sabe que es $n \times \log_2(n)$? ¿Qué se estaría asumiendo en todos los casos?

1. Caso 1: El tiempo de ejecución del algoritmo es n^2 .

i. Se puede usar una regla de tres simple.

ii. 1000 items → 1 segundo

iii. 10000 items → x

iv. Como se sabe que el tiempo de ejecución es n^2 , se puede plantear la siguiente ecuación:

$$v. x = \frac{(10000^2) \times 1}{1000^2}$$

$$vi. x = \frac{100000000 \times 1}{1000000}$$

$$vii. x = 100 \text{ segundos.}$$

2. Caso 2: El tiempo de ejecución del algoritmo es $n \times \log_2(n)$.

i. Se puede usar una regla de tres simple.

ii. 1000 items → 1 segundo

iii. 10000 items → x

iv. Como se sabe que el tiempo de ejecución es $n \times \log_2(n)$, se puede plantear la siguiente ecuación:

$$v. x = \frac{(10000 \times \log_2(10000)) \times 1}{(1000 \times \log_2(1000))}$$

$$vi. x = \frac{(10000 \times 13.287)}{(1000 \times 9.965)}$$

$$vii. x = \frac{132870}{9965}$$

$$viii. x \approx 13.333 \text{ segundos.}$$

En ambos casos se está asumiendo que no hay casos a analizar, es decir, que el tiempo de ejecución del algoritmo es siempre el mismo independientemente de la entrada que se reciba.

4. Un algoritmo toma n^2 días y otro n^3 segundos para resolver una instancia de tamaño n de un problema. Mostrar que el segundo algoritmo superará en tiempo al primero solamente en instancias que requieran más de 20 millones de años para ser resueltas.

- $t_1(n) = n^2$ días

- $t_2(n) = n^3$ segundos

- Convertir a la misma unidad de tiempo:

- $t_1(n) = n^2 \times 86400$ segundos (ya que un día tiene 86400 segundos)

- $t_2(n) = n^3$ segundos

- Se quiere encontrar el valor de n tal que $t_2(n) > t_1(n)$:

- $n^3 > n^2 \times 86400$

- Dividiendo ambos lados por n^2 (asumiendo $n > 0$):

- $n > 86400$

- Es decir, el segundo algoritmo solo se vuelve más lento cuando la entrada es estrictamente mayor a 86400.

- Reemplazando en la función: $t_1(86400) = (86400)^2$ días = 7464960000 días

- Usando un [conversor online de días a años](#) se tiene que 7464960000 días ≈ 20438366 años, es decir, más de 20 millones de años.

- Por lo tanto, se ha demostrado que el segundo algoritmo superará en tiempo al primero solamente en instancias que requieran aproximadamente más de 20 millones de años para ser resueltas.

5. ¿Cuáles y cuántas serían las operaciones elementales necesarias para multiplicar dos enteros n y m por medio del algoritmo enseñado en la escuela primaria? ¿Esta cantidad depende de la entrada? Justifique.

El algoritmo enseñado en la escuela primaria consiste en multiplicar cada dígito del segundo número por cada dígito del primer número y luego sumar los resultados parciales.

Sea n un número con d_n dígitos y m un número con d_m dígitos. Por ejemplo, si $n = 1998$, entonces $d_n = 4$, y si $m = 123$, entonces $d_m = 3$.

Una operación elemental en este contexto es una multiplicación de dos dígitos o una suma de dos dígitos. Por ejemplo, 8×3 es una operación elemental, al igual que $24 + 15$.

El algoritmo realiza los siguientes pasos:

1. Multiplicar cada dígito de m por cada dígito de n , uno por uno.
 - i. Como m tiene d_m dígitos y n tiene d_n dígitos, esto produce $(d_n \times d_m)$ multiplicaciones.
 - ii. Por ejemplo, si $n = 1998$ y $m = 123$, se realizan $4 \times 3 = 12$ multiplicaciones.
2. Una vez se hicieron todas las multiplicaciones, se tiene una lista de resultados parciales que deben sumarse de forma vertical.
 - i. Se deben sumar d_m filas, donde cada fila tiene aproximadamente d_n dígitos
 - ii. El total de sumas aproximado es $(d_n \times d_m)$
3. Por lo tanto, el total de operaciones elementales necesarias para multiplicar dos enteros n y m es aproximadamente:
 - i. Total de multiplicaciones: $d_n \times d_m$
 - ii. Total de sumas: $d_n \times d_m$
 - iii. Total de operaciones elementales: $2 \times (d_n \times d_m)$
4. Entonces el orden de complejidad del algoritmo es $O(d_n \times d_m)$. Si ambos números tienen la misma cantidad de dígitos d , entonces la complejidad es $O(d^2)$, es decir, cuadrática en función del número de dígitos.
5. La cantidad de operaciones depende directamente de la entrada, y en particular de la cantidad de dígitos de los números que se están multiplicando. A medida que los números crecen en tamaño (más dígitos), el número de operaciones elementales necesarias para realizar la multiplicación también aumenta.

6. Dar el tiempo de ejecución en función de n de los siguientes algoritmos y una $f(n)$ tal que el tiempo de ejecución pertenezca a $\Theta(f(n))$.

Determine si cada algoritmo o partes del mismo tiene casos de análisis (peor, mejor, etc).

a.

```
p ← 0
for i ← 1 to n do
    for j ← 1 to n2 do
        for k ← 1 to n3 do
            p ← p + 1
```

1. $1 + \sum_{i=1}^n \sum_{j=1}^{n^2} \sum_{k=1}^{n^3} 2$
2. $1 + \sum_{i=1}^n \sum_{j=1}^{n^2} (2n^3)$
3. $1 + \sum_{i=1}^n n^2 \cdot (2n^3)$
4. $1 + \sum_{i=1}^n (2n^5)$
5. $1 + n \cdot (2n^5)$
6. $1 + 2n^6$
7. El tiempo de ejecución en función de n del algoritmo es $t(n) = 1 + 2n^6$.
8. Se pide hallar una función $f(n)$ tal que $t(n) \in \Theta(f(n))$.
9. Como $t(n)$ es un polinomio de grado 6 y el orden Θ ignora las constantes multiplicativas y los términos de menor grado, se tiene que $t(n) \in \Theta(n^6)$, es decir $f(n) = n^6$.
10. El análisis del algoritmo no tiene casos especiales, ya que el tiempo de ejecución es siempre el mismo independientemente de la entrada que se reciba, por lo que no hay un mejor o peor caso.

b.

```
p ← 0
for i ← 1 to n do
    for j ← 1 to i do
        for k ← 1 to n do
            p ← p + 1
```

1. $1 + \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^n 2$
2. $1 + \sum_{i=1}^n \sum_{j=1}^i n \cdot 2$
3. $1 + \sum_{i=1}^n \sum_{j=1}^i 2n$
4. $1 + \sum_{i=1}^n i \cdot 2n$
5. Usamos la equivalencia $\sum_{i=1}^n i = \frac{n(n+1)}{2}$
6. $1 + \frac{n(n+1)}{2} \cdot 2n$

$$7. 1 + \frac{n^2+n}{2} \cdot 2n$$

$$8. 1 + (n^2 + n) \cdot n$$

$$9. 1 + n^3 + n^2$$

10. El tiempo de ejecución en función de n del algoritmo es $t(n) = 1 + n^3 + n^2$.

11. Se pide hallar una función $f(n)$ tal que $t(n) \in \Theta(f(n))$.

12. Como $t(n)$ es un polinomio de grado 3 y el orden Θ ignora las constantes multiplicativas y los términos de menor grado, se tiene que $t(n) \in \Theta(n^3)$, es decir $f(n) = n^3$.

13. El análisis del algoritmo no tiene casos especiales, ya que el tiempo de ejecución es siempre el mismo independientemente de la entrada que se reciba, por lo que no hay un mejor o peor caso.

7. Definir y analizar el tiempo de ejecución de la multiplicación de una matriz triangular inferior por una matriz completa (en la que todos sus elementos pueden ser diferentes de 0). ¿Qué formas tiene de hallar $t(n)$? ¿De cuántas formas podría encontrar la pertenencia a $O()$ o a $\Theta()$ del algoritmo?

1. Una matriz triangular inferior es una matriz cuadrada en la que todos los elementos por encima de la diagonal principal son cero. Su forma general es:

$$\begin{pmatrix} a_{1,1} & 0 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & \cdots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{pmatrix}$$

2. Una matriz completa, por otro lado, es una matriz en la que todos sus elementos pueden ser diferentes de cero. Su forma general es:

$$\begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,n} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & b_{n,3} & \cdots & b_{n,n} \end{pmatrix}$$

3. La multiplicación de matrices se realiza tomando cada fila de la primera matriz y multiplicándola por cada columna de la segunda matriz, sumando los productos correspondientes. Como la primera matriz es triangular inferior, muchos de sus elementos son cero, lo que reduce la cantidad de multiplicaciones necesarias.

4. Para calcular el tiempo de ejecución $t(n)$ de la multiplicación de una matriz triangular inferior por una matriz completa, se puede observar que:

i. En la primera fila de la matriz triangular inferior, solo hay un elemento distinto de cero, por lo que se realizan n multiplicaciones.

ii. En la segunda fila, hay dos elementos distintos de cero, por lo que se realizan $2n$ multiplicaciones.

iii. En la tercera fila, hay tres elementos distintos de cero, por lo que se realizan $3n$ multiplicaciones.

iv. Este patrón continúa hasta la última fila, donde hay n elementos distintos de cero, por lo que se realizan $nn = n^2$ multiplicaciones.

v. Por lo tanto, el total de multiplicaciones necesarias es: $T(n) = n + 2n + 3n + \cdots + nn = n(1 + 2 + 3 + \cdots + n) = (n \cdot \sum_{i=1}^n i) = n \cdot (\frac{n(n+1)}{2}) = n \cdot (\frac{\frac{n^2+n}{2}}{2}) = \frac{n^3+n^2}{2}$.

vi. Así, el tiempo de ejecución en función de n es $t(n) = \frac{n^3+n^2}{2}$.

vii. Se pide hallar una función $f(n)$ tal que $t(n) \in \Theta(f(n))$.

viii. Como $t(n)$ es un polinomio de grado 3 y el orden Θ ignora las constantes multiplicativas y los términos de menor grado, se tiene que $t(n) \in \Theta(n^3)$, es decir $f(n) = n^3$.

Lo anterior es el método analítico de hallar $t(n)$. Otra forma de hallarlo es planteando el pseudocódigo correspondiente al problema, y luego convirtiendo sus bucles en sumatorias, similar al ejercicio 6.

8. ¿Encuentra algún inconveniente para analizar las iteraciones `while` y `repeat` como recurrencias?

Las iteraciones `while` y `repeat` suelen ser complicadas de analizar como recurrencias ya que su número de iteraciones puede variar según condiciones que dependen de la entrada o del estado interno del algoritmo. A diferencia de los bucles `for`, donde el número de iteraciones es generalmente conocido de antemano, los bucles `while` y `repeat` pueden continuar ejecutándose hasta que se cumpla una condición específica, lo que puede hacer que el análisis sea más complejo.

9. Considerar las matrices $A, B, C \in \mathbb{R}^{(n \times n)}$, y la notación tal que $X_{i,j}$, con $1 \leq i, j \leq 2$ y X cualquiera de las matrices A, B o C , identifica una de las cuatro submatrices de orden $\frac{n}{2}$

a. Dar el orden del tiempo de ejecución del algoritmo D&C que se describe con las ecuaciones:

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$

$$C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$$

$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$

$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}$$

¿Sería necesario definir algo más?

Se plantea una forma de multiplicar matrices distinta a la tradicional. En el enfoque dado, la estrategia es de tipo Divide & Conquer, donde cada matriz, A y B , se divide en cuatro submatrices de tamaño $\frac{n}{2} \times \frac{n}{2}$. Luego, se realizan multiplicaciones y sumas de estas submatrices para obtener las submatrices de la matriz resultante C . Por ejemplo, si $n = 16$, entonces cada submatriz tendrá un tamaño de 8×8 .

Para dar el orden del tiempo de ejecución del algoritmo D&C, se puede armar la ecuación de recurrencia $t(n)$. Como el problema es de tipo Divide & Conquer, la receta que se usa es la de reducción por división, que tiene la fórmula general: $t(n) = a \cdot t(\frac{n}{b}) + f(n)$, donde:

- a es la cantidad de llamadas recursivas, es decir, subproblemas.
- b es el factor por el cual se divide el tamaño del problema en cada llamada recursiva.
- $f(n)$ son operaciones extra a las llamadas recursivas, normalmente el costo de combinar los resultados parciales.

Analizando las ecuaciones dadas, tenemos 8 multiplicaciones de submatrices (cada una de tamaño $\frac{n}{2} \times \frac{n}{2}$) y 4 sumas de matrices (cada una de tamaño $\frac{n}{2} \times \frac{n}{2}$). Por lo tanto: $a = 8$, $b = 2$ y $f(n) = O(n^2)$ ya que sumar dos matrices de tamaño $\frac{n}{2} \times \frac{n}{2}$ toma tiempo proporcional a $(\frac{n}{2})^2 = \frac{n^2}{4}$, y hay 4 sumas, por lo que se tiene $4 \cdot \frac{n^2}{4} = n^2$.

La ecuación de recurrencia queda entonces: $t(n) = 8 \cdot t(\frac{n}{2}) + n^2$.

$f(n) \in O(n^k)$, con $k = 2$.

Se tiene que $a > b^k$ porque $8 > 2^2$. Entonces, por teorema maestro, el tiempo de ejecución es: $t(n) \in O(n^{\log_b(a)}) = O(n^{\log_2(8)}) = O(n^3)$.

Por lo tanto, el orden del tiempo de ejecución del algoritmo D&C es $O(n^3)$.

b. Buscar el algoritmo de Strassen, dar su definición en función de las submatrices $X_{i,j}$ anteriores y dar su orden de tiempo de ejecución.

El algoritmo de Strassen es otro algoritmo de multiplicación de matrices que tiene como ventaja un menor tiempo de ejecución en comparación con el método tradicional. Su definición en función de las submatrices $X_{i,j}$ es la siguiente:

- $M_1 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$
- $M_2 = (A_{2,1} + A_{2,2}) \times B_{1,1}$
- $M_3 = A_{1,1} \times (B_{1,2} - B_{2,2})$
- $M_4 = A_{2,2} \times (B_{2,1} - B_{1,1})$
- $M_5 = (A_{1,1} + A_{1,2}) \times B_{2,2}$
- $M_6 = (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2})$
- $M_7 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$

Luego, las submatrices de la matriz resultante C se calculan como:

- $C_{1,1} = M_1 + M_4 - M_5 + M_7$
- $C_{1,2} = M_3 + M_5$
- $C_{2,1} = M_2 + M_4$
- $C_{2,2} = M_1 - M_2 + M_3 + M_6$

Para obtener el orden de tiempo de ejecución del algoritmo de Strassen, se puede armar la ecuación de recurrencia $t(n)$. En este caso, se realizan 7 multiplicaciones de submatrices (cada una de tamaño $\frac{n}{2} \times \frac{n}{2}$) y varias sumas y restas de matrices (cada una de tamaño $\frac{n}{2} \times \frac{n}{2}$). Por lo tanto: $a = 7$, $b = 2$ y $f(n) = O(n^2)$.

La ecuación de recurrencia queda entonces: $t(n) = 7 \cdot t(\frac{n}{2}) + n^2$.

Usando teorema maestro, se tiene que $a > b^k$ porque $7 > 2^2$. Entonces, el tiempo de ejecución es: $t(n) \in O(n^{\log_b(a)}) = O(n^{\log_2(7)})$. Calculando $\log_2(7) \approx 2.807$, se tiene que el tiempo de ejecución es $O(n^{2.807})$.

c. Comparar los dos algoritmos de multiplicación de matrices. ¿Los dos son algoritmos D&C? ¿Alguno de los dos es "mejor" que el otro en cuanto a tiempo de ejecución?

Claramente ambos algoritmos son de tipo Divide & Conquer, ya que ambos dividen las matrices en submatrices más pequeñas, las multiplican, y suman los resultados parciales para obtener la matriz resultante.

El algoritmo de Strassen es mejor que el algoritmo D&C tradicional si hablamos estrictamente del tiempo de ejecución, ya que tiene un orden de tiempo de ejecución de $O(n^{2.807})$, que es menor que el orden de tiempo de ejecución del algoritmo D&C tradicional, que es $O(n^3)$. Esto implica que para matrices de gran tamaño, el algoritmo de Strassen será más eficiente y rápido en comparación con el método tradicional. Sin embargo, es importante tener en cuenta que el algoritmo de Strassen puede ser más complejo de implementar y además usa mucha más memoria debido a las múltiples submatrices y operaciones adicionales involucradas.