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) viene a ser una medida del rendimiento de una función. El valor que nos da corresponde a la cantidad de unidades de tiempo que tarda en ser resuelto un problema especifico. Estas unidades, asumimos que se corresponden con un microsegundo cada una (0.000001 seg)

Teniendo esto en cuenta tomemos el caso uno como ejemplo.

t(n) = log2(n) = 1 mic

Entonces ¿Que tan grande puede ser N como para corresponderse con esta asignación temporal?

Bueno, n es recibido como parámetro en la función, pensemos que en un logaritmo uno puede obtener el valor n buscando su equivalencia en potencia.



log2(n) = 1 mic <=> 21 = n

21 = n

2 = n

t(n) = log2(n) = 1 mic <=> 2 = n

¿Cuanto para 1 seg?

log2(n) = 1 seg = 1000000 mic <=> 2 1000000 = n

¿Cuanto para x unidades?

log2(n) = x unidades <=> 2 x = n

-----

Caso 2:

t(n) = √n = 1 mic <=> n=12 <=> n=1

¿Para 1 seg?

t(n) = √n = 1 seg <=> √n = 1000000 mic <=> n=10000002

¿Para x seg?

t(n) = √n = x unidades <=> n = x2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| t(n) | 1 seg | 1 min | 1 hora | 1 día | 1 mes | 1 año | 1 siglo |
| log2(n) | 21000000 | 2x60 | 2x3600 | 2x86400 | 2x2592000 | 2x946080000 | 2x9608x10^10 |
| √n | 10000002 | x602 | x36002 | x864002 | x25920002 | x9460800002 | x9608x10^102 |
| n | 1000000 |  |  |  |  |  |  |
| n x log2(n) |  |  |  |  |  |  |  |
| n2 | √1000000 |  |  |  |  |  |  |
| 2n | log2(1000000) |  |  |  |  |  |  |
| n! | 9 |  |  |  |  |  |  |

2) Si el tiempo de ejecución en el mejor caso de un algoritmo, tm(n), es tal que tm(n) ∈ Ω(f(n)) y el tiempo de ejecución en el peor caso de un algoritmo, tp(n), es tal que tp(n) ∈ O(f(n)), ¿Se puede afirmar que el tiempo de ejecución del algoritmo es Θ(f(n))?

Pensemos en lo que dice el enunciado:

a- *en el mejor caso tm(n) ∈ Ω(f(n))*

El mejor caso es el que tiene el valor mínimo de tiempo. El Umbral determina el orden de crecimiento mínimo de la función, así si el peor caso cumple el umbral todos los demás casos también lo harán.

Misma lógica para la segunda sentencia:

b- *el peor caso tp(n) ∈ O(f(n))*

Si el caso máximo en tiempo tiene un orden de crecimiento máximo de f(n) entonces todos los casos inferiores también lo tendrán.

Con a b tenemos que en esencia todo el algoritmo independientemente del caso va a respectar *Ω(f(n)) y O(f(n)).*

Recordando las propiedades sabemos que t(n) ∈ Θ(f(n)) ↔ t(n) ∈ Ω(f(n)) ∧ t(n) ∈ O(f(n))

∴ tm(n) ∈ Θ(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 n2? ¿Y si se sabe que es n × log2n?

¿Que se estaría asumiendo en todos los casos?

En 1 segundo procesa 1000 items.

T(1000) = 10002 = 1000000 => Esto tiene sentido si fueran 1000000 mic = 1 seg

¿Cuánto tomará procesar 10000 items si se sabe t(n)=n2?

T(10000)=100002 mic=100000000 mic = 100 seg

∴t(10000)=100 seg

¿Y si se sabe que t(n)= n × log2n?

Primero identificamos cuanto tiempo le corresponde al caso base

t(1000)= 1000 × log21000 = 1000 x (log101000 / log102) = 1000 x (3 / log102) = 9965,78 = 1 seg

Osea que cada 9965,78 unidades de lo que diga t(n) significara que a la función le corresponde +1seg.

Ahora si, caso con n = 10000

t(10000)= 10000 × log2(10000)=10000x13,28771237954945=132877,1237

132877,1237 / 9965,78 = 13.33 seg

∴t(10000)=13.33 seg

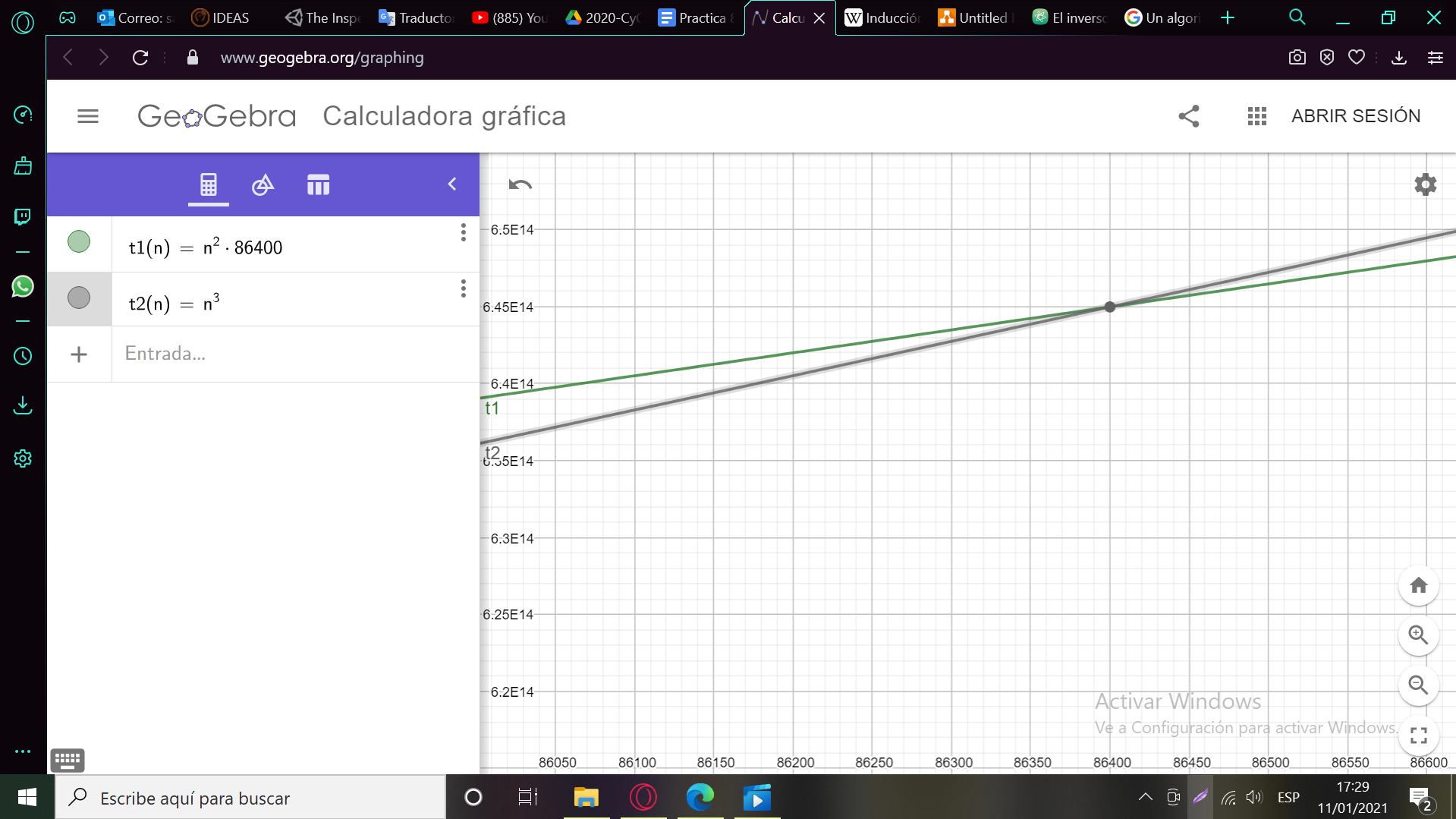
4) Un algoritmo toma n2 días y otro n3 segundos para resolver una instancia de tamaño n de un problema. Mostrar que el segundo algoritmo superará en tiempo al primero solamente en instancias que requieran más de 20 millones de años para ser resueltas.

Tenemos:

* Una instancia de tamaño n
* Un algoritmo que se toma n2 días para resolverla, también podríamos decir que se toma 86400.n2 segundos.
* Otro algoritmo que se toma n3 segundos para resolverla

Debo mostrar que el segundo algoritmo superará en tiempo al primero solamente en instancias que requieran mas de 20 millones de años para ser resueltas.

si vemos donde intersectan nuestras dos rectas tenemos que se chocan en:



Osea en (86400,6.45x1014).

Se logra apreciar que solo existe esta intersección, por lo que antes de esta t1 siempre tardo mas que t2, pero despues de esta t2 tarda mas que t1, según la cuenta entonces para problemas que requieran mas de 6.45x1014 segundos para resolverse será mejor t2, pero si es menos tiempo el requerido será mejor usar t2, a cuantos año se traduce esto, sería 6.45x1014 segundos≈20452815.83 años osea aproximadamente 20 millones de años.

∴ Con esto queda demostrado que el segundo algoritmo superará en tiempo al primero solamente en instancias que requieran mas de 20 millones de años para ser resueltas.

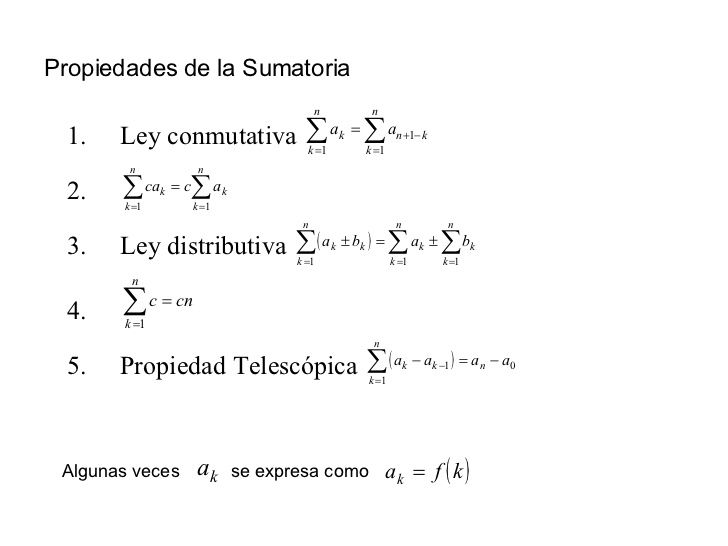
5) ¿Cuáles y cuántas serían las operaciones elementales necesarias para multiplicar dos enteros n y m por medio del algoritmo enseñado en la escuela primaria? ¿Esta cantidad depende de la entrada? Justifique.

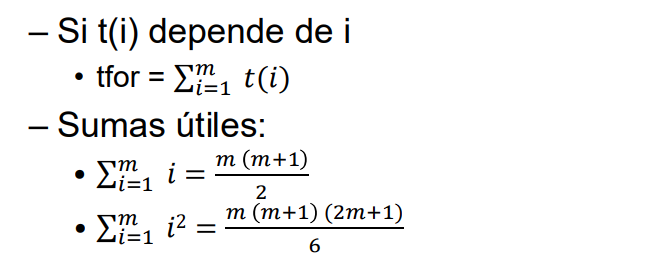
No estoy seguro cual seria este algoritmo, pero voy a describir lo que creo que es:

1. tomamos el dígito de mas a la derecha de m
2. lo multiplicamos por cada dígito de n anotando el resultado abajo, si el resultado de la multiplicación tiene mas de un dígito y este no era el último dígito de n, entonces para la próxima multiplicación sumamos tanto como el dígito sobrante indique, si era el último anotamos todos los dígitos en el resultado
3. repetimos para los demás dígitos de m, cada resultado que nos de lo escribimos pero le agregamos ceros a la derecha según cuantos ceros se les agrego al anterior + 1, osea, al primero le agregamos 0 ceros, al segundo 1 cero, al tercero 2 ceros y asi, hasta que m no0 tenga dígitos
4. sumamos todos estos resultados fila por fila fi los resultados tienen mas de 1 digito el sobrante sumará en la siguiente suma, si era la última suma entonces se anota el numero entero.

Obviamente la cantidad de operaciones dependerá del tamaño de la entrada, mientras mas dígitos tienen las entradas, mas operaciones se deben hacer y mientras mas grandes sean estos dígitos mas acarreos se deberán realizar.

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

Dejo esto acá para que tenga sentido lo que hago más adelante:  


a)

p ← 0

for i ← 1 to n do

for j ← 1 to n2 do

for k ← 1 to n3 do

p ← p + 1

Primero intentare encontrar el tiempo de ejecución en función de n del algoritmo:

podemos decir que

t(n) = c1 + sumatoriaDe1aNde(sumatoriaDe1aN2de(sumatoriaDe1aN3de(c2))) =

= c1 + sumatoriaDe1aNde(sumatoriaDe1aN2de(n3\*c2)) =

= c1 + sumatoriaDe1aNde(n2\*n3\*c2) =

= c1 + n1\*n2\*n3\*c2 =

= c1 + n6\*c2 =

= c2\*n6 + c1

∴ Tenemos que el tiempo de ejecución de este algoritmo en función de n es t(n)=c2\*n6+c1

Ahora debo buscar una f(n) tal que el tiempo de ejecución pertenezca a Θ(f(n)):

t(n) ∈ Θ(f(n)) sii t(n) ∈ O(f(n)) y t(n) ∈ Ω(f(n))

Voy a demostrar que t(n) ∈ Θ(f(n)) para f(n) siendo f(n)=n6 y t(n)=c2\*n6+c1.

* O(f(n)) = {t:N → R+ / ∃ c ∈ R+, n0 ∈ N tq t(n) ≤ c f(n), n ≥ n0}

Esto obviamente es cierto ya que podemos utilizar como c =c2+c1 y n0=1 y siempre se cumplirá ya que es evidente que c2\*n6+c1 ≤ c\*n6 no sabemos cuánto vale c1, pero podemos ajustar el valor de c según cuanto valga c1 para cumplir lo que se requiere.

* Ω(f(n)) = {t:N → R+ / ∃ c ∈ R+, n0 ∈ N tq t(n) ≥ c f(n), n ≥ n0}

Esto obviamente es cierto ya que podemos utilizar como c a c2 y n0=1 y siempre se cumplirá ya que es evidente que c2\*n6+c1 ≥ c2\*n6.

∴ Por A y B tenemos que t(n) ∈ Θ(f(n))

b)

p ← 0

for i ← 1 to n do

for j ← 1 to i do

for k ← 1 to n do

p ← p + 1

Primero intentare encontrar el tiempo de ejecución en función de n del algoritmo:

podemos decir que

t(n)= c1+sumatoria1aNconi(sumatoria1aIiconj(sumatoria1aINconk(c2))) =

= c1+sumatoria1aNconi(sumatoria1aIiconj(n\*c2\*1)) =

= c1+(n\*c2)\*sumatoria1aNconi(sumatoria1aiconj(1)) =

= c1+(n\*c2)\*sumatoria1aNconi(i\*1) =

= c1+(n\*c2)\*sumatoria1aNconi(i) =

= c1+(n\*c2)\*((n\*(n+1))/2) =

= c1+c2\*n\*(n2+n)/2) =

= c1+(c2/2)\*(n3+n2) =

= c1+(c2/2)\*n3+(c2/2)\*n2) =

= (c2/2)\*n3+(c2/2)\*n2) + c1

∴ Tenemos que el tiempo de ejecución de este algoritmo en funcion de n es t(n)= (c2/2)\*n3+(c2/2)\*n2) + c1.

Ahora debo buscar una f(n) tal que el tiempo de ejecución pertenezca a Θ(f(n)):

t(n) ∈ Θ(f(n)) sii t(n) ∈ O(f(n)) y t(n) ∈ Ω(f(n))

Se puede demostrar que t(n) ∈ Θ(f(n)) para f(n) siendo f(n)=n3 y t(n)=(c2/2)\*n3+(c2/2)\*n2) + c1.

Utilizando la regla de limites podemos hacer

lim ((c2/2)\*n3+(c2/2)\*n2) / n3

→ lim c2/2+(c2/2)/n

->c2/2+(c2/2)/inf

->c2/2+0 = c2/2 entonces ∈ R+

∴ t(n) ∈ Θ(f(n))

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 Θ() del algoritmo?

En el foro de ideas encontré el siguiente algoritmo para la situación, según lo que respondió el profe a la consulta es correcto:

for j=1 to n

for i=1 to n

c[ i ][ j ] = 0

for k=1 to i

c[ i ][ j ] = A[ i ][ k ]\*B[ k ][ j ] + c[ i ][ j ]

* ¿Qué formas tiene de hallar t(n)?

Sinceramente no se me ocurren varias formas, la única que conozco es esta:

t(n)= Sumjde1aNde(Sumide1aNde(c1+Sumkde1aide(c2))) =

= Sumjde1aNde(Sumide1aNde(c1+i\*c2)) =

= n\*(Sumide1aNde(c1+i\*c2)) =

= n\*(Sumide1aNde(c1)+Sumide1aNde(i\*c2)) =

= n\*(n\*c1+Sumide1aNde(i\*c2)) =

= n\*(c1\*n+c2\*Sumide1aNde(i)) =

= n\*(c1\*n+c2\*(n\*(n+1)/2)) =

= n\*(c1\*n+c2/2\*(n2+n)) =

= n\*(c1\*n+c2\*n2/2+c2\*n/2) =

= n\*(c2\*n2/2+c1\*c2\*n/2) =

= c2\*n3/2+c1\*c2\*n2/2 =

= c2\*n3/2+c3\*n2/2

∴ t(n) = c2\*n3/2+c3\*n2/2

No se si el método es correcto, pero es lo que se debería hacer, podría tener mas seguridad si enseñaran ejemplos de esto en la teoría.

Capaz otro método viable seria con recurrencias

* ¿De cuántas formas podría encontrar la pertenencia a O() o a Θ() del algoritmo?

Podría demostrar todo paso a paso, pero es mas fácil usar el teorema del límite

Uso f(n) = n3:

limn->inf t(n)/f(n) ∈ R+ ⇒ t(n) ∈ Θ(f(n))

limn->inf t(n)/f(n) =

= limn->inf ((c2\*n3/2+c3\*n2/2)/(n3)) =

= limn->inf ((c2\*n3/2)/(n3)+(c3\*n2/2)/(n3)) =

= limn->inf ((c2/2)+(c3/2)/n) =

= (c2/2)+(c3/2)/inf =

= (c2/2)+0 =

= c2/2

Sabemos que c2/2 ∈ R+

∴ t(n) ∈ Θ(f(n))

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

Lo único que logré encontrar en las teorías es que son complejas y existen algoritmos para los cuales no es posible hallar una función o definir una recurrencia.

9) Considerar las matrices A, B, C ∈ IR(n×n), y la notación tal que Xi,j , con 1 ≤ i, j ≤ 2 y X cualquiera de las matrices A, B o C, identifica una de las cuatro submatrices de orden n/2.

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

C1,1 = A1,1 × B1,1 + A1,2 × B2,1

C1,2 = A1,1 × B1,2 + A1,2 × B2,2

C2,1 = A2,1 × B1,1 + A2,2 × B2,1

C2,2 = A2,1 × B1,2 + A2,2 × B2,2

¿Sería necesario definir algo más?

Según la teoría hay que usar las recetas de recurrencia.

Con respecto a lo que falta definir es obvio que sí, falta definir el caso base, podría definirlo así:

| t(n)= c1 n≤1

<|

| t(n)= --- n>1

Para usar las recetas debemos identificar 3 cosas:

a cantidad de llamadas recursivas

b constante relacionada con la reducción de tamaño

k grado del polinomio de trabajo extra

En el algoritmo que nos dan podemos identificar:

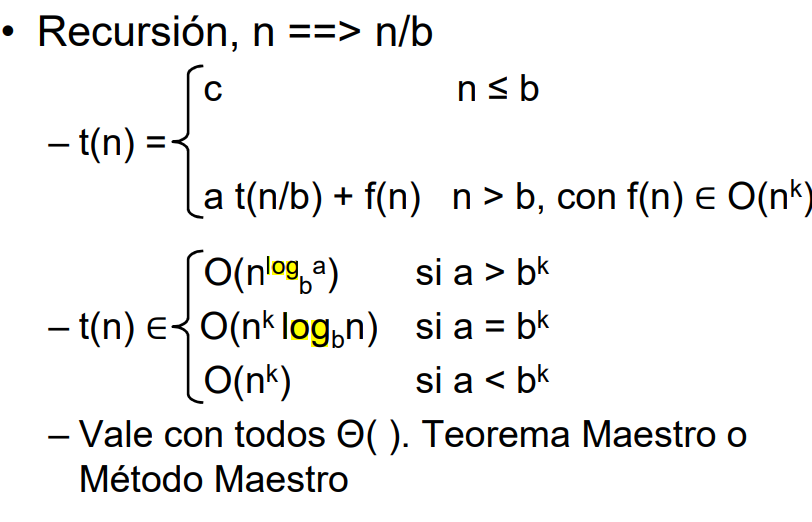
a = 8

b = 2

k = 2 (por que el trabajo extra en este caso se basa en la suma de matrices cuadradas)

CREO que podría ser sera ya que solo hay 8 operaciones base, aunque depende de como se plantee la solución.

Así usando la receta para división de tamaño tenemos que:



a>bk osea 8>22 <-> 8>4 así que O(nlog b(a)) osea t(n) es orden O(nlog 2(8)) que haciendo las cuentas nos da que t(n) es O(n3)

∴ t(n) es O(n3)

b) Buscar el algoritmo de Strassen, dar su definición en función de las submatrices Xi,j anteriores y dar su orden de tiempo de ejecución.

Lo encontré, aparentemente es muy conocido, y en wikipedia encontre la definicion en funcion de las submatrices:

M1=(A1,2 - A2,2).(B2,1 + B2,2)

M2=(A1,1 + A2,2).(B1,1 + B2,2)

M3=(A1,1 - A2,1).(B1,1 + B2,1)

M4=(A1,1 + A1,2).B2,2

M5=A1,1 .(B1,2 -B2,2)

M6=A2,2.(B2,1 - B1,1)

M7=(A2,1 + A2,2).B1,1

| 1 n≤2

t(n)<

| 7\*t(n/2) + n2 n>2

Asi que su orden es O(nlog b(a)) osea t(n) es orden O(nlog 2(7)) que haciendo las cuentas nos da que t(n) es aproximadamente O(n2.807354922057604)

∴ t(n) es O(n2.807354922057604)

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?

Ambos son algoritmos divide y conquista pero el algoritmo Strassen tiene un orden menor y su tiempo de ejecución no será menor necesariamente siempre, ya vimos esto en un punto anterior, si el Strassen mide lo que tarda en días y el normal en segundos, solo tardaría menos para instancias muy grandes del problema, lo cual usualmente no suele convenir.

∴ No necesariamente es mejor en tiempo de ejecución y ambos son algoritmos D&C,