75.41 Algoritmos y Programación II
Análisis de Algoritmos

Dr. Mariano Méndez [1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

23 de junio de 2020

1. Introducción

En este apunte se verán y recopilarán algunos ejercicios de análisis de algoritmos.

2. Análisis de Algoritmos Secuenciales

Ejercicio

Sea la función long suma (int numeros) que calcula la suma de los n primeros números. Calcular detalladamente su orden de complejidad en notación big-O:

```
long suma (int numeros){
    long acumulador=0;

    for(int i=1; i<=n; i++)
    acumulador+=n;

    return acumulador;
}

```

Solución del ejercicio

Esta función es un bloque de tres instrucciones ejecutables: S1, S2 y S3:

```
/*S1*/  long acumulador=0;

/*S2*/  for(int i=1; i<=n; i++)
    acumulador+=n;

/*S3*/  return acumulador;

```

Aplicando las reglas del modelo de análisis computacional propuesto, se tiene que se está analizando una secuencia de instrucciones. Por ende, la regla de cálculo del tiempo de ejecución de una secuencia de instrucciones, se basa en aplicar el teorema de aditividad:

$$O(\{S1, S2, S3\}) = max\{O(S1), O(S2), O(S3)\}$$

De esta forma:

$$O(S1) = O(S3) = O(1)$$

Sólo falta determinar $O(S2)$, para ello se estudiará el peor de los casos posibles que puede ocurrir en la iteración $S2$, esto es (el caso normal, justo en este caso) cuando $i = n$:

$$O(S2) = n$$

Ya que se debe iterar por todos los casos desde el 1 hasta n. Finalmente se obtiene:

$$O(\{S1, S2, S3\}) = max\{O(1), O(n), O(1)\}$$

por ende el $\mathbf{T(n) = O(n)}$

Solución del ejercicio

Otra forma de realizar el análisis es cuantificar el número total de instrucciones que ejecuta el algoritmo:

- S1 ejecuta una asignación;
- S2 es una estructura de control for del lenguaje C que ejecuta :
  - una asignación: i=0 se ejecuta 1 vez
  - una comparación: esta se realiza n veces
  - una suma y una asignación: que también se realizan n veces.

  Además la instrucción dentro del for ejecuta una suma y una asignación, que se ejecutan n veces.
- S3 ejecuta una instrucción primitiva de retorno

Contando las instrucciones se tiene:

$$1 + 1 + n + n + n + n = 2 + 4n + 4n + 2$$

Entonces si $\mathbf{T(n) = 4n + 2}$ lo que implica que $\mathbf{T(n) = O(n)}$

Ejercicio

Sea la función int suma_pot_tres(int tope) que calcula la suma de los n primeros números. Calcular detalladamente su orden de complejidad en notación big-O:

```
1 int suma_pot_tres (int n){
2
3     int contador=1;
4     for(i=1; i<n; i*=3)
5         contador++;
6     return contador;
7 }
```

Solución del ejercicio

Esta función cuenta con tres instrucciones ejecutables S1,S2,S3:

```
1 int suma_pot_tres (int n){
2
3 /*S1*/  int contador=1;
4 /*S2*/  for(i=1; i<n; i*=3)
5         contador++;
6 /*S3*/  return contador;
7 }
```

Teniendo en cuenta las reglas del modelo de análisis computacional propuesto, se tiene que se está analizando una secuencia de instrucciones. Por ende, la regla de cálculo del tiempo de ejecución de una secuencia de instrucciones, se basa en aplicar el teorema de aditividad:

$$O(\{S1, S2, S3\}) = max\{O(S1), O(S2), O(S3)\}$$

De esta forma:

$$O(S1) = O(S3) = O(1)$$

Sólo falta determinar $O(S2)$, para ello se estudiará el peor de los casos posibles que puede ocurrir en la iteración $S2$, esto es (el caso normal, justo en este caso) cuando $i < n$. Para ello hay que determinar la cantidad de iteraciones que hace el

i	contador	iteraciones
1	1	1
2	2	2
4	3	3
8	4	4
16	5	5
32	6	6
64	7	7

lo que se ve en la tabla es que la cantidad de potencias de 2 equivale a la cantidad de iteraciones que se realizará en el ciclo for, por ende:

$$n = 2^x$$

Donde x es la cantidad de potencias de dos en el intervalo 1..N. Tomando logaritmos en ambos lados de la igualdad:

$$log\ (n) = log(2^x)$$

$$log\ (n) = log(2) * x$$

$$log\ (n) = x$$

la cantidad de iteraciones de S2 es x, $log\ (n)$. Por ende $\mathbf{O(S2) = log\ (n)}$. Finalmente se obtiene:

$$O(\{S1, S2, S3\}) = max\{O(1), O(log\ (n)), O(1)\}$$

por ende el $\mathbf{T(n) = O(log\ (n))}$

Ejercicio

Calcular el tiempo de ejecución del siguiente algoritmo en el peor de los casos:

```
1  void comple( int n){
2  int i,k;
3
4  /*S1*/  i=1;
5  /*S2*/  k=1;
6  /*S3*/  while (i<n){
7              if (i==k){
8                  printf("\nk=%i i=",k);
9                  k++;
10                 i=1;
11             }
12             printf("%i",i);
13             i++;
14         }
15 /*S4*/  printf("\n");
16 }
```

Solución del ejercicio

En este algoritmo hay 4 instrucciones secuenciales consecutivas. Aplicando la regla de análisis de algoritmos para una secuencia de instrucciones consecutivas: **El tiempo de ejecución de instrucciones consecutivas es el máximo de entre ambas instrucciones**. Es decir que:

$$T_{S1}(n) + T_{S2}(n) + T_{S3}(n) + T_{S4}(n) = max(O(f(n)), O(g(n)), O(h(n)), O(i(n)))$$

el tiempo de ejecución de la función comple() es el máximo de los tiempos de ejecución de cada una de sus 4 instrucciones. Según el modelos de análisis computacional utilizado, los tiempos de S1, S2 y S4 son exactamente el mismo por ende se consideran $O(1)$, ya que todas se consideran acciones simples cuyo tiempo de duración es el mismo.

Con lo cual la ecuación anterior queda:

$$T_{S1}(n) + T_{S2}(n) + T_{S3}(n) + T_{S4}(n) = max(O(1), O(1), O(h(n)), O(1))$$

la incógnita del ejercicio es cuanto vale $O(h(n))$. Entonces, se debe analizar con detalle S3, que es una iteración con un bloque de instrucciones:

```
...
/*S3*/  while (i<n){
            if (i==k){
                printf("\nk=%i i=",k);
                k++;
                i=1;
            }
            printf("%i",i);
            i++;
        }
...
```

a simple vista no es posible determinar cual es el tiempo de ejecución de S4, muchos dirían que es $O(n)$, pero para tener un grado de certeza lo mejor es hacer el seguimiento, en este caso es sencillo ya que i,k valen 1 ambos. Entonces es posible hacer el seguimiento del loop:

	# iteraciones
k=1, i=1	1
k=2, i=1,2	2
k=3, i=1,2,3	3
k=4, i=1,2,3,4	4
k=5, i=1,2,3,4,5	5
...	
k=n-2, i=1,2,3,4,5,....,n-2	n-2
k=n-1, i=1,2,3,4,5,....,n-1,n	n-1
	= 1+2+3+....+ n-1

Esto se puede ver en una corrida de la función con n=10:

```
k=1 i=1
k=2 i=12
k=3 i=123
k=4 i=1234
k=5 i=12345
k=6 i=123456
k=7 i=1234567
k=8 i=12345678
k=9 i=123456789
```

Por ende el total del número de iteraciones es :

$$1+2+3+4+5+...+n-2+n-1=$$

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

por lo cual $O(h(n)) = \frac{n^2}{2} - \frac{n}{2}$ por ende cuando se toma el limite cuando n tiende a infinito el término lineal no aporta nada y el $1/2$ que multiplica al n^2 tampoco, en consecuencia $O(h(n)) = n^2$, retomando y sustituyendo :

$$T_{S1}(n) + T_{S2}(n) + T_{S3}(n) + T_{S4}(n) = max(O(1), O(1), O(n^2), O(1))$$

con lo cual el tiempo de ejecución del mismo está acotado superiormente por $h(n) = n^2$, por ende la complejidad algorítmica del mismo es :

$$O(n^2)$$

Ejercicio

Demostrar que $f(x) = x + 7$ es $O(x)$ encontrar c y k.

Solución del ejercicio

En en primer lugar se debe realizar el análisis de la función, para tener una idea de lo que se esta buscando. Y recordad que la definición de $\mathbf{O(g(n))}$: Se dice que $\mathbf{T(n) = f(n) = O(g(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * g(n)}$, para todo $\mathbf{n \geq n_0}$

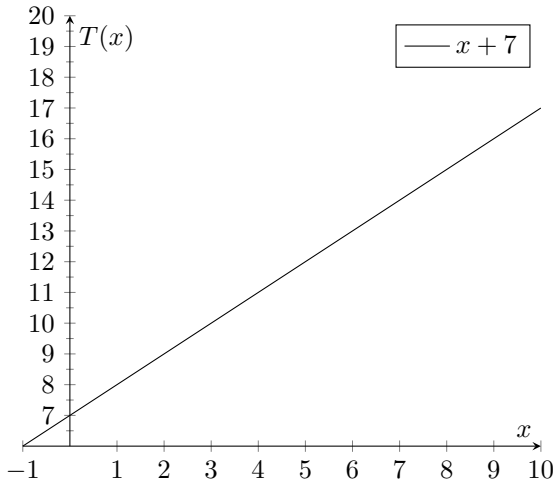

Figura 1: x+7

Entonces se plantea la ecuación de la definición de Big-O:

$$f(x) \leq c * g(x)$$

$x + 7 \leq x + 7x$, esto es válido si $\forall x > 1$

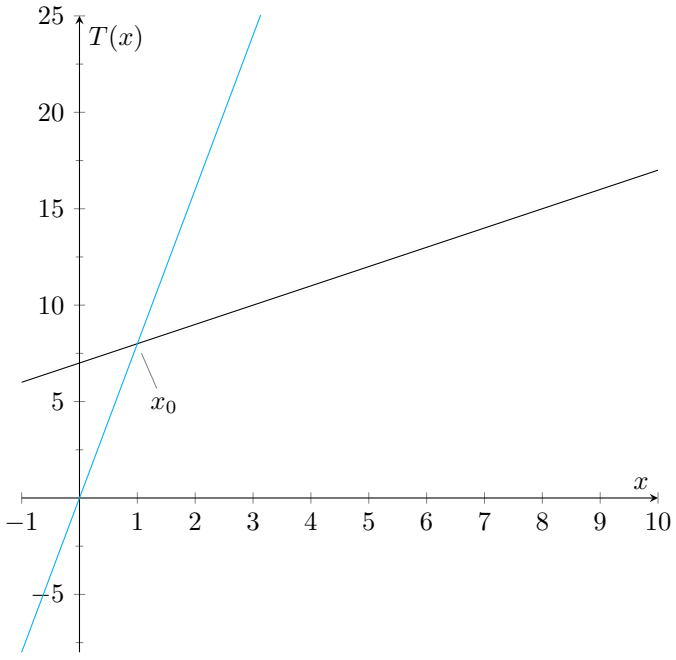

Figura 2: x+7

$x + 7 \leq 8x$, esto es válido si $\forall x > 1$

$$|x + 7| \ \leq \ |8x| \ \forall x > 1$$

$$|T(x)| = |x + 7| \leq |g(x)| = \ |8x|$$

Entonces $\mathbf{f(x)}$ **es** $\mathbf{O(x)}$, donde $\mathbf{x_0 = 1}$ y $\mathbf{C = 8}$.

Ejercicio

Demostrar que $4x_3 + 7x_2 + 12$ es $O(x_3)$ encontrar c y k.

Solución del ejercicio

Al igual que en el ejemplo anterior, en primer lugar se debe realizar el análisis de la función, para tener una idea de lo que se esta buscando. Y recordad que la definición de $\mathbf{f(n) = O(g(n))}$: Se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * g(n)}$, para todo $\mathbf{n \geq n_0}$

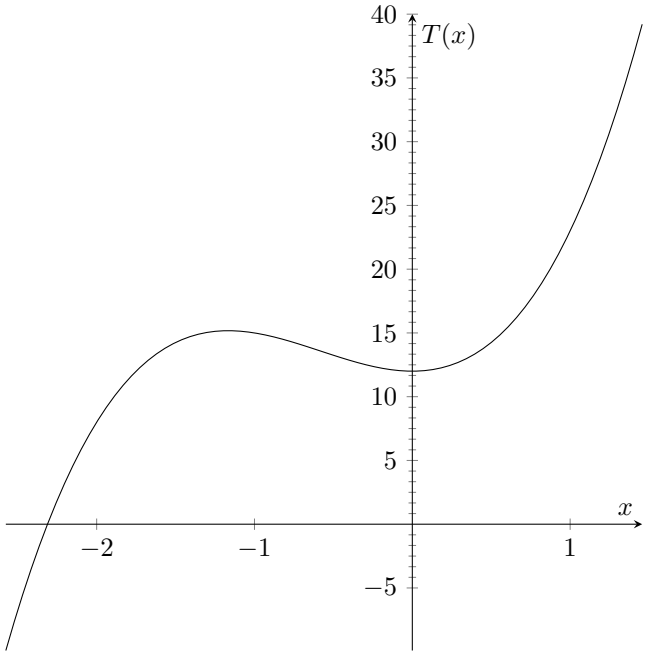

Figura 3: $4 * x^3 + 7 * x^2 + 12$

Entonces se plantea la ecuación de la definición de Big-O:

$$f(x) \leq c * g(x)$$

$4 * x^3 + 7 * x^2 + 12 \leq 4 * x^3 + 7 * x^3 + 12 * x^3$, esto es válido si $\forall x > 1$

$|4 * x^3 + 7 * x^3 + 12x^3| \leq |23 * x^3|$, esto es válido si $\forall x > 1$

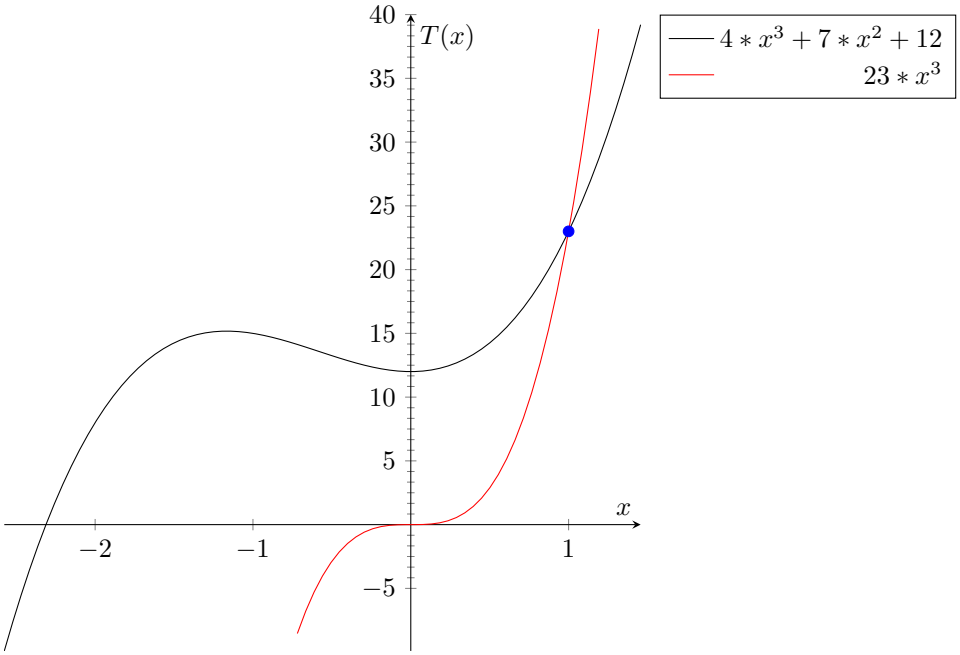

Figura 4: $4 * x^3 + 7 * x^2 + 12$

Por definición: se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * f(n)}$, para todo $\mathbf{n \geq n_0}$. Entonces $\mathbf{f(x)}$ **es** $\mathbf{O(x)}$, donde $\mathbf{x_0 = 1}$ y $\mathbf{C = 23}$.

Ejercicio

Demostrar que $x_3 + 20x + 1$ es $O(x_3)$ encontrar c y k.

Solución del ejercicio

Existen varias formas de hacer esta demostración, una es la misma que en el ejercicio anterior, en primer lugar se debe realizar el análisis de la función, para tener una idea de lo que se esta buscando. Y recordad que la definición de $\mathbf{f(n) = O(g(n))}$: Se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * g(n)}$, para todo $\mathbf{n \geq n_0}$

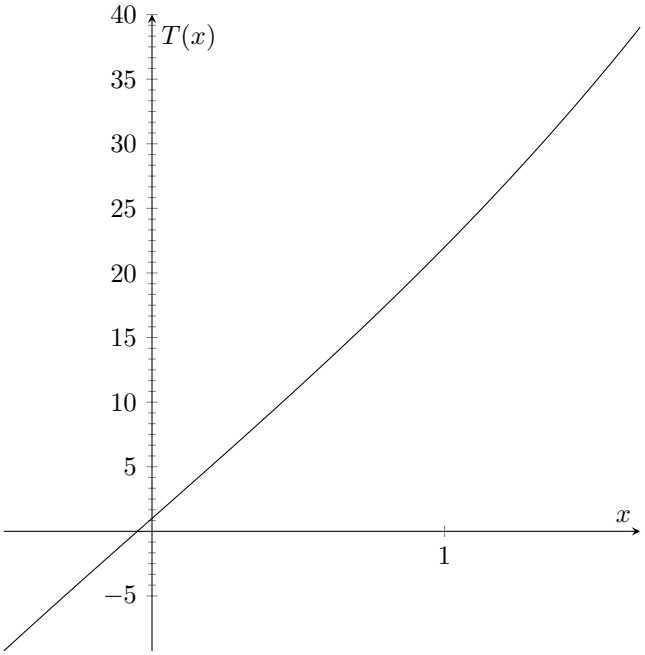

Figura 5: $x^3 + 20 * x + 1$

Entonces se plantea la ecuación de la definición de Big-O:

$$f(x) \leq c * g(x)$$

$x^3 + 20 * x + 1 \leq x^3 + 20 * x^3 + 1 * x^3$, esto es válido si $\forall x > 1$

$|x^3 + 20 * x^3 + x^3| \leq |22 * x^3|$, esto es válido $\forall x > 1$

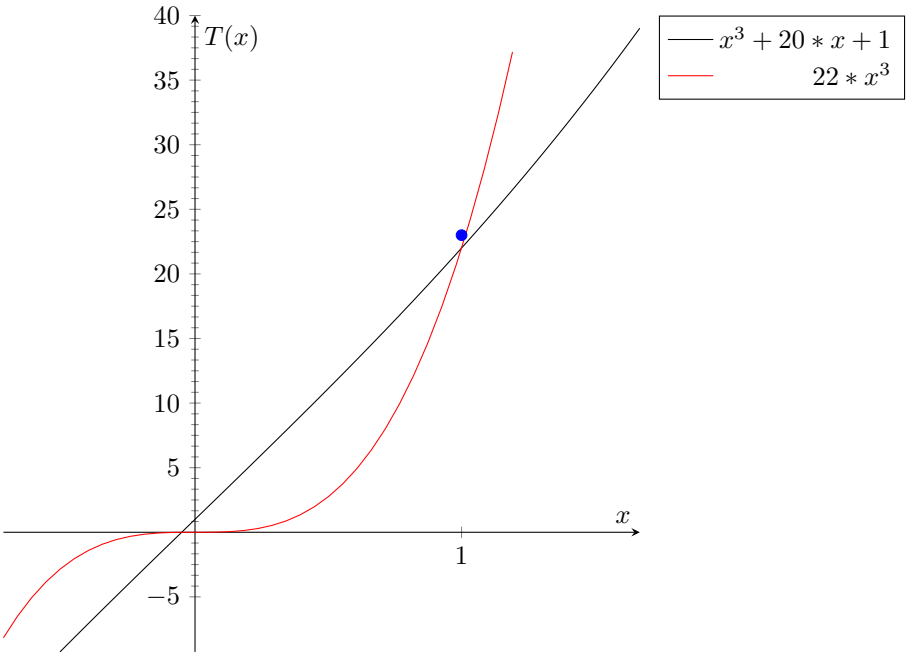

Figura 6: $x^3 + 20 * x + 1$

Por definición: se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * f(n)}$, para todo $\mathbf{n \geq n_0}$. Entonces $\mathbf{f(x)}$ **es** $\mathbf{O(x)}$, donde $\mathbf{x_0 = 1}$ y $\mathbf{C = 22}$.

Solución del ejercicio

Otra forma de demostrar lo mismo es planteando lo siguiente:

Siempre hay que recordar que la definición de $\mathbf{f(n) = O(g(n))}$: Se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * g(n)}$, para todo $\mathbf{n \geq n_0}$

Entonces se plantea la ecuación de la definición de Big-O:

$$f(x) \leq c * g(x)$$

$$x^3 + 20 * x + 1 \leq C * x^3$$

$$1 + \frac{20}{x^2} + \frac{1}{x^3} \leq c$$

Sea x =1 entonces, $1 + 20 + 1 \leq c$

$$22 \leq c$$

$$22x^3 = g(x) \qquad , \forall\ x \geq 1$$

$$O(g(x)) = O(22x^3) \qquad , \forall\ x \geq 1$$

$$O(g(x)) = O(x^3) \qquad , \forall\ x \geq 1$$

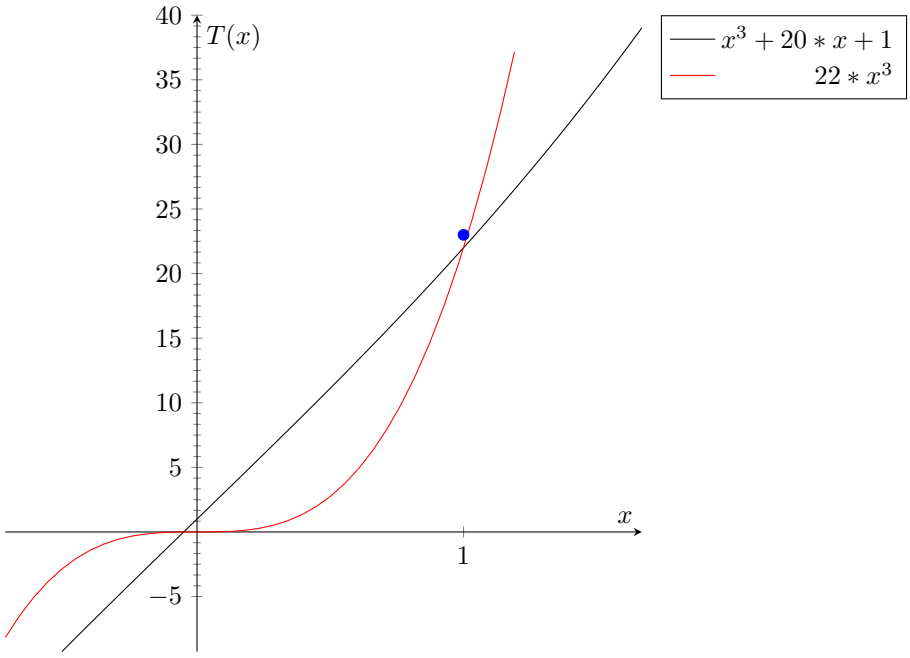

Figura 7: $x^3 + 20 * x + 1$

Por definición: se dice que $\mathbf{T(n) = O(f(n))}$ si y solo si existe un $\mathbf{c > 0}$ y un $\mathbf{n_o}$ tal que, $\mathbf{f(n) \leq c * f(n)}$, para todo $\mathbf{n \geq n_0}$. Entonces $\mathbf{f(x)\ es\ O(x)}$, donde $\mathbf{x_0 = 1}$ y $\mathbf{C = 22}$.

3. Análisis de Algoritmos Recursivos

El análisis de algoritmos recursivos se divide en dos grandes grupos:

- Los que se solucionan gracias a la aplicación y validez del Teorema Maestro,
- y los que no.

3.0.1. Aplicación del Teorema Maestro

Ejercicio Aplicación del Teorema Maestro 1

Calcular la complejidad del algoritmo recursivo cuya ecuación de recurrencia es:

$$T(n) = 4T(\frac{n}{2}) + 3n^2$$

Solución del ejercicio

Dado que se está trabajando con una ecuación de recurrencia del estilo:

$$T(n) = aT(n/b) + f(n)$$

Donde las constantes $\mathbf{a} >= \mathbf{1}$ y $\mathbf{b} > \mathbf{1}$ y por otro lado $\mathbf{f(n)}$ es una función asintóticamente positiva. Para calcular la complejidad del mismo se suele utilizar el teorema maestro:
Para ello hay que determinar la relación ente los dos términos de la ecuación de recurencia:

- Si $f(n)$ **es polinomicamente más pequeña** que $n^{\log_b a}$ el tiempo de ejecución que prevalece es el tiempo de división recursiva.
- Si finalmente $f(n)$ y $n^{\log_b a}$ **son asintóticamente iguales**.
- Si por el contrario $f(n)$ **es polinomicamente dominante sobre** $n^{\log_b a}$ entonces el tiempo de ejecución que prevalece es el de $f(n)$.

Para ello se debe simplemente hallar a,b y f(n). Una vez obtenidos esos valores se calcula $n^{\log_b a}$ y se compara con $f(n)$. En este caso $T(n) = 4T(\frac{n}{2}) + 3n^2$

$$f(n) = 3n^2$$

$$a = 4$$

$$b = 2$$

dados estos valores de a y b se calcula $n^{\log_b a} = n^{\log_2 4} = n^2$

Por lo tanto se aplica el caso 2 del teorema maestro:

Por ende $n^{\log_b a} = n^2 = f(n)$

Entonces $T(n) = O(n^{\log_b a} * \log\ n) = \mathbf{O(n^2 * \log\ n)}$

Ejercicio Aplicación del Teorema Maestro 2

Calcular la complejidad del algoritmo recursivo cuya ecuación de recurrencia es:

$$T(n) = 9T(\frac{n}{3}) + n$$

Solución del ejercicio

Dado que se está trabajando con una ecuación de recurrencia del estilo:

$$T(n) = aT(n/b) + f(n)$$

Donde las constantes $\mathbf{a} >= \mathbf{1}$ y $\mathbf{b} > \mathbf{1}$ y por otro lado $\mathbf{f(n)}$ es una función asintóticamente positiva. Para calcular la complejidad del mismo se suele utilizar el teorema maestro:
Para ello hay que determinar la relación ente los dos términos de la ecuación de recurencia:

- Si $f(n)$ **es polinomicamente más pequeña** que $n^{\log_b a}$ el tiempo de ejecución que prevalece es el tiempo de división recursiva.
- Si finalmente $f(n)$ y $n^{\log_b a}$ **son asintóticamente iguales**.

- Si por el contrario $f(n)$ **es polinomicamente dominante sobre** $n^{\log_b a}$ entonces el tiempo de ejecución que prevalece es el de $f(n)$.

Para ello se debe simplemente hallar a,b y f(n). Una vez obtenidos esos valores se calcula $n^{\log_b a}$ y se compara con $f(n)$. En este caso $T(n) = 9T(\frac{n}{3}) + n$

$$f(n) = n$$

$$a = 9$$

$$b = 3$$

Dados estos valores de a y b se calcula $n^{\log_b a} = n^{\log_3 9} = n^2$

Por lo tanto:

Por ende $n^{\log_b a} = n^2 > f(n)$

Entonces según el Teorema Maestro, aplica el caso 1 $T(n) = O(n^{\log_b a}) = \mathbf{O(n^2)}$

Ejercicio Aplicación del Teorema Maestro 3

Calcular la complejidad del algoritmo recursivo cuya ecuación de recurrencia es:

$$T(n) = T(\frac{2n}{3}) + 1$$

Solución del ejercicio

Para ello se debe simplemente hallar a,b y f(n). Una vez obtenidos esos valores se calcula $n^{\log_b a}$ y se compara con $f(n)$. En este caso $T(n) = T(\frac{2n}{3}) + 1$

$$f(n) = 1$$

$$a = 1$$

$$b = \frac{3}{2}$$

Dados estos valores de a y b se calcula $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Por lo tanto:

Por ende $n^{\log_b a} = 1 = f(n)$

Entonces según el Teorema Maestro, aplica el caso 2 $T(n) = O(n^{\log_b a} * \log\ n) = \mathbf{O(n^0 * \log\ n)} = \mathbf{O(\log\ n)}$

Ejercicio Aplicación del Teorema Maestro 4: Búsqueda Binaria

Calcular la complejidad del algoritmo recursivo de Búsqueda Binaria

Solución del ejercicio

En primer lugar se debe conocer el algoritmo de la búsqueda binaria recursiva:

```c
int busqueda_binaria(vector_t vector, int inicio,   int fin, elemento_t elemento_buscado) {
    int centro;

    if(inicio<=fin){
        centro=(fin+inicio)/2;
        if(vector[centro]==elemento_buscado)
            return centro;
        else if(elementoBuscado < vector[centro])
            return busqueda_binaria(vector,inicio,centro-1,elemento_buscado);
        else
            return busqueda_binaria(vector,centro+1,fin,elemento_buscado);
    } else
    return -1;
}
```

Code 1: Búsqueda Binaria Recursiva

Ahora se debe armar la ecuación de recurrencia, para ello se sabe que la función sólo realiza UNA llamada recursiva por scope y divide a la mitad el set de datos. Finalmente el costo de la parte no recursiva, $f(n)$ es 1, el que equivale al return:

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

Como cumple con el tipo de ecuación de recurrencia al que puede aplicarse el teorema maestro. Simplemente hallar a,b y f(n). Una vez obtenidos esos valores se calcula $n^{\log_b a}$ y se compara con $f(n)$. En este caso $T(n) = T(\frac{n}{2}) + 1$

$$f(n) = 1$$

$$a = 1$$

$$b = 2$$

Dados estos valores de a y b se calcula $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Por lo tanto:

Por ende $n^{\log_b a} = 1 = f(n)$

Entonces según el Teorema Maestro, aplica el caso 2 $T(n) = O(n^{\log_b a} * \log\ n) = \mathbf{O(n^0 * \log\ n)} = \mathbf{O(\log\ n)}$