



Instituto Politécnico Nacional  
Escuela Superior de Cómputo



Análisis de Algoritmos, Sem: 2022-2, 3CV11, Práctica 3, 03/04/2022

## PRÁCTICA 3: FUNCIONES RECURSIVAS VS ITERATIVAS.

Steiner Vázquez Anthony Francis

*asteinerv1700@alumno.ipn.mx*

**Resumen:** Este documento presenta una comparación de distintos algoritmos que resuelven el problema de dividir 2 números enteros; posteriormente se resolverá el problema de buscar un número en un arreglo de enteros, de manera similar al algoritmo de Búsqueda Binaria, solo que en vez de partir el arreglo en 2, se hará en 3 partes; dicho problema se resolverá de manera recursiva tanto como iterativa, para hacer una comparación de ambos algoritmos.

**Palabras Clave:** C++, Iteratividad, Recursividad, Recurrencias

### 1 Introducción

En la práctica pasada hablamos de lo que era la recursión y la iteración; vimos que los algoritmos pueden expresarse de ambas formas, aunque a veces un problema se puede representar de manera más natural en alguna de las 2 formas, y que a veces varía la complejidad, aunque la complejidad, hasta ahora, solo la hemos podido ver de manera experimental, es decir, a *posteriori*, así que en esta práctica estaremos viendo cómo hacer el análisis a *priori* de algoritmos recursivos, y en general de algoritmos que generen recurrencias.

Se van a tratar los problemas de la división de 2 números enteros, y también el problema de buscar un número en un arreglo de enteros, vamos a ver diferentes algoritmos para resolver este problema, algunos iterativos, otros recursivos, y haremos una comparación del desempeño que tienen los mismos.

### 2 Conceptos Básicos

Vamos a explorar más a fondo la recursividad. Empezando por checar los elementos que componen a una función recursiva.

Primero tenemos el tamaño del problema. Aunque el tamaño de problema no es propio de los algoritmos recursivos, y ya tratamos esto en una práctica anterior, hay que recordar que el tamaño de problema o tamaño de las entradas de un algoritmo, que como sabemos tiene entrada y salida, y que para resolver un problema debemos entender bien la relación entre las entradas y salidas. Se puede dar como una expresión matemática que involucra cantidades relacionadas a los parametros de entrada y estos determinan la complejidad en término del número de operaciones que el algoritmo debe hacer para resolverlo. Cuando se ocupa la recursión, es especialmente importante puesto que los casos recursivos, y base, claramente, dependen de este. En muchos problemas el tamaño del problema viene en alguno de los parámetros de entrada, pero en otros casos puede estar implícita debido al lenguaje de programación y la forma en que nos permita acceder a través de sintaxis un tamaño en particular.

Segundo tenemos los casos base. Estos son instancias de problemas que pueden ser resueltos sin uso alguno de recursión, más aún, sin acarrear llamadas recursivas. El tipo más común generalmente está asociado con las instancias más pequeñas del problema, para el cual resulta resolverlo de manera trivial, a veces no se necesita hacer ningún cálculo. Algunas funciones pueden tener más de un caso base, aunque se debe evitar tener casos base incorrectos, o innecesarios, puesto que son redundantes, esto se da cuando estamos poniendo un caso base, que puede ser calculado mediante el caso recursivo; también esto puede restar generalidad, y esto ocasionaría que nuestra función no estuviera definida en todos los puntos, y podría llevar a errores como exceder el máximo número de llamadas a función permitidas. Y peor aún, no tener un caso base, implicaría tener una recursión infinita, y por tanto nunca pararía, lo que resultaría en el mismo error. Habíamos visto que para poder definir una función recursiva es imperativo escribir un caso base, el cual ayuda a terminar de calcular y parar la recursión.

Tercero tenemos la descomposición del problema. Esto es buscar subproblemas y problemas adicionales; los subproblemas generalmente son similares al problema original pero menores en tamaño. Los subproblemas son ocupados para establecer los casos recursivos. Se puede ver como un proceso que involucra el decrementar el tamaño del problema, y es por eso que es importante determinar correctamente el tamaño del problema. Se suele expresar como una fórmula puesto que es una función matemática.

Finalmente tenemos los casos recursivos. Definir los casos recursivos, involucra el obtener una forma de construir la solución total del problema original mediante el uso de soluciones de subproblemas similares, que fueron previamente identificados.

Para construir un algoritmo recursivo es importante tener en cuenta sus partes, y observar como modificar, extender, o combinar estas soluciones de los subproblemas para poder resolver el problema original.

Los algoritmos recursivos pueden ser divididos en categorías, dependiendo de algunos criterios, es decir, existen tipos de algoritmos recursivos, y pueden estar en diferentes categorías.

Primero vamos a ver las recursiones Lineales, estas ocurren cuando las funciones que de alguna manera procesan el resultado de la llamada recursiva, antes de retornar el valor, y solo son llamadas una vez.

Ahora veremos recursión de Cola, este tipo de recursión, es lineal también, así que igualmente solo hace una llamada a sí misma una vez, pero la llamada recursiva es la última operación que se se acarrea del caso recursivo, debido a esto no manipula directamente el resultado de la llamada recursiva. Generalmente estas recursiones modifican el conjunto de argumentos hasta que sea posible calcular la solución de manera simple.

Luego tenemos las recursiones Múltiples, esta ocurre cuando una función se llama a sí misma múltiples veces, para algún caso recursivo.

También se tiene las recursiones mutuas, se dice que un conjunto de funciones son mutuamente recursivas cuando pueden llamarse entre ellas en un orden cíclico. En otras palabras se puede tener una función  $f$  que puede llamar a una función  $g$ , y a su vez puede llamar a una función  $h$ , y esta termina mandando a llamar a la función  $f$ . A este tipo de recursión, también se le conoce como recursión indirecta.

Finalmente tenemos la recursión Anidada, esta ocurre cuando un argumento de una función recursiva es definida mediante otra llamada recursiva.

Los algoritmos recursivos suelen expresarse mediante relaciones de recurrencia, que son funciones matemáticas recursivas, digamos  $T$ , que describe su costo computacional. Si un algoritmo depende del tamaño de un parámetro de entrada, digamos  $n$ , entonces  $T$  será una función de  $n$ . Entonces checamos cuántas operaciones hace en los casos base, y se hace una relación; ahora para el caso recursivo simplemente veremos como modifica el tamaño de problema  $n$ , y se expresa, de nuevo  $T$  como una función de  $n$  pero con las modificaciones que se le harán durante cada llamada recursiva. Todo junto representa una función por partes, es decir una función definida por múltiples subfunciones, que representan a la función en diferentes intervalos del Dominio de la función. Aunque las recurrencias no son propias de los algoritmos recursivos, puesto que un algoritmo iterativo que modifique el tamaño del problema, también puede generar una recurrencia.

Vamos a ver un ejemplos de relaciones de recurrencia para diferentes algoritmos recursivos de distintas categorías para Fibonacci.

Fibonacci mediante recursión Lineal:

$$f(n) = \begin{cases} 1 & \text{si } n = 2 \text{ o } n = 2, \\ \lfloor \Phi * f(n-1) + 1/2 \rfloor & \text{si } n > 2, \end{cases} \quad (1)$$

donde  $\Phi = \frac{1 + \sqrt{5}}{2}$ , de tal modo que  $F(n) = f(n)$ .

Fibonacci mediante recursión de Cola:

$$f(n, a, b) = \begin{cases} b & \text{si } n = 1, \\ f(n-1, a+b, a) & \text{si } n > 1. \end{cases} \quad (2)$$

así  $F(n) = f(n, 1, 1)$ .

Fibonacci mediante recursión Múltiple:

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \text{ o } n = 2, \\ [f(\frac{n}{2} + 1)]^2 - [f(\frac{n}{2} - 1)]^2 & \text{si } n > 2 \text{ y } n \text{ es par}, \\ [f(\frac{n+1}{2})]^2 + [f(\frac{n-1}{2})]^2 & \text{si } n > 2 \text{ y } n \text{ es impar}. \end{cases} \quad (3)$$

de este modo  $F(n) = f(n)$ .

Fibonacci mediante recursión Mutua:

$$A(n) = \begin{cases} 0 & \text{si } n = 1 \\ A(n-1) + B(n-1) & \text{si } n > 1, \end{cases} \quad B(n) = \begin{cases} 1 & \text{si } n = 1 \\ A(n-1) & \text{si } n > 1. \end{cases} \quad (4)$$

con esto tenemos  $F(n) = B(n) + A(n)$ .

Fibonacci mediante recursión Anidada:

$$f(n, s) = \begin{cases} 1 + s & \text{si } n = 1 \text{ o } n = 2, \\ f(n-1, s + f(n-2, 0)) & \text{si } n > 2. \end{cases} \quad (5)$$

de modo que  $F(n) = f(n, 0)$ .

Aunque el hecho de que  $T(n)$  describe el costo computacional, aún debemos obtener el la función de su orden de crecimiento. Para hacer esto, se debe resolver la relación de recurrencia, y existen varios métodos para hacer esto.

A la expresión matemática a resolver la vamos a llamar, simplemente, recurrencia, y a la llamada recursiva le vamos a sumar el costo de lo que hace que nuestra llamada recursiva sea ejecutada, y al caso base le llamaremos condición de frontera. Resolver una recurrencia significa encontrar el valor para el  $n$ -ésimo término generado por la recurrencia.

Ahora veremos el método de sustitución hacia adelante.

Para resolver una recurrencia mediante este método se debe empezar por la condición de frontera, e ir calculando los demás términos, de manera sucesiva. Generando así una sucesión, el  $n$ -ésimo término de esta sucesión es una función, esta función es la función de complejidad temporal, únicamente, de la llamada recursiva, puesto que la recurrencia puede ser parte de un algoritmo.

Luego tenemos el método de sustitución hacia atrás.

Este método es el contrario a sustitución hacia adelante, aquí se comienza por la recursión original  $T(n)$ , y se va sustituyendo con el caso recursivo, y así sucesivamente se va sustituyendo hasta que el argumento de la llamada recursiva se convierte en nuestra condición de frontera, de tal manera que podemos observar la función de complejidad temporal hasta ese momento.

Ahora tenemos el método de Recurrencias Lineales de Orden  $N$  con coeficientes constantes homogéneas.

Este método es el análogo a resolver una ecuación diferencial lineal de orden  $n$ , con coeficientes constantes homogénea. Aquí tienen la forma:

$a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k}$  donde cada  $C_i$  es una constante y  $C_k$  es diferente de 0; el valor  $k$  es llamado el grado de la relación de recurrencia.

Esta expresión tiene un polinomio característico el cual es:

$r^k = C_1 r^{k-1} + C_2 r^{k-2} + \cdots + C_k$ , es decir reemplazar  $a_i$  con  $r^{i-(n-k)}$ , ahora debemos calcular las raíces características de la relación de recurrencia, que son las raíces del polinomio característico.

Para resolver este tipo de recurrencias tenemos los siguientes Teoremas.

**Teorema 1.**

Dada una relación de recurrencia  $a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k}$  con  $k$  distintas raíces características  $r_1, \dots, r_k$ .

Entonces la forma cerrada de la solución para  $a_n$  tiene la forma:

$\alpha_1 r_1^n + \alpha_2 r_2^n + \cdots + \alpha_k r_k^n$ , además dadas  $k$  condiciones iniciales, las constantes  $\alpha_1, \dots, \alpha_k$  están determinadas de manera única.

Si no tenemos  $k$  distintas raíces tenemos lo siguiente:

**Teorema Generalizado.**

Tenemos una relación de recurrencia  $a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k}$  con  $t$  raíces características distintas  $r_1, \dots, r_k$ , con multiplicidades  $m_1, \dots, m_k$ .

Entonces la solución tiene la forma:

$$a_n = \sum_{i=0}^t (\alpha_{i,0} + \alpha_{i,1} \cdot n + \cdots + \alpha_{i,m_i-1} \cdot n^{m_i-1}) r_i^n$$

Finalmente con la condiciones iniciales, armamos un sistema de ecuaciones, y al resolverlo, tendremos los valores para cada  $C_i$ . Con lo cual tendríamos nuestra función de complejidad.

Ahora tenemos el método de Recurrencias Lineales de Orden  $N$  con coeficientes constantes no homogéneas.

Si tenemos una relación de recurrencia lineal de coeficientes constantes como:  $a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k} + F(n)$ , se dice que es no homogénea. La relación de recurrencia que se obtiene al eliminar  $F(n)$  se conoce como la relación de recurrencia homogénea asociada. Para resolver recurrencias de este tipo, también recurriremos al análogo en ecuaciones diferenciales del mismo tipo, donde combinaremos la solución de la homogénea, con una solución particular. Para ello usaremos los siguientes teoremas:

**Teorema 1.**

Se supone que  $a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k} + F(n)$  tiene una solución particular  $a_n^p$ , y  $a_n^h$  es la solución de la recurrencia homogénea asociada. Entonces toda solución tiene la forma:  $a_n^p + a_n^h$ . Si podemos encontrar una solución particular, entonces, mecánicamente podemos encontrar una solución que satisfaga las condiciones iniciales. Se encuentra de la siguiente manera.

**Teorema 2.**

Considere  $a_n = C_1 a_{n-1} + C_2 a_{n-2} + \cdots + C_k a_{n-k} + F(n)$  donde:

$$F(n) = (b_t n^t + b_{t-1} n^{t-1} + \cdots + b_1 n + b_0) s^n$$

Caso 1. Si  $s$  no es una raíz de la ecuación característica asociada, entonces existe una solución particular de la forma:

$$(p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0) s^n$$

Caso 2. Si  $s$  es una raíz con multiplicidad  $m$  de la ecuación característica, entonces existe una solución de la forma:

$$n^m (p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0) s^n$$

Ahora solo igualamos la forma de la solución particular con nuestra recurrencia, y resolvemos para los  $p_i$  acomodando todo para que este igualado a 0. Y esa sería la función de complejidad temporal.

Ahora se presenta el método de árboles de recurrencia.

Este método ocupa el construir un árbol de sustituir los valores de una recurrencia con la forma  $T(n) = aT(\frac{n}{b}) + f(n)$ , de tal manera que cada nodo del árbol tendrá  $a$  hijos, y los valores de esos nodos serán las  $f(n)$  evaluadas en la  $T$  de ese nivel. Luego se busca obtener cuál es el nivel del árbol donde solo hay  $T(1)$ , luego vemos cuántas operaciones realiza por cada nivel, se suman, y

así se obtiene la función de complejidad.

Ahora veremos el método heurístico.

Para ocupar este método se propone una función  $f(n)$ , la cuál chequearemos por medio de inducción matemática si es la complejidad de nuestro algoritmo, en caso contrario, se prueba con otra función y así sucesivamente, de tal modo que se checa a prueba y error.

Finalmente tenemos el método Maestro.

Para este método haremos uso del Teorema Maestro, que es el siguiente:

**Teorema Maestro.**

Sean  $a \leq 1$  y  $b > 1$  constantes. Sea  $f(n)$  una función, y sea  $T(n)$  una función definida sobre los números enteros no negativos, por la relación de recurrencia:

$$T(n) = \begin{cases} d & d > 0 \\ aT(\frac{n}{b}) + f(n) & \text{o.c.} \end{cases} \quad (6)$$

donde  $\frac{n}{b}$  puede ser  $\lfloor \frac{n}{b} \rfloor$  o  $\lceil \frac{n}{b} \rceil$ . Entonces  $T(n)$  está acotado asintóticamente por:

Caso 1. Si  $f(n) = O(n^{\log_b a - \epsilon})$  para algún  $\epsilon > 0$ , entonces  $T(n) = \Theta(n^{\log_b a})$ .

Caso 2. Si  $f(n) = \Theta(n^{\log_b a})$ , entonces  $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ .

Caso 3. Si  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para algún  $\epsilon > 0$ , y si  $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$  para alguna constante  $c < 1$ , y todas la  $n$  suficientemente grandes, entonces  $T(n) = \Theta(f(n))$ .

Ahora vamos a ver los tipos de recurrencias comunes.

Primero vamos a ver Decremento por 1.

La cual es una recurrencia de la forma  $T(n) = T(n - 1) + f(n)$ , este tipo de recurrencia se puede resolver por el método de sustitución hacia atrás, lo que nos da como resultado, que para saber la complejidad de este tipo de recurrencia basta con saber el valor de:  $\sum_{j=1}^n f(j)$ .

Luego tenemos Decremento por un factor constante.

Estas son recurrencias de la forma  $T(n) = T(\frac{n}{b}) + f(n)$ , con  $b > 1$ , igualmente mediante sustitución hacia atrás, haciendo el cambio de variable  $n = b^k$ , i.e.  $k = \log_b n$ , tenemos que la función de complejidad es:  $\sum_{j=1}^k f(b^j)$  o lo que es lo mismo:  $\sum_{j=1}^{\log_b n} f(b^j)$ .

Finalmente tenemos las recurrencias de Divide y Vencerás.

Una recurrencia Divide y Vencerás es de la forma:  $T(n) = aT(\frac{n}{b}) + f(n)$  con  $a \geq 1$  y  $b \geq 2$ . Igualmente mediante sustitución hacia atrás, con el cambio de variable  $n = b^k$ , tenemos que la función de complejidad es:  $n^{\log_b a} \sum_{j=1}^{\log_b n} \frac{f(b^j)}{a^j}$ .

Con esto vamos a ver los siguientes algoritmos y vamos a calcular su orden de complejidad de cada uno.

Empezaremos con el problema de la división de 2 números enteros, del cual veremos 3 algoritmos.

Tenemos el algoritmo 1, que llamaremos División 1, que es iterativo.

---

**Algorithm 1** Division1( $n, div, res$ ):

---

```
 $q \leftarrow 0$   
while  $n \geq div$  do  
     $n \leftarrow n - div$   
     $q \leftarrow q + 1$   
end while  
 $res \leftarrow n$   
return  $q$ 
```

---

Veamos su funcionamiento, si  $n=3$ , y  $div = 1$ , entonces el algoritmo comienza poniendo la variable auxiliar  $q$  en 0, después itera mientras que  $n$  sea menor o igual a  $div$ , haciendo que  $n$  ahora valga  $n - div$  es decir para la primera iteración  $n$  valdrá 2, y le aumenta 1 a  $q$  es decir ahora valdrá 1, luego para la 2da iteración  $n$  sigue siendo mayor o igual a  $div$  por lo tanto entra al ciclo y ahora  $n$  valdrá 1, y  $q$  2, finalmente entra por última vez al ciclo, puesto que  $n$  ahora es igual a  $div$ , por lo tanto  $n$  ahora vale 0, y  $q$  vale 3, por lo tanto nuestra variable  $res$  que tendrá el residuo de la operación será igual a  $n$ , y retornamos  $q$ , lo que nos diría que el resultado de dividir 3 entre 1, es 3, con residuo 0, por lo tanto el algoritmo es correcto.



Ahora veremos el algoritmo 2, que llamaremos División 2, también iterativo.

---

**Algorithm 2** Division2( $n, div, r$ ):

---

```

 $dd \leftarrow div$ 
 $q \leftarrow 0$ 
 $r \leftarrow n$ 
while  $dd \leq n$  do
     $dd \leftarrow 2 \cdot dd$ 
end while
while  $dd > div$  do
     $dd \leftarrow \frac{dd}{2}$ 
     $q \leftarrow 2 \cdot q$ 
    if  $dd \leq r$  then
         $r \leftarrow r - dd$ 
         $q \leftarrow q + 1$ 
    end if
end while
return  $q$ 

```

---

Este algoritmo comienza declarando una variable auxiliar  $dd$  e inicializandola con el valor de  $div$ , la variable auxiliar  $q$  con 0, y  $r$  con  $n$ , la cual será nuestro residuo. Después iteramos mientras que  $dd$  sea menor o igual a  $n$  duplicando su valor cada vez. Posteriormente, iteramos mientras que  $dd$  sea mayor que  $div$ , reduciendo a la mitad el valor de  $dd$  en cada iteración, luego duplicando el valor de  $q$ , y luego checamos si  $dd$  es menor o igual a  $r$  en tal caso restamos  $dd$  de  $r$ , y le aumentamos 1 a  $q$ , finalmente retornamos  $q$ . Veamos en funcionamiento este algoritmo para cuando  $n = 3$  y  $div = 1$ , tenemos que  $dd$  será 1,  $q$  será 0, y  $r$  será 3. Así en el primer ciclo, para la primer iteración  $dd$  valdrá 2, vuelve a entrar y aumenta a 4, y se sale del ciclo. Luego para el siguiente ciclo, como  $dd$  vale 4 es mayor que  $div$  que vale 1, entonces entra, y reduce el valor de  $dd$  a 2, luego duplica el valor de  $q$  pero sigue en 0, luego checamos si  $dd$  es menor o igual a  $r$  que vale 3, lo cuál es cierto, entonces ahora  $r$  vale 1, y aumentamos 1 a  $q$  por lo tanto ahora vale 1, en la segunda iteración  $dd$  que vale 2, sigue siendo mayor a  $div$  que vale 1, entonces entra y ahora  $dd$  vale 1, y  $q$  ahora vale 2, checamos si  $dd$  que vale 1 es menor o igual que  $r$  que vale 1, lo cual es cierto entonces entra, y ahora  $r$  vale 0, y  $q$  vale 3. Ya no se cumple y deja de iterar, por lo que solo restar retornar  $q$  es decir 3. Lo cual nos dice que el resultado de dividir 3 entre 1, es 3 con residuo de 0, lo cual indica que el algoritmo es correcto.

Finalmente veremos el algoritmo 3, que llamaremos División 3, recursivo.

---

**Algorithm 3** Division3( $n, div$ ):

---

```
if  $div > n$  then
    return 0
else
    return  $1 + \text{Division3}(n - div, div)$ 
end if
```

---

Este algoritmo recursivo simplemente chequea si  $div$  es mayor a  $n$  en tal caso simplemente retorna 0, pero en caso contrario retorna  $1 +$  la llamada recursiva con tamaño  $n-div$ . Veamos cómo funciona si  $n$  es igual a 3, y  $div$  vale 1. Primero chequeamos si  $div$  es mayor a  $n$  lo cual no se cumple, por lo tanto entra al else, y retorna  $1 + \text{Division3}(2,1)$ , la cual, también, hay que calcular, así que ahora chequearemos  $\text{Division3}(2,1)$ , aquí  $n$  es igual a 2, y  $div$  sigue valiendo 1, tampoco se cumple que  $div$  sea mayor que  $n$ , entonces retornamos  $1 + \text{Division3}(1,1)$  ahora debemos calcular  $\text{Division3}(1,1)$ , como tampoco se cumple que  $div$  sea mayor que  $n$ , retorna  $1 + \text{Division3}(0,1)$ , para este caso  $div$  es mayor a  $n$ , por lo tanto retorna 0, ahora se puede calcular todo, que sería sumar  $1 + 1 + 1 + 0$ , lo cual es 3, que es el resultado de dividir 3 entre 1, por lo tanto el algoritmo es correcto.

Ahora veremos el 2do problema, es decir el de buscar un número en un arreglo de enteros.

En su forma Recursiva:

---

**Algorithm 4** *BusquedaEn3Bloques*( $A, busca, inicio, final$ ):

---

```

if  $inicio > final$  then
    return  $-1$ 
else
     $i \leftarrow (final - inicio)/3 + inicio$ 
     $j \leftarrow 2 * (final - inicio)/3 + inicio$ 
    if  $busca == A[i]$  then
        return  $i$ 
    else if  $busca == A[j]$  then
        return  $j$ 
    else if  $busca > A[j]$  then
         $inicio \leftarrow j + 1$ 
        return BusquedaEn3BloquesRecursiva( $A, busca, inicio, final$ )
    else if  $busca < A[i]$  then
         $inicio \leftarrow i + 1$ 
         $final \leftarrow j - 1$ 
        return BusquedaEn3BloquesRecursiva( $A, busca, inicio, final$ )
    else
         $final \leftarrow i - 1$ 
        return BusquedaEn3BloquesRecursiva( $A, busca, inicio, final$ )
    end if
end if

```

---

Este algoritmo parte en 3 bloques nuestro arreglo, y checa si está en la 1ra parte, o en la 2da parte, si no está ahí, entonces checa el rango en el que se encuentra nuestro valor, y checa ahora solo en ese pedazo volviendo a partirlo en 3, y así, hasta que o encuentra el valor buscado, o no lo encuentra cuando los índices se cruzan.

Por ejemplo si tenemos un arreglo con los elementos: 1,2,3,4,5,6,7,8,9,10; nuestras variables  $i, j$  toman los valores de 4, y 8 respectivamente, supongamos que buscamos el 6, entonces ahora solo checara entre el subarreglo 5,6,7, en este caso  $i$  valdría 5, y  $j$  valdría 7, finalmente solo checa en el subarreglo 6; como este es el valor que estabamos buscando entonces regresa su índice.

En su forma Iterativa:

---

**Algorithm 5** BusquedaEn3Bloques( $A, n, busca$ ):

---

```

inicio  $\leftarrow 0$ 
final  $\leftarrow n - 1$ 
while inicio  $\leq$  final do
     $i \leftarrow (final - inicio)/3 + inicio$ 
     $j \leftarrow 2 * (final - inicio)/3 + inicio$ 
    if busca ==  $A[i]$  then
        return  $i$ 
    else if busca ==  $A[j]$  then
        return  $j$ 
    else if busca >  $A[j]$  then
         $inicio \leftarrow j + 1$ 
    else if busca >  $A[i]$  then
         $inicio \leftarrow i + 1$ 
         $final \leftarrow j - 1$ 
    else
         $final \leftarrow i - 1$ 
    end if
end while
return  $-1$ 

```

---

El algoritmo funciona exactamente igual que en su forma recursiva, simplemente está escrito de manera iterativa.

Finalmente vamos a calcular sus ordenes de complejidad.

### 3 Experimentación y Resultados

Vamos a empezar con nuestro 1er problema, la división de 2 números enteros, y los 3 algoritmos que ya presentamos, para todos los algoritmos, el tamaño del problema es  $n$ , el mejor caso es cuando  $div$  es mayor a  $n$ , y el peor caso cuando  $div$  vale 1.

Para División1:

Veamos en ejecución este algoritmo.

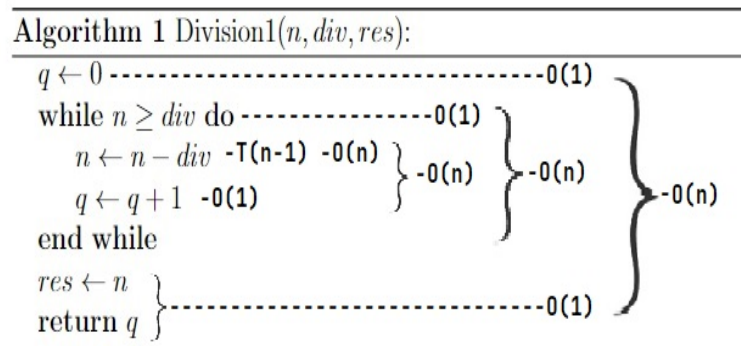
```

) ./Div1
5 1
La división entre 5 y 1 es: 5 con residuo: 0
El algoritmo Division1 hizo: 19 operaciones.

```

Figura 1. Ejecución de Division1

Ahora veamos su complejidad temporal *a priori*.

Figura 2. Análisis *a priori* de Division1

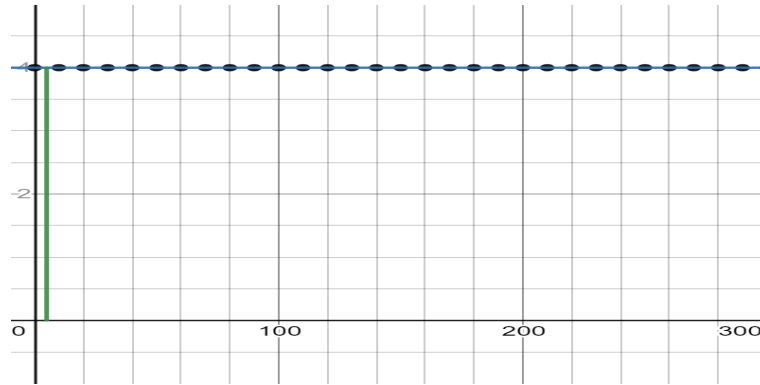
Ya que el peor caso es cuando  $div$  vale 1, entonces la recurrencia es del tipo Decremento por 1, por lo tanto es lineal.

Ahora realizaremos el análisis *a posteriori*  
 Para el mejor caso:

n	# operaciones
0	4
10	4
20	4
30	4
40	4
50	4
60	4
70	4
80	4
90	4
100	4

Table 1: Mediciones para el mejor caso de Division1

Así proponemos la función  $f(n) = 4$ .

Figura 3. Gráfica de la función  $f(n) = 4$ .

Vemos que la función pasa por todos los puntos. De la definición de  $\Omega$ :  
 Necesitamos encontrar un  $c_1 g(n) \leq 4 \forall n \geq n_1$ , de ahí que:  
 $3(1) \leq 4 \forall n \geq 1$ , entonces  $c_1 = 3$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 4 \in \Omega(1)$ .

Para el peor caso:

n	# operaciones
0	4
10	19
20	34
30	49
40	64
50	79
60	94
70	109
80	124
90	139
100	154

Table 2: Mediciones para el peor caso de Division1

Así proponemos la función  $f(n) = 2n$ .

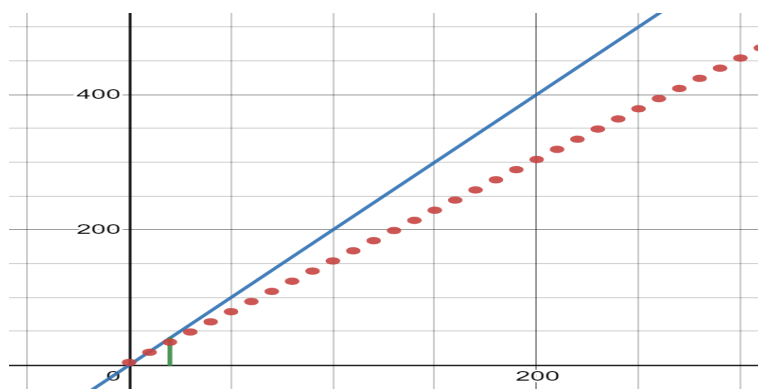


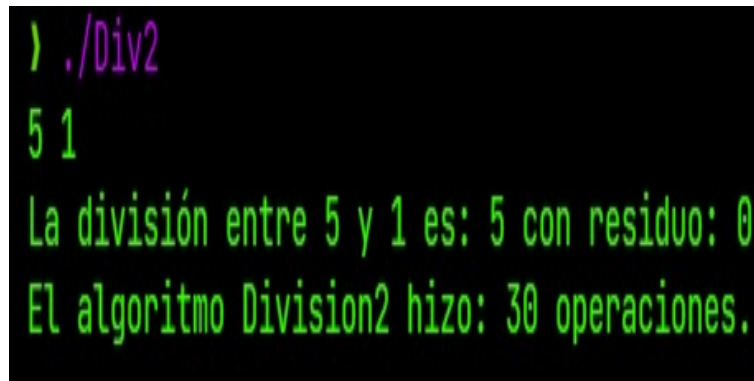
Figura 4. Gráfica de la función  $f(n) = 2n$ .

Podemos observar que algunos de estos puntos pasan por la función, y los demás están por debajo. De la definición de  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 2n \forall n \geq n_1$ , de ahí que:

$2n \leq 3n \forall n \geq 20$ , entonces  $c_1 = 3$ ,  $g(n) = n$ , y  $n_1 = 20 \therefore f(n) = 2n \in O(n)$ .

Para División2:  
Veamos en ejecución este algoritmo.



```

./Div2
5 1
La división entre 5 y 1 es: 5 con residuo: 0
El algoritmo Division2 hizo: 30 operaciones.

```

Figura 5. Ejecución de Division2

Ahora veamos su complejidad temporal a *priori*.

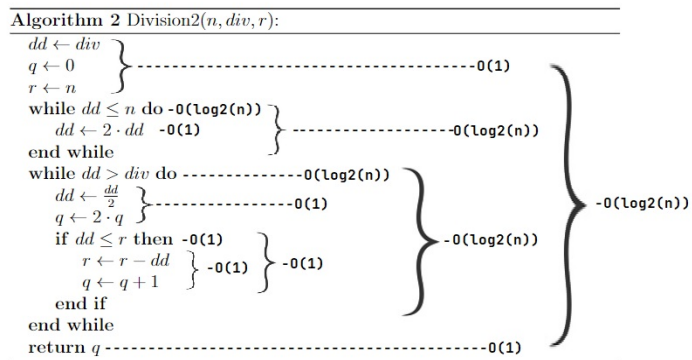


Figura 6. Análisis a *priori* de Division2

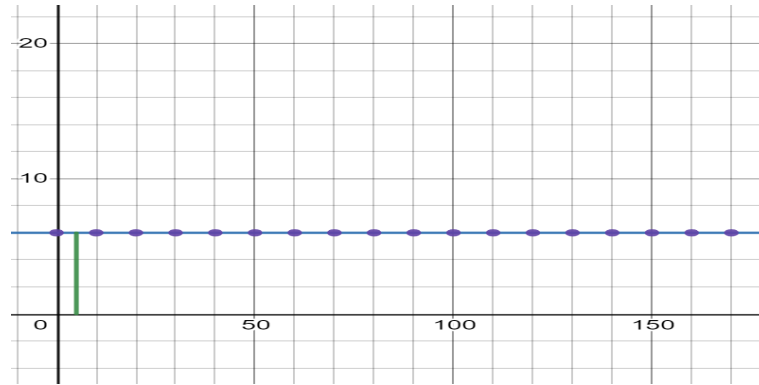
Ahora realizaremos el análisis a *posteriori*  
Para el mejor caso:



n	# operaciones
0	6
10	6
20	6
30	6
40	6
50	6
60	6
70	6
80	6
90	6
100	6

Table 3: Mediciones para el mejor caso de Division2

Así proponemos la función  $f(n) = 6$ .

Figura 7. Gráfica de la función  $f(n) = 6$ .

Vemos que la función pasa por todos los puntos. De la definición de  $\Omega$ :  
 Necesitamos encontrar un  $c_1 g(n) \leq 6 \forall n \geq n_1$ , de ahí que:  
 $5(1) \leq 6 \forall n \geq 1$ , entonces  $c_1 = 5$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 6 \in \Omega(1)$ .

Para el peor caso:

n	# operaciones
0	6
10	36
20	42
30	48
40	48
50	51
60	54
70	57
80	54
90	60
100	57

Table 4: Mediciones para el peor caso de Division2

Así proponemos la función  $f(n) = 26 \log_2(n)$ .

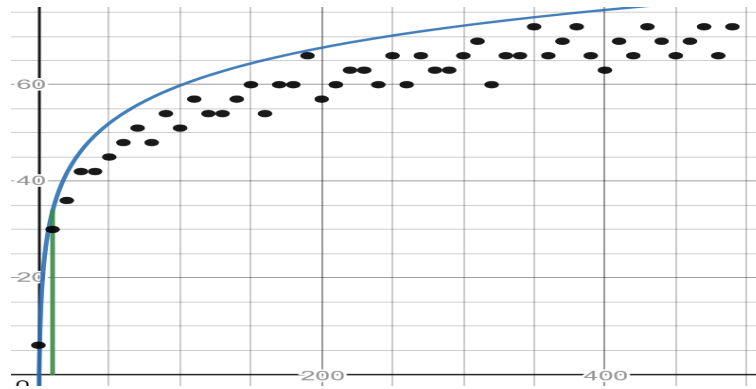


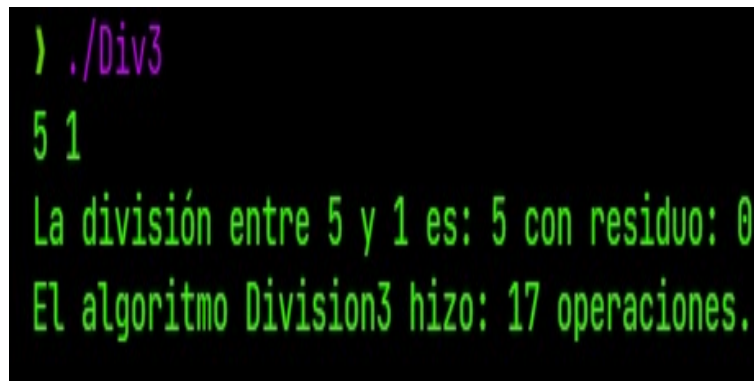
Figura 8. Gráfica de la función  $f(n) = 26 \log_2(n)$ .

Podemos observar que algunos de estos puntos pasan por la función, y los demás están por debajo. De la definición de  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 26 \log_2(n) \forall n \geq n_1$ , de ahí que:

$26 \log_2(n) \leq 27 \log_2(n) \forall n \geq 10$ , entonces  $c_1 = 27$ ,  $g(n) = \log_2(n)$ , y  $n_1 = 10$   
 $\therefore f(n) = \log_2(n) \in O(\log_2(n))$ .

Para División3:  
Veamos en ejecución este algoritmo.



```

./Div3
5 1
La división entre 5 y 1 es: 5 con residuo: 0
El algoritmo Division3 hizo: 17 operaciones.

```

Figura 9. Ejecución de Division3

Ahora veamos su complejidad temporal *a priori*.

---

Algorithm 3 Division3( $n, div$ ):	
if $div > n$ then $\Theta(1)$	} $\Theta(1)$
return 0 $\Theta(1)$	
else	} $\Theta(n)$
return $1 + \text{Division3}(n - div, div)$ $\Theta(n-1) \Theta(n)$	
end if	

---

Figura 10. Análisis *a priori* de Division3

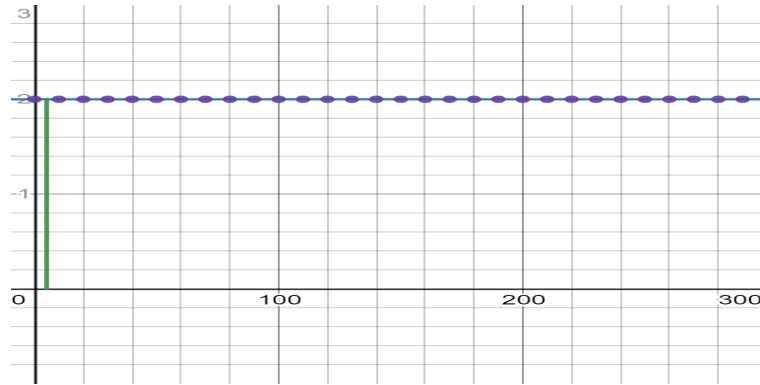
Ya que el peor caso es cuando  $div$  vale 1, entonces la recurrencia es del tipo Decremento por 1, por lo tanto es lineal.

Ahora realizaremos el análisis *a posteriori*  
Para el mejor caso:

n	# operaciones
0	2
10	2
20	2
30	2
40	2
50	2
60	2
70	2
80	2
90	2
100	2

Table 5: Mediciones para el mejor caso de Division3

Así proponemos la función  $f(n) = 2$ .

Figura 11. Gráfica de la función  $f(n) = 2$ .

Vemos que la función pasa por todos los puntos. De la definición de  $\Omega$ :  
 Necesitamos encontrar un  $c_1 g(n) \leq 2 \forall n \geq n_1$ , de ahí que:  
 $1(1) \leq 2 \forall n \geq 1$ , entonces  $c_1 = 1$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 2 \in \Omega(1)$ .

Para el peor caso:

n	# operaciones
0	2
10	17
20	32
30	47
40	62
50	77
60	92
70	107
80	122
90	137
100	152

Table 6: Mediciones para el peor caso de Division3

Así proponemos la función  $f(n) = 2n$ .

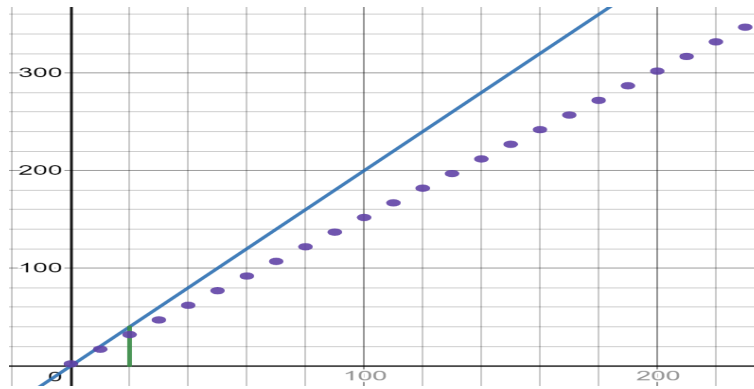


Figura 12. Gráfica de la función  $f(n) = 2n$ .

Podemos observar que algunos de estos puntos pasan por la función, y los demás están por debajo. De la definición de  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 2n \forall n \geq n_1$ , de ahí que:

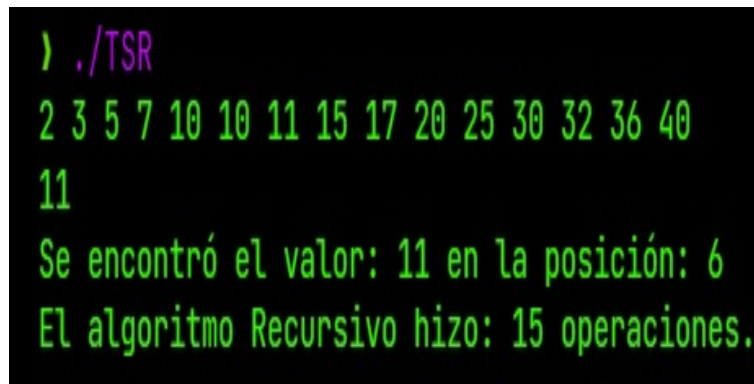
$2n \leq 3n \forall n \geq 20$ , entonces  $c_1 = 3$ ,  $g(n) = n$ , y  $n_1 = 20 \therefore f(n) = 2n \in O(n)$ .

Vemos que el algoritmo recursivo Division3, es mejor que el algoritmo iterativo Division1, aunque por poco, sin embargo, el algoritmo iterativo Division2, es el mejor de los 3.

Ahora veremos el segundo problema, de buscar un número en un arreglo de enteros. El tamaño del problema es el tamaño del arreglo, el mejor caso es cuando el valor que buscamos está en la posición del 1er tercio o en la posición del 2do tercio del arreglo, y el peor caso es cuando no está el valor en el arreglo.

Así ahora veremos este problema en su versión recursiva.

Veamos en ejecución este algoritmo.



```

) ./TSR
2 3 5 7 10 10 11 15 17 20 25 30 32 36 40
11
Se encontró el valor: 11 en la posición: 6
El algoritmo Recursivo hizo: 15 operaciones.

```

Figura 13. Ejecución de BusquedaEn3BloquesRecursiva

Ahora veamos su complejidad temporal *a priori*.

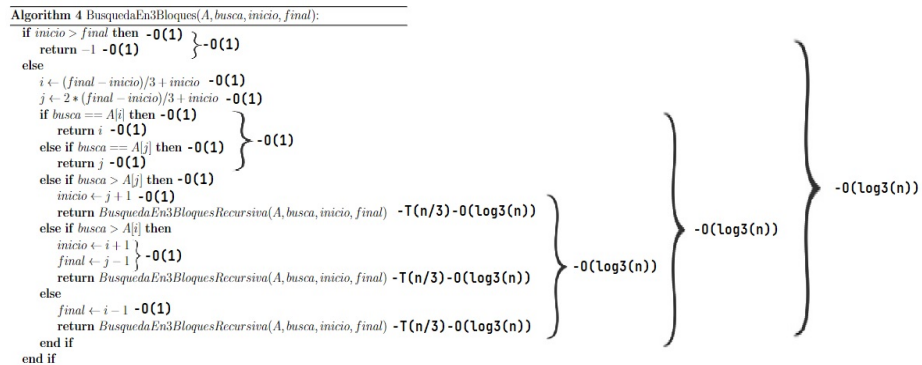


Figura 14. Análisis *a priori* de BusquedaEn3BloquesRecursiva

Esta es una recursión del tipo Decremento por un factor constante, por lo tanto su complejidad es  $\log_3 n$ .

Ahora realizaremos el análisis *a posteriori*

Para el mejor caso:

n	# operaciones
1	5
11	5
21	5
31	5
41	5
51	5
61	5
71	5
81	5
91	5
101	5

Table 7: Mediciones para el mejor caso de BusquedaEn3BloquesRecurSiva

Así proponemos la función  $f(n) = 5$ .

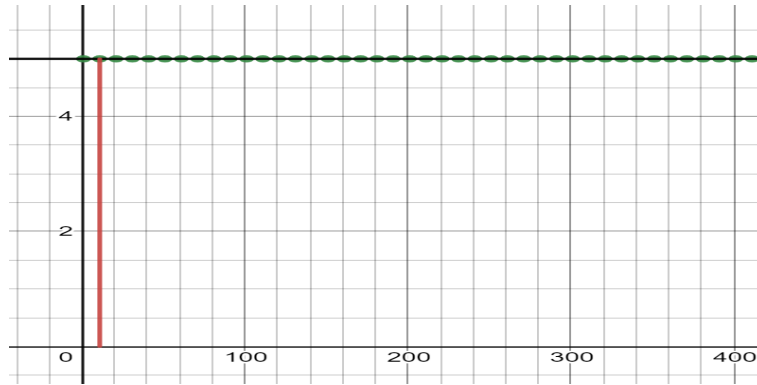


Figura 15. Gráfica de la función  $f(n) = 5$ .

Vemos que la función pasa por todos los puntos. De la definición de  $\Omega$ :  
 Necesitamos encontrar un  $c_1 g(n) \leq 5 \forall n \geq n_1$ , de ahí que:  
 $4(1) \leq 5 \forall n \geq 1$ , entonces  $c_1 = 4$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 5 \in \Omega(1)$ .

Para el peor caso:

n	# operaciones
1	11
11	28
21	29
31	28
41	42
51	29
61	38
71	39
81	45
91	42
101	38

Table 8: Mediciones para el peor caso de BusquedaEn3BloquesRecurсивa

Así proponemos la función  $f(n) = 20 \log_3(n)$ .

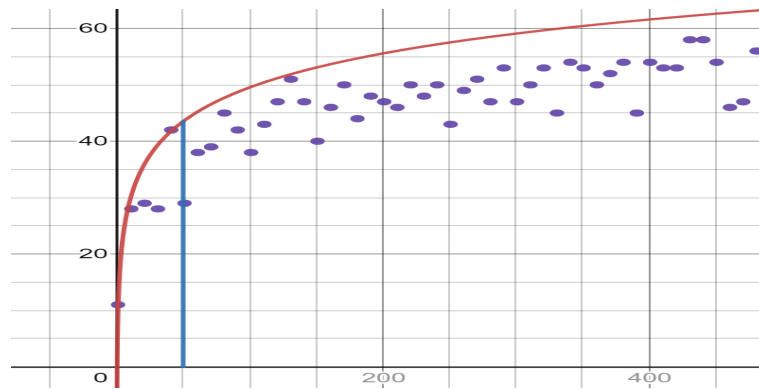


Figura 16. Gráfica de la función  $f(n) = 20 \log_3(n)$ .

Podemos observar que algunos de estos puntos pasan por la función, y los demás están por debajo. De la definición de  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 20 \log_3(n) \forall n \geq n_1$ , de ahí que:

$20 \log_3(n) \leq 21 \log_3(n) \forall n \geq 50$ , entonces  $c_1 = 21$ ,  $g(n) = \log_3(n)$ , y  $n_1 = 50$   
 $\therefore f(n) = 20 \log_3(n) \in O(\log_3(n))$ .



Ahora vamos a ver su versión iterativa.  
Veamos en ejecución este algoritmo.

```

) ./TSI
2 3 5 7 10 10 11 15 17 20 25 30 32 36 40
11
Se encontró el valor: 11 en la posición: 6
El algoritmo Iterativo hizo: 18 operaciones.

```

Figura 17. Ejecución de BusquedaEn3BloquesIterativa

Ahora veamos su complejidad temporal a *priori*.

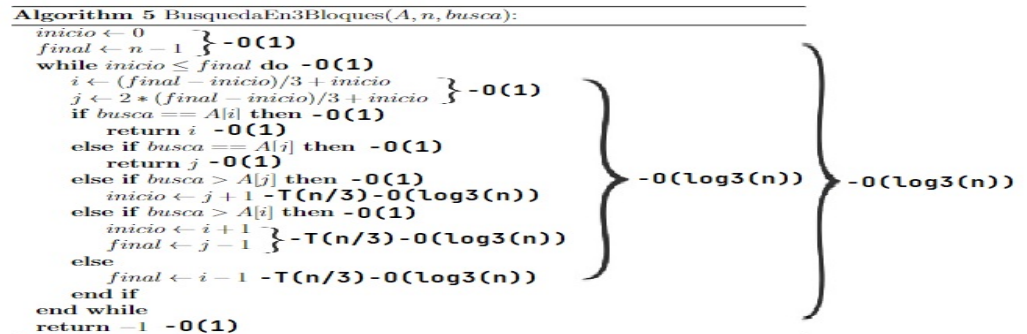


Figura 18. Análisis a *priori* de BusquedaEn3BloquesIterativa

Este algoritmo genera la misma recurrencia, cabe recordar que las recurrencias no son propias de los algoritmos recursivos. Entonces al ser la misma que la del caso recursivo, será la misma complejidad también.

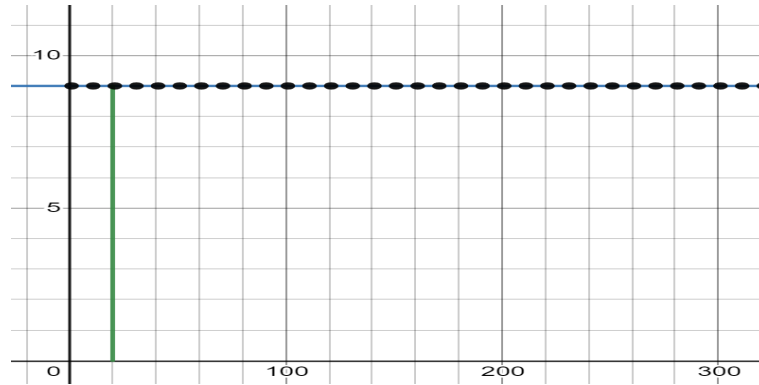
Ahora realizaremos el análisis a *posteriori*

Para el mejor caso:

n	# operaciones
1	9
11	9
21	9
31	9
41	9
51	9
61	9
71	9
81	9
91	9
101	9

Table 9: Mediciones para el mejor caso de BusquedaEn3BloquesIterativa

Así proponemos la función  $f(n) = 9$ .

Figura 19. Gráfica de la función  $f(n) = 9$ .

Vemos que la función pasa por todos los puntos. De la definición de  $\Omega$ :  
 Necesitamos encontrar un  $c_1 g(n) \leq 9 \forall n \geq n_1$ , de ahí que:  
 $8(1) \leq 9 \forall n \geq 1$ , entonces  $c_1 = 8$ ,  $g(n) = 1$ , y  $n_1 = 1 \therefore f(n) = 9 \in \Omega(1)$ .

Para el peor caso:

n	# operaciones
1	13
11	27
21	34
31	34
41	41
51	41
61	41
71	41
81	41
91	41
101	41

Table 10: Mediciones para el peor caso de BusquedaEn3BloquesIterativa

Así proponemos la función  $f(n) = 19 \log_3(n)$ .

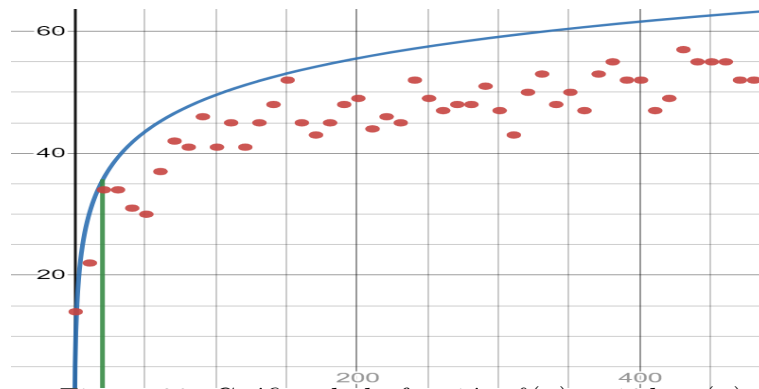


Figura 20. Gráfica de la función  $f(n) = 19 \log_3(n)$ .

Podemos observar que algunos de estos puntos pasan por la función, y los demás están por debajo. De la definición de  $O$ :

Necesitamos encontrar un  $c_1 g(n) \geq 19 \log_3(n) \forall n \geq n_1$ , de ahí que:

$19 \log_3(n) \leq 20 \log_3(n) \forall n \geq 50$ , entonces  $c_1 = 20$ ,  $g(n) = \log_3(n)$ , y  $n_1 = 50$   
 $\therefore f(n) = 19 \log_3(n) \in O(\log_3(n))$ .

Vemos que ambos algoritmos tienen la misma complejidad, sin embargo el recursivo hace menos operaciones para el mejor caso, pero el iterativo parece mejor cuando el peor caso, así que para decir cuál sería mejor, se podría decir que el iterativo, puesto que no ocupa espacio extra, y es mejor para el peor caso, que es probable que se presente más veces que el mejor caso.

## 4 Conclusiones

### Conclusión General

Para encontrar las complejidades de algunos ciclos de los algoritmos fue complicado observar cuál era la complejidad, hasta que se encontró cuál era el peor caso y mejor caso, y tratarlos de manera separada. Los algoritmos funcionan bien, así que podemos decir que los resultados fueron los esperados, todos tienen complejidades aceptables, aunque quizás el algoritmo de búsqueda en 3 bloques podría ser mejor si fuera  $\log_2 n$  puesto que crece menos rápido que  $\log_3 n$ .

### Conclusión de Anthony

Esta práctica en lo personal me gustó bastante, los temas fueron realmente interesantes, creo que resolver recurrencias puede llegar a ser divertido, es un tema muy interesante, en especial las recurrencias no homogéneas, puesto que se pueden encontrar recurrencias, y resolverlas, que probablemente no puedan ser vistas en algún algoritmo, y regresa esta abstracción tan interesante que se encuentra regularmente en las matemáticas, de resolver cosas tan abstractas que no puedan ser vistas en práctica, o que no puedan ni imaginarse. Y aunque generalmente los algoritmos recursivos ocupan mucha memoria, me llama mucho la atención lo elegante que se puede llegar a escribir un problema, cuando se representa de forma recursiva; y el hecho de existan muchas funciones recursivas de distintas categorías que pueden resolver el problema, me da mucha curiosidad.



## 5 Anexo

Para este caso no se realizaron problemas de tarea para anexar.

## 6 Bibliografía

Dillig, I. CS 311H. Revisado Marzo 27, 2022, de [https : //www.cs.utexas.edu/ isil/cs311h/](https://www.cs.utexas.edu/isil/cs311h/)  
Rubio-Sanchez, M, (2018). Introduction to Recursive Programming. CRC  
Press.