Instituto Politécnico Nacional
Escuela Superior de Cómputo

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

Práctica 3: Funciones Recursivas vs Iterativas

Luis Francisco Renteria Cedillo, Denzel Omar Vazquez Perez.

*lrenteriac*1400@*alumno.ipn.mx*, *dvazquezp*1600@*alumno.ipn.mx*

Resumen: Se realiza el análisis y comparaciones entre algoritmos recursivos e iterativos, con el objetivo de determinar el grado de complejidad para cada uno de ellos. Se llevaron a cabo análisis a priori y posteriori, para determinar el orden de complejidad de manera teórica y práctica.

Palabras Clave: Lenguaje C, Análisis, Recursividad, Iterativo, Algoritmos

1 Introducción

Dentro de los lenguajes de programación, las estructuras de control permiten establecer un flujo de datos según las instrucciones de un programa.Un ciclo o bucle es ampliamente usado para definir un conjunto de instrucciones que se ejecutaran n numero de veces hasta que una condición de control lo permita.

Con la llegada de problemas más complejos y extensos, llegó la programación modular, que utiliza la técnica de "Divide y Vencerás", donde su principal objetivo es dividir el problema en pequeños módulos para facilitar su posterior solución individual. Dichos módulos, en la práctica, se les conoce comúnmente como funciones, métodos o procedimientos.

Un algoritmo puede implementarse en un lenguaje de programación como una función, donde esta debe de definirse con un nombre, un tipo de dato a regresar, un conjunto de parámetros o argumentos y un conjunto de instrucciones a ejecutar. Es aquí cuando técnicas como recursividad o iteración son utilizados para resolver diversos problemas, mismos que serán explicados en la siguiente sección.

2 Conceptos teóricos

2.1 Algoritmos Recursivos

Un algoritmo recursivo utiliza la técnica de "divide y vencerás", donde el problema principal se divide en varios subproblemas que tienen características muy similares.

La recursividad está estrechamente relacionado con funciones recurrentes e incluso fractales, ya que la recursividad implica construir o generar, a partir de características de si mismo o mismo tipo.

En ciencias de la computación, la recursividad es un proceso en donde uno de sus pasos es invocar al mismo proceso. Esta técnica es un pilar fundamental para otras técnicas como lo es la programación dinámica.

2.2 Algoritmos Iterativos

Un algoritmo iterativo es aquel que se ejecuta dentro de un ciclo mientras que un conjunto de condiciones lo permitan. A comparación de un algoritmo recursivo, un algoritmo iterativo es generalmente más extenso en su escritura.

2.3 Pseudocódigos de los problemas a analizar

Nuestro primer problema a analizar serán tres algoritmos diferentes que calculan el cociente de una división de números enteros. Cada función tiene una forma diferente de llegar al mismo resultado, ya sea usando funciones recursivas o iterativas.

A continuación se mostrarán los pseudocódigos de dichas funciones:

Algorithm 1 Division1

Input: *int n*, *int div*, *int res*
Output: cociente(n/div)

```
1: q = 0
2: while n ≥ div do
3:     n = n-div
4:     q = q+1
5: end while
6: res = n
7: return q
```

Algorithm 2 Division2

Input: *int n*, *int div*, *int res*
Output: cociente(n/div)

```
1: dd = div
2: q = 0
3: r = n
4: while dd ≤ n do
5:     dd = (2)(dd)
6: end while
7: while dd > div do
8:     dd = dd/2
9:     q = (2)(q)
10:    if dd ≤ r then
11:        r = r - dd
12:        q = q + 1
13:    end if
14: end while
15: return q
```

Algorithm 3 Division3

Input: *int n*, *int div*, *int res*
Output: cociente(n/div)

```
1: if div > n then
2:     return 0
3: else
4:     return 1+Division3(n-div,div)
5: end if
```

La segunda parte del problema a analizar serán dos algoritmos de búsqueda, uno recursivo y otro iterativo, que tienen como objetivo encontrar un numero K en un arreglo, esto al dividir este en tres partes iguales y comparar con los extremos interiores i y j.

A continuación se mostrarán los pseudocódigos de dichas funciones:

Algorithm 4 search_num

Input: *int A[], int n, int inicio, int final*
Output: Número natural

```
1: while inicio≤final do
2:     i = inicio + ((final-inicio)/3)
3:     j = inicio + (2*((final-inicio)/3))+1
4:     if n==A[i] then
5:         return i
6:     else if n==A[j] then
7:         return j
8:     else if n<A[i] then
9:         final = i-1
10:    else if n>A[j] then
11:        inicio = j+1
12:    else
13:        final = j-1;
14:        inicio = i+1;
15:    end if
16: end while
17: return -1
```

Algorithm 5 search_num_R

Input: *int A[], int n, int inicio, int final*
Output: Número natural

```
1: i = inicio + ((final-inicio)/3)
2: j = inicio + (2*((final-inicio)/3))+1
3: if inicio>final then
4:     return -1
5: end if
6: if n==A[i] then
7:     return i
8: else if n==A[j] then
9:     return j
10: else if n<A[i] then
11:    search_num_R(A, n, inicio, i-1)
12: else if n>A[j] then
13:    search_num_R(A, n, j+1, final)
14: else
15:    search_num_R(A, n, i+1, j-1);
16: end if
```

3 Experimentación y resultados

3.1 Algoritmos de la división

3.1.1 División 1

3.1.1.1 Análisis a Priori

En la Figura 1 podemos observar la implementación de la función división 1. Empleando las definiciones formales llegamos a concluir que las sentencias simples fuera del ciclo while tienen una complejidad de O(1), mientras que la del propio ciclo while tiene complejidad de O(n), debido a que en cada iteración, a n se le resta el valor de div y este ciclo se repite hasta que n sea mayor a div.

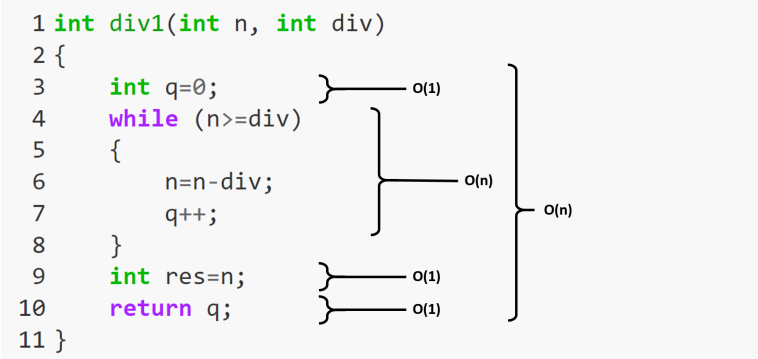

Figura 1: Análisis por bloques de código de la función div1().

3.1.1.2 Análisis a Posteriori

En el siguiente gráfico que se muestra en la Figura 2 representa los conjuntos de pares ordenados los cuales representan el valor del cociente vs numero de ejecuciones. Dichos puntos se han obtenido al generar números aleatorios en el rango del 1 al 100 en el dominio de los números enteros. Dado el comportamiento de la gráfica, se verifica que es lineal, no habiendo diferencia entre el mejor y peor caso, y retomando que Θ se define como:

$$\Theta(g(n)) = \{f(n) \mid \exists_n\ C_1,\ C_2 > 0\ \&\ n_0 > 0\ tal\ que$$

$$0 \leq\ C_1 g(n)\ \leq\ f(n)\ \leq\ C_2 g(n)\ \forall\ n \geq\ n_0\}$$

Se proponen valores para $C_1 = 2$, $C_2 = 4$ y $n_0 = 10$. Dichos valores se acotan la parte superior como la inferior así como el punto de cruce en el que se cumple la inecuación, por tanto se puede decir que $T(n) \in \Theta(n)$.

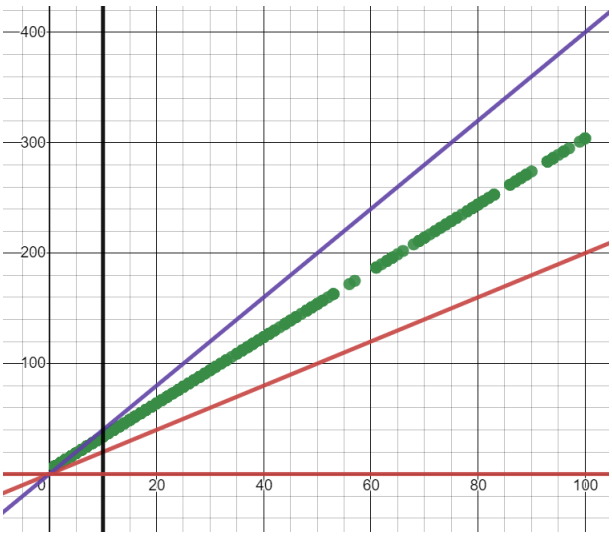

Figura 2: Gráfico del peor y mejor caso para la función div1().

3.1.2 División 2

3.1.2.1 Analisis a Priori

La Figura 3 muestra la implementación del código, se observa que las sentencias simples tienen complejidad O(1) mientras que las estructuras de control while tienen complejidad $O(log_2(n))$.

```
1 int div2(int n, int div){
2     int dd=div;
3     int q=0;
4     int r=n;
5
6     while(dd<=n)
7         dd=2*dd;
8     while(dd>div)
9     {
10        dd=dd/2;
11        q=2*q;
12        if(dd<=r)
13        {
14            r=r-dd;
15            q=q+1;
16        }
17    }
18    return q;
19 }
```

O(1)
O(1)
O(1)
O(log₂(n))
O(log₂(n))
O(log₂(n))
O(1)

Figura 3: Análisis por bloques de código de la función div2().

3.1.2.2 Análisis a Posteriori

En el siguiente gráfico (Figura 4) se muestra el gráfico de los valores obtenidos. El comportamiento de la gráfica es logarítmico, sin diferencia entre el mejor o el peor caso. Se proponen valores para $C_1 = 4$, $C_2 = 12$ y $n_0 = 10$. Dichos valores se acotan la parte superior como la inferior así como el punto de cruce en el que se cumple la desigualdad, por tanto $T(n) \in \Theta(log_2(n))$.

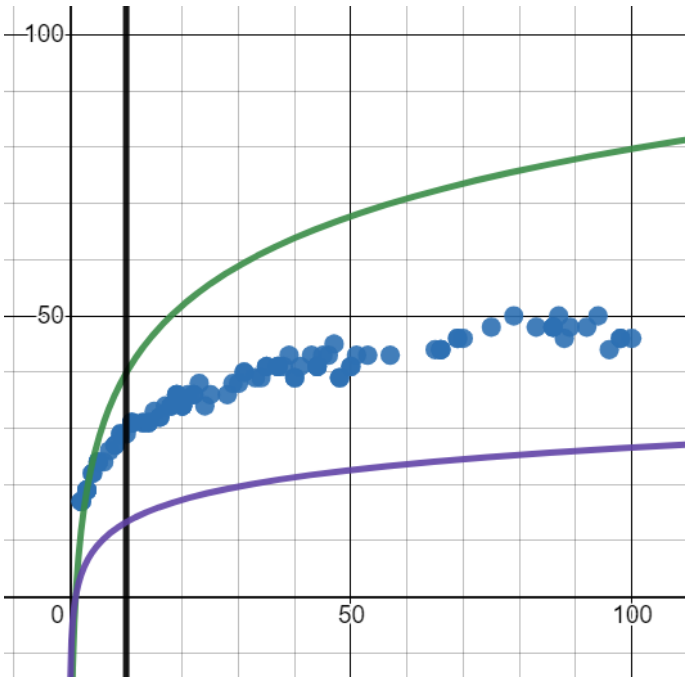

Figura 4: Gráfico del peor y mejor caso para la función div2().

3.1.3 División 3

3.1.3.1 Análisis a Priori

En la Figura 5 se muestra la implementación de la función división 3. Observamos que es una función recursiva que consiste en una sentencia if, la cual, si se cumple regresa un valor entero, pero en caso contrario, se llama a la misma función con uno de sus argumentos decrementado en uno respecto al original y a ese resultado se le suma un valor entero. Notamos que la sentencia recursiva tiene una complejidad de $T(n) = T(n-1) + O(1)$, y se analiza a continuación:

$$T(n) = \begin{cases} C & si\ div > n \\ T(n-1) + C & si\ div \leq n \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + C \\ &= [T(n-2) + C] + C \\ &= T(n-2) + 2C \\ &= [T(n-3) + C] + 2C \\ &= T(n-3) + 3C \\ &\vdots \\ &= T(n-i) + i \end{aligned}$$

Dado n - i = 0 $\Rightarrow$ i=n. Entonces:

$$\begin{aligned} &= T(0) + n \\ &= 0 + n \\ &= n \end{aligned} \tag{1}$$

$\therefore T(n) \in \Theta(n)$ Para el mejor y peor caso

```
1 int div3(int n, int div)
2 {
3     if (div>n)
4         return 0;
5     else
6         return 1+div3(n-div,div);
7 }
```

O(1)

T(n-1) + O(1)

Figura 5: Análisis por bloques de código de la función div3().

3.1.3.2 Análisis a Posteriori

A continuación se muestra el gráfico de los valores obtenidos. El comportamiento de la gráfica es lineal, sin diferencia entre el mejor o el peor caso. Se proponen valores para $C_1 = 1$, $C_2 = 3$ y $n_0 = 10$. Dichos valores se acotan la parte superior como la inferior así como el punto de cruce en el que se cumple la desigualdad, por tanto, el algoritmo de la división 3, $T(n) \in O(n)$.

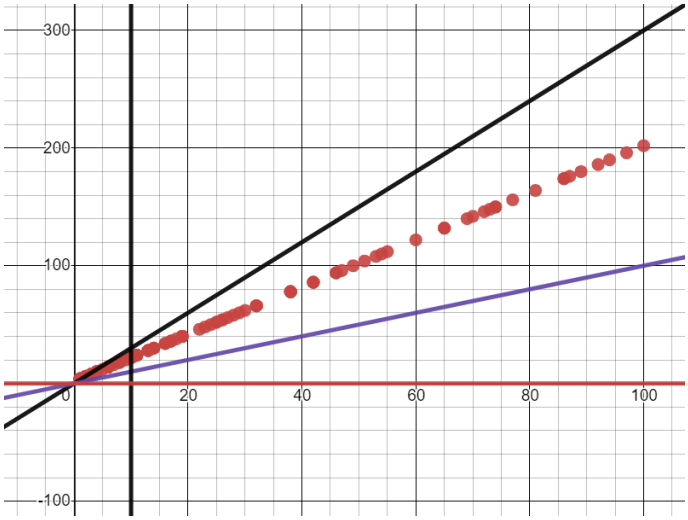

Figura 6: Gráfico del peor y mejor caso para la función div3().

```
PS D:\Desktop\ALG> & .\"Practica-3-1-b.exe"
Div1(1/1) = 1
Div2(1/1) = 1
Div3(1/1) = 1
Div1(26/5) = 5
Div2(26/5) = 5
Div3(26/5) = 5
Div1(82/20) = 4
Div2(82/20) = 4
Div3(82/20) = 4
```

Figura 7: Salida del programa al ejecutar div1(), div2() y div3() con los mismas entradas.

Comparando la complejidad de los tres algoritmos anteriores, el algoritmo de división 2 es el más eficiente dado que tiene complejidad logarítmica. Sin embargo, cabe destacar que este algoritmo es el más extenso e implica que

posiblemente sea más difícil de comprender por parte del programador, lo que puede llegar a sugerir realizar una buena documentación del algoritmo.

3.2 Búsqueda de un elemento en un arreglo en bloques de 3 en 3.

El siguiente algoritmo hace la búsqueda de un elemento k en un arreglo de tamaño n, basándose en búsqueda binaria sin embargo se parte en tres bloque iguales a este, donde dos pivotes i, j son lo puntos para determinara si el número se encuentra dentro por medio de comparaciones en cuanto si es mayor, menor o igual a dichas posiciones. Se realizo dos versiones de este algoritmo una iterativa y otra recursiva, en seguida se muestra su análisis a *priori* y a *posteriori* de cada uno, para determinan cual de estos dos en mas eficiente.

3.2.1 Versión iterativa

Para esta versión de búsqueda se implementa el algoritmo en el lenguaje de programación C.

3.2.1.1 Analisis a Priori

Por medio de definiciones formales se obtendrá la complejidad del algoritmo dado un análisis por secciones de código, determinando el tiempo de solución del problema planteado.
Sin embargo, surge una pregunta , ¿Cómo obtener la complejidad del **while**?, esta saldra del peor caso que se logre obtener en la reducción del arreglo, siendo que un parámetro importante es la condición de la línea 3 de código donde se busca que se cumpla que $inicio \leq final$, por lo que para el peor caso el tamaño del arreglo A[] es de 1, así mismo la reducción de este se presenta en la Tabla 1.

Tabla 1: Complejidad temporal del algoritmo iterativo

Iteración	**Peor caso T(n)**
0	n
1	$\frac{n}{3}$
2	$\frac{n}{9}$
3	$\frac{n}{27}$
4	$\frac{n}{81}$
$\vdots$	$\vdots$
k	$\frac{n}{3^K}$

Conociendo que el tamaño del arreglo para el pero caso es 1 y dada la gener-

alización del peor caso para las iteraciones del while el orden del complejidad se obtiene de la siguiente igualdad.

$$\frac{n}{3^K} = 1$$

$$entonces,\ n = 3^K$$

$$luego,\ log_3(n) = log_3(3^K)$$

$$\therefore\ log_3(n) = K$$

Dado cada bloque de sentencia del algoritmo, se muestra la obtención del orden de complejidad para el peor caso de este por medio de un análisis de segmentos de código , véase Figura 8, donde se muestra que T(n)$\in O(log_3(n))$.

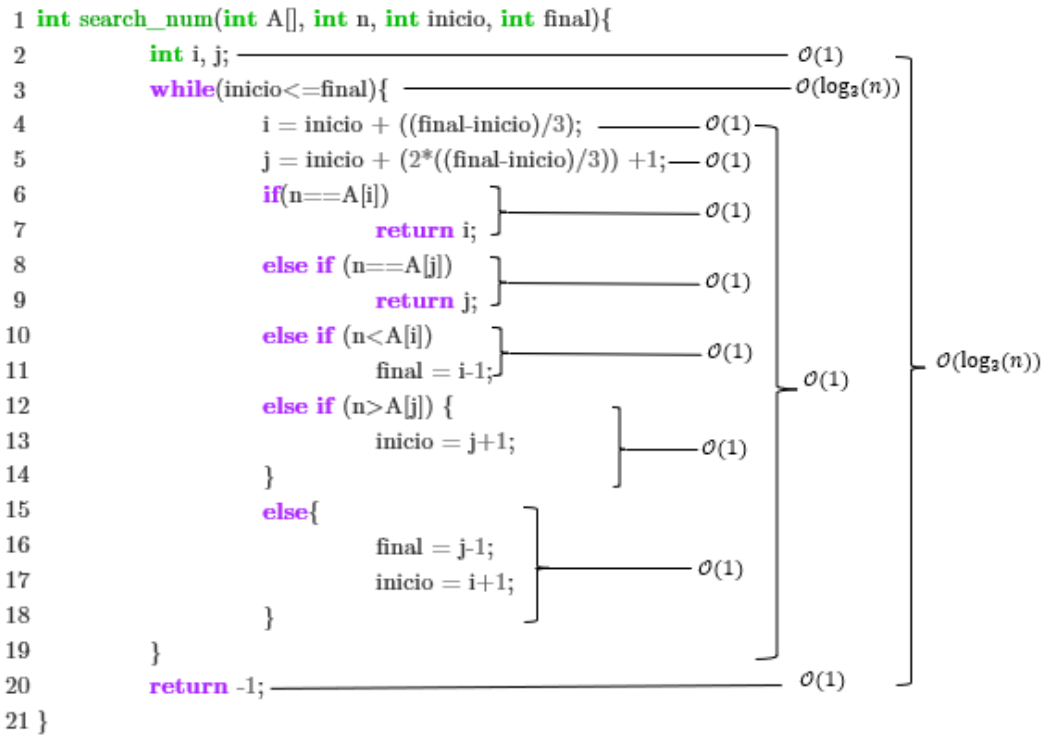

Figura 8: Análisis del por bloque de código del algoritmo iterativo

3.2.1.2 Analisis a Posteriori

Al ejecutar la función con el nombre *"search_num"*, se tiene que la asignación de valores de entrada esta dada por un arreglo A[] y tres enteros que definen el inicio, el final y el número n a buscar dentro de A[], por tanto al finalizar la

ejecución se obtiene el número de operaciones que se realizan ante la búsqueda de n. Posterior a estos se logra obtener los tiempos de ejecución de tales búsquedas, los cuales se muestran en la Figura 9.

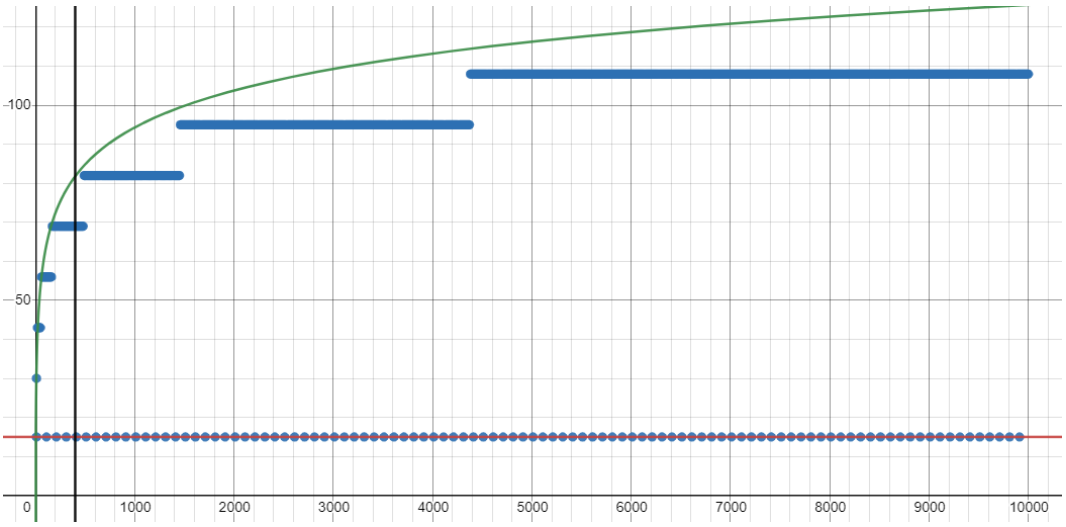

Figura 9: Gráfico del peor y mejor caso para la versión iterativa del algoritmo.

Ante el resultado de los puntos prueba se determina que el mejor caso es continuo por lo que se puede afirmar que $T(n) \in \Omega(1)$, sin embargo para el peor caso se muestra que el crecimiento de los puntos de prueba tiene un comportamiento logarítmico por lo que recordando la definición de $\mathcal{O}$:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists_n\ C_1 > 0\ \&\ n_1 > 0\ tal\ que$$

$$0 \leq\ f(n)\ \leq\ C_1 g(n)\ \forall\ n \geq\ n_1\}$$

Se tiene que $f(n)$ es la función que acota los resultados no encontrados en A[], sin embargo no se conoce el orden de complejidad de este algoritmo, por lo que proponiendo a $g(n) = log_3(n)$, $C_1 = 20$ y $n_1 = 200$ se puede observar que acota por arriba a $f(n)$ por tanto se puede decir que $T(n) \in \mathcal{O}(log_3(n))$, entonces el algoritmo presenta un orden de complejidad logarítmica.

3.2.2 Versión recursiva

La versión recursiva de este algoritmo es implementada en el lenguaje de programación C.

3.2.2.1 Análisis a Priori

La versión recursiva del algoritmo analizado muestra las mismas condiciones

para el peor caso de la anterior sección, donde este se presenta cuando el elemento K no se encuentra en un arreglo A[].

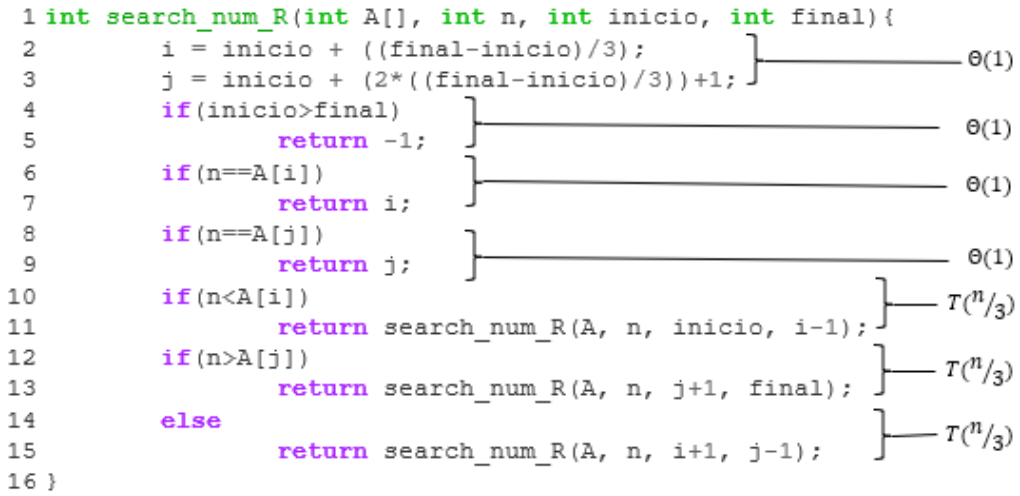

Figura 10: Análisis del por bloque de código del algoritmo recursivo

Observando el comportamiento del algoritmo recursivo, divide en 3 partes el arreglo A[], deteniéndose cuando se cumpla la condición $inicio > final$ donde la tamaño del A en este instante es de 1, así mismo cada una de las sentencias condicionales se vuelven constantes por lo que la ecuación de recurrencia que representa este comportamiento esta dado por:

$$T(n) = \begin{cases} C & si \quad n = 0 \\ T(\frac{n}{3}) + C & si \quad n > 0 \end{cases}$$

Ahora, sea $n = 3^k \Rightarrow k = \log_3(n)$ se tiene que

$$T(3^k) = \begin{cases} C & si \quad k = 0 \\ T(3^{k-1}) + C & si \quad k > 0 \end{cases}$$

Resolviendo por sustitución hacia atrás se tiene

$$
\begin{aligned}
T(3^k) &= T(3^{k-1}) + C \\
&= (T(3^{k-2}) + C) + C = T(3^{k-2}) + 2C \\
&= (T(3^{k-3}) + C) + 2C = T(3^{k-3}) + 3C \\
&= (T(3^{k-4}) + C) + 3C = T(3^{k-4}) + 4C \\
&\vdots \\
&= T(3^{k-i}) + iC \\
\textit{para } K = i \Rightarrow K - i = 0& \\
&= T(3^0) + kC \\
&= T(1) + kC \\
&= C + kC \\
&= C + \log_3(n)C \\
\therefore T(n) \in \mathcal{O}(\log_3(n))&
\end{aligned}
$$

3.2.2.2 Análisis a Posteriori

Cuando se se hace uso de la función "*search_num_R*", al igual que en su versión iterativa, se obtiene los tiempos de ejecución $T(n)$ ante la búsqueda de un número n en el arreglo A[]. Al ser graficados estos puntos, muestran el comportamiento que se visualiza en la Figura 11.

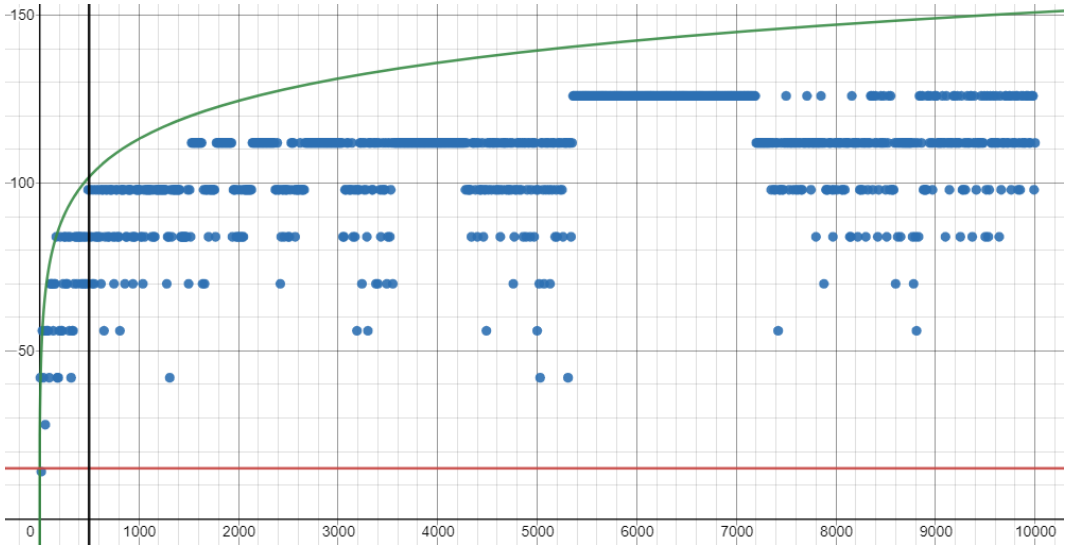

Figura 11: Gráfico del peor y mejor caso para la versión recursiva del algoritmo.

Al analizar el patrón de crecimiento ante el tamaño del problema se muestra

que para el mejor caso el orden de complejidad del algoritmo es constante, así que $T(n) \in \Omega(1)$. Por lo que al identificar los puntos prueba que pertenecen a $f(n)$ y que no lograron ser encontrados en A[] muestran un comportamiento logarítmico, entonces dada la definición de $\mathcal{O}$:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists_n\ C_1 > 0\ \&\ n_1 > 0\ tal\ que$$

$$0 \leq\ f(n)\ \leq\ C_1 g(n)\ \forall\ n \geq\ n_1\}$$

Se propone $g(n) = log_3(n)$, $C_1 = 16$ y $n_1 = 500$ lo cual es una cota superior para $f(n)$ por lo que se puede decir que $T(n) \in \mathcal{O}(log_3(n))$, entonces el algoritmo presenta un orden de complejidad logarítmica.

4 Conclusiones

Luis Francisco Renteria Cedillo

Al desarrollar esta práctica, se logró obtener la complejidad computacional de varios algoritmos equivalentes. Un dato importante a mencionar es que las funciones recursivas son más fáciles de comprender que algunas funciones iterativas equivalentes, en especial cuando los algoritmos son cortos, ya que se tienen contempladas las acciones a realizar en cada caso y sabemos que finaliza al llamar la instrucción return junto con una constante. En el caso de que la instrucción return llame a la misma función, es fácil de comprender los argumentos implicados en ella dado que tienen una variación definida que obedece a la definición recurrente de la función

Denzel Omar Vazquez Perez

La forma en la que se soluciona y codifica problemas, son aspectos que influyen en la determinación del orden de complejidad que tomara dicho algoritmo, cada uno de los problemas en la practica se tenia visto que se analizaría algoritmos iterativos y recursivos del mismo orden sin embargo, el segundo algoritmo "*Division2*" mostró tener un orden de cota inferior al de los otros dos por lo se tiene una comparativa una mejor alternativa a los que es la solución iterativa y recursiva tradicional. Para el segundo problema una de la peculiaridades que se tuvo, sea la solución iterativa o recursiva tendrán ambas la misma complejidad por lo que cualquiera que se emplee sera indistinto para resolver el problema de la búsqueda del numero K en el arrglo A[].

5 Bibliografía

Brassard, G. (1997). *Fundamentos de Algoritmia*. España: Ed. Prentice Hall.

Cormen, E. A. (2022). *Introduction To Algorithms*, 3Rd Ed. Phi.

Luna, Benjamín. *Fundamentos para el análisis de eficiencia algorítmica*. Escuela Superior de Computo, IPN. México. 03 de abril de 2022.

Sánchez, P. J. I. (2006). *Análisis y diseño de algoritmos: un enfoque teórico y práctico*. Servicio de Publicaciones y Divulgación Científica de la UMA.