Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem.: 2022-1, 3CV11, Práctica 2, 9 octubre 2021

Práctica 3: Funciones Recursivas vs Iterativas

Vanegas García Andrés, Vaquera Aguilera Ethan Emiliano.

*avanegasg*1601@*alumno.ipn.mx*, *evaqueraa*@*alumno.ipn.mx*

Resumen: Se realizo el primer ejercicio que consistió en la realización de tres programas que en teoría se supone que son los mismo, pero solo que cambia el método de calculo de estos, ya que usan iteración los dos primeros, solo que usa una comprobación de cálculos los el segundo, y el ultimo usa la recursividad para calcular el cociente del numero en cuestión. Para el segundo se realizo un programa de la misma manera solo que el anterior buscando la solución a la búsqueda de un numero Num dentro de un arreglo de tamaño N, usando de primeras interacciones para la búsqueda de este numero y después hacer la misma implementación para solucionar el problema pero ahora con recursividad.

Palabras Clave

C: C es un lenguaje de programación estructurado, usado por ya poco más de 52 años, esto brindándole un amplio repertorio de bibliotecas, además de un gran soporte de la comunidad que lo utiliza. C al ser un lenguaje de programación estructurado comparte similitudes con otros lenguajes desarrollados en la época además de que es un lenguaje de alto nivel, que de igual manera pude ser utilizado a bajo nivel para programar en ensamblador y rutinas de memoria.

Complejidad de un Algoritmo: Este es el punto en el que se mide la complejidad de resolución de un algoritmo por medio de una computadora, poniendo énfasis en el uso de la memoria de este sistema además de, el tiempo que este toma para desarrollarlo. Esta desde el inicio de la computación es una problemática que ha ido evolucionando a modo de que hoy en día se utiliza el análisis de algoritmos para so resolución y determinar la complejidad de un algoritmo en especifico.

Análisis a priori y posterior: Según lo comprendido en clase. El análisis a priori en computación es analizar un algoritmo, de manera formalmente teórica y matemáticamente, primero haciendo un análisis de sus posibles pseudocódigos y después en base a esto realizar una formulación del comportamiento de este en base a una función matemática.

Por otro lado el análisis a posterior, es la recolección de datos por medio de la experimentación, que en el caso de un algoritmo es la programación de este para así tomar una serie de pruebas en base a las veces ejecutadas y la cantidad de veces que este algoritmo tarda para terminar.

Iteración: Una iteración es la forma de hacer una serie de pasos repetidas veces hasta llegar al resultado esperado en un rango esperado sin sobrepasar este, las iteraciones pueden ir de cualquier manera en la vida real siendo como realizar un ejercicio matemático de una raíz, una potencia numérica o un logaritmo, y en computación serian las sentencias de while, for, do-while, etc.

Recursividad: La recursividad consiste mas que nada en la realización iterativa de un mismo lenguaje descriptivo, pero sin tener que mandar a llamar a una función iterativa. En palabras mas simples se trata de el desarrollo de una función que se llama a si misma, y después de llegar a un resultado esta regresa de vez en vez al inicio y finalmente otorgar el resultado final.

1 Introducción

Hoy en día el uso de sistemas lógicos para la resolución de problemas en un tiempo récord es mas necesario que nunca, por ello el ser humano a creado varias formas de desarrollar problemas que requieren de cientos de cálculos para eso, hoy veremos dos de ellos que son las forma iterativa de un programa y su contra parte que es su forma recursiva.

Para cualquiera de estas dos formas como se dijo anteriormente ambas son la contra parte de la otra pero así mismo, ambas pueden ser expresadas una en forma de la otra, osea un programa que se desarrolla de manera recursiva puede ser programado de manera iterativa y vice versa, siendo que pueden tener el mismo comportamiento o, este cambie de manera muy drástica debido a que la manera en la que se implementa un programa puede ser totalmente diferente dependiendo de los requerimiento que necesites de manera iterativa a la manera recursiva.

Comúnmente los programas desarrollados de manera iterativa suelen tener ordenes de la forma $\Theta(n)$, donde la n se puede expresar en variados términos de la forma n^k, n, $\log n$, *etc*, por otro lado las funciones recursivas se expresan

de la forma $T(n)$, donde n se expresa de las misma forma que las iterativa, pero suele ser mas común la manera de la forma lineal (n), o la manera logarítmica $(\log n)$.

De manera resumida ambas funciones son eficientes y con una buena implementación cumplen su trabajo de manera eficiente, pero se logra diferenciar que las funciones recursivas son mas eficientes en el punto de que en lugar de usar la llamada a una función iterativa (que puede crear un orden n^k que es mas volátil y tardado para la computadora), se usa la propia función recursiva para medio de calcular en si misma lo que se necesita para la salida del programa, siendo que siempre puede decrecer y detenerse en un punto final, no como una función iterativa que para provocar que se detenga debes de usar una función de retorno o u break, ya que si no se usa eso en dado caso la función iterativa continuara hasta terminar con todos los valores que se le ingresan a esta.

2 Conceptos Básicos

Θ: La sigla griega Teta (Θ) es la denotación que describe de manera clara, el comportamiento similar de las funciones matemáticas del peor y mejor caso, esto siendo una referencia clara para hacer distinguir de Ω y O, que tiene definiciones diferentes.

Definición formal: $\Theta(\mathrm{g(n)}) = \{f(n) : existe\, n, c_1, c_2, > 0\, y\, n_0 > 0 \,||\, 0 \leq c_1 g(n \leq f(n) \leq c_2 g(n) \,\forall\, n \geq n_0\}$

O**:** Big O es la denotación usada para el conjunto de funciones que se acotan por encima de f(n), describiendo de manera clara el comportamiento da la función del peor caso, y denota los puntos experimentales de esta también.

Definición Formal: $O(g(n)) = \{f(n) : existen, c > 0 \,\&\, n_0 > 0\, constantes \,||\, 0 \leq f(n) \leq cg(n) \,\forall\, n \geq n_0\}$

Ω: La denotación Omega (Ω) es la denotación usada para acotar por abajo a la función f(n), siendo que es usada para describir al conjunto de funciones que están por debajo de la función experimental f(n), y de igual manera sirve para describir los puntos experimentales de igual manera.

Definición Formal: $\Omega(g(n)) = \{f(n) : existen, c > 0 \,\&\, n_0 > 0\, constantes \,||\, cg(n) \leq f(n) \,\forall\, n \geq n_0\}$

Recursividad: La recursividad es la serie de procesos definidos, en la cual se quiere la sucesión de varios factores y en vez de usar una función de la clase iterativa, se usan los mismos procesos ya definidos para crear esa sucesión de procesos, Imaginemos una muñeca de Matryoshkas (tipo de muñecas rusas),

cada que abrimos una para sacar una nueva estas son mas pequeñas y además, todas y cada una de de ellas son iguales, usando ese ejemplo podemos definir un poco mejor la función recursiva, donde cada vez que mandamos a llamar a la función los valores de esta pueden ir disminuyendo o aumentando, dependiendo de lo que se quiera solucionar, hasta llegar a una condición de frontera, y de igual manera a pesar de que los valores cambien, lo que nunca se modifica son los procesos de la función recursiva que se mantiene de la misma manera.

Algorithm 1 Algoritmo Ejemplo Recursivo

```
function RECURSIVO(n)
    a ← n
    if a ≠ 0 then
        return Recursivo(a ← a − 1)
    else
        return a
    end if
end function
```

Función Iterativa: Las funciones iterativas son ciclos del tipo iterativo donde se hace el uso de una repetición de pasos sucesivos para llegar a una conclusión finita del programa, comúnmente se usa para operaciones matemáticas, y el limite del numero de operaciones esta dado por la cantidad de valores que nosotros programadores queramos predisponerle al sistema.

Algorithm 2 Algoritmo Ejemplo Iterativo

```
function ITERATIVO(n)
    a ← n
    while a ≤ n do
        a ← a − 1
    end while
    return a
end function
```

3 Experimentación y Resultados

3.1 Primera Parte

En esta primera parte de la practica se realizo el análisis tres algoritmos (dos iterativos y un recursivo) que resuelven el mismo problema, para con base en ese análisis podamos realizar nuestras conclusiones.

3.1.1 Primer Algoritmo (Division1)

Para el análisis de este primer algoritmo iterativo se realizo como primer paso el análisis a priori que se muestra en la **Figura 1**. Este algoritmo tiene como peor caso cuando el divisor (div) es igual a 1 para cualquier numero y como mejor caso cuando el divisor es igual al dividendo (n = div), sin embargo en la **Figura 1** solo se muestra el análisis del peor caso debido a que es el mas interesante, ya que en el mejor caso el orden es contante como se mostrara mas adelante.

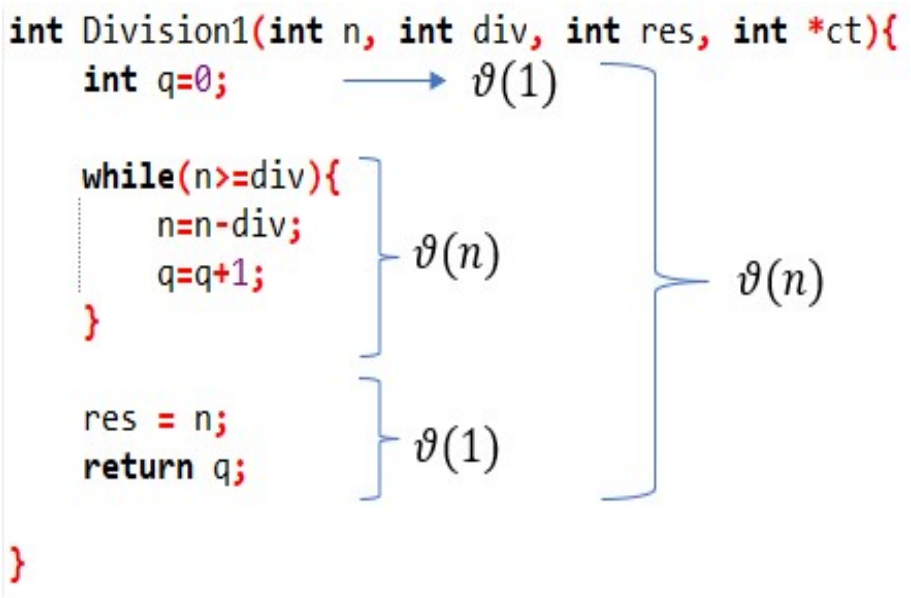


Figura 1. Análisis a priori Division1

La forma en la que se calculo el orden de complejidad del ciclo while fue la siguiente. Si tomamos n=40 y div=1 debido a lo anteriormente mencionado y analizando el incremento de n tenemos :

n	div
40	1
39	1
38	1
37	1
36	1
35	1
...	1

Si k es el numero de iteraciones.

$$n = -1 - 1 - 1 - ... = \sum_{0}^{k} -1 = -1 - k$$

Entonces el ciclo while termina cuando.

$$-1 - k < div \implies k \in \mathcal{O}(n)$$

Por lo tanto con el análisis y las reglas para el calculo de complejidad se tiene que el algoritmo para el peor caso tiene orden $\mathcal{O}(n)$. Ahora una vez realizado el análisis a priori se continuo con el análisis a posteriori donde se graficaron los primeros 100 numeros del 1 al 100 con un dividor de 1.

Figura 2. Analisis a posteriori de Divisor1

Como se observa en la **Figura 2** se corrobora que el algoritmo tiene un comportamiento asintótico lineal para el peor caso (color verde), mientras que para el mejor caso (color azul) permanece constante.

3.1.2 Segundo Algoritmo (Division2)

Observando el algoritmo se concluyo que el peor y mejor caso es el mismo que el algoritmos pasado, por que nuevamente para el análisis a priori solo se realizo para el peor caso, ya que el mejor caso va a tener un comportamiento contante.

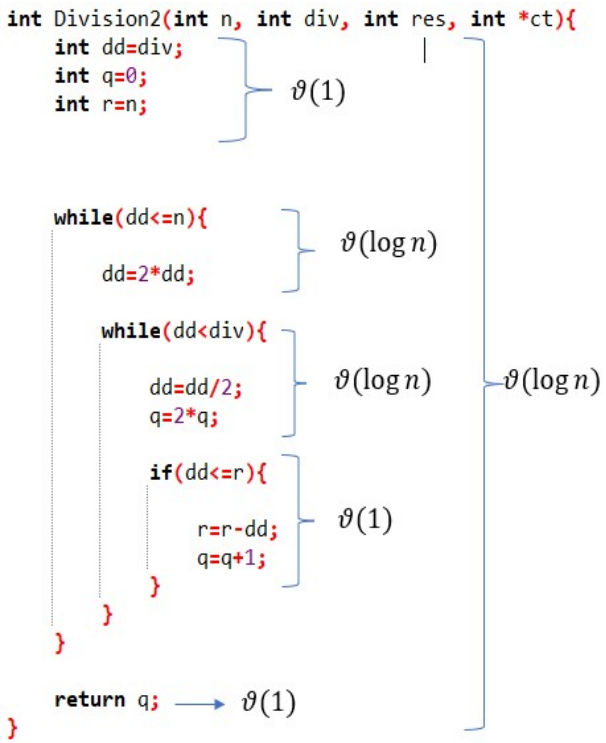


Figura 3. Análisis a posteriori de Division1

De igual forma para calcular el crecimiento de los dos ciclos while mostrado en la **Figura 3** realizamos el mismo procedimiento anterior.

n	dd
40	1
40	2
40	4
40	8
40	16
40	32
40	...

Si k es el numero de iteraciones.

$$dd = 1 + 2 + 4 + 8 + 16 + 32 + ... = 2^k$$

Entonces el ciclo while termina cuando.

$$2^k > n$$
$$2^k = n \implies k = \log_2(n)$$
$$\therefore k \in \mathcal{O}(\log(n))$$

El mismo procedimiento se realiza para el segundo while que tiene $T(n/2)$. Ahora una vez realizado el análisis a priori, para realizar la gráfica igualmente se realizo para los primeros 100 números, tanto para el mejor caso como para el peor caso.

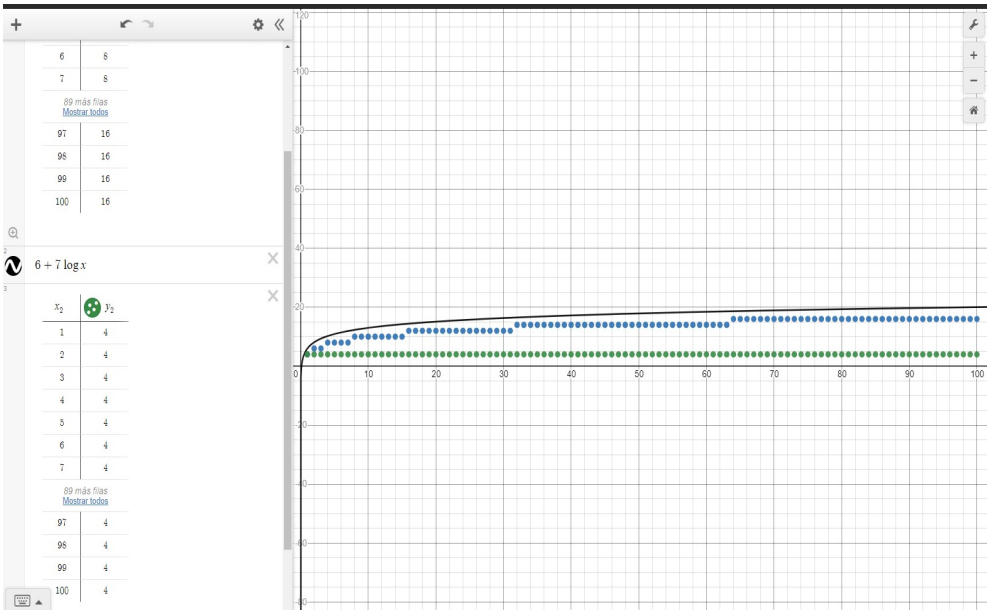


Figura 4. Análisis a posteriori de Division2

Como se observa en la **Figura 4** efectivamente el peor caso (color azul) tiene un comportamiento logarítmico y para el mejor caso sigue siendo igualmente contante. En esta ocasion para mostras mejor el crecimiento asintótico colocamos como cota superior a la funcion $6 + 7\log(x)$ para observar mejor el comportamiento.

3.1.3 Tercer Algoritmo (Division3)

Este algoritmo es el único recursivo de los tres, por lo que para realizar el analisis a priori utilizamos el método de sustitución hacia atrás como se muestra a continuación.

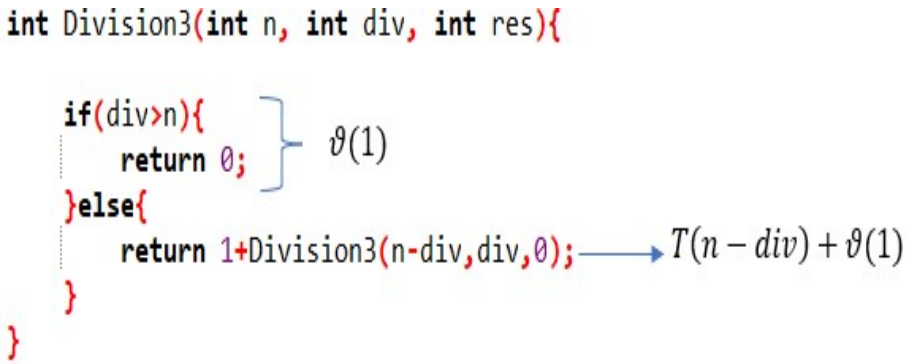


Figura 5. Análisis a Priori de Division3.

$$
\begin{aligned}
T(n) &= T(n - div) + \mathcal{O}(1) \\
&= [T(n - 2div) + c] + c \\
&= T(n - 2div) + 2c \\
&= T(n - 2div) + 2c \\
&= T(n - 3div) + 3c \\
&\vdots \\
(i) &= T(n - idiv) + ic \\
&\vdots \\
n - idiv = 0 &\Longrightarrow i = n/div \\
&= T(0) + n/divC \\
&\therefore T(n) \in \mathcal{O}(n)
\end{aligned}
$$

De esta forma el análisis a priori quedaría como se muestra en la siguiente figura.

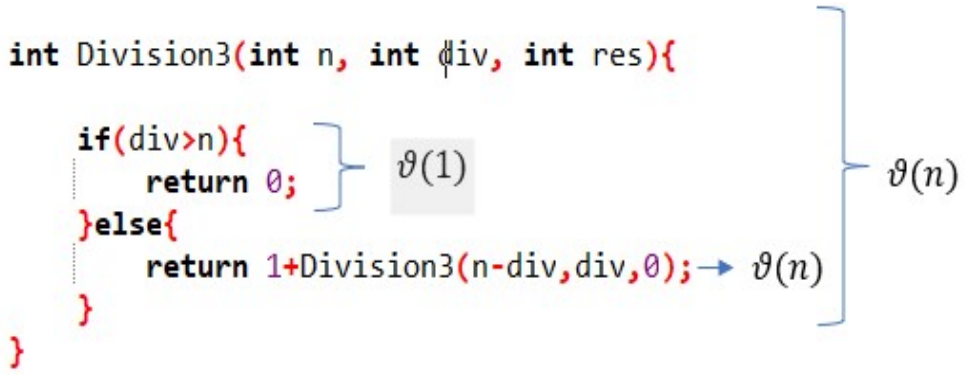


Figura 6. Análisis a Priori de Division3.

Ahora para el análisis a posteriori se realizo el mismo procedimiento que en los algoritmos anteriores, se utilizo para los 100 primeros números con un divisor de 1 para el peor caso y para el mejor caso tanto divisor como el dividendo eran el mismo.

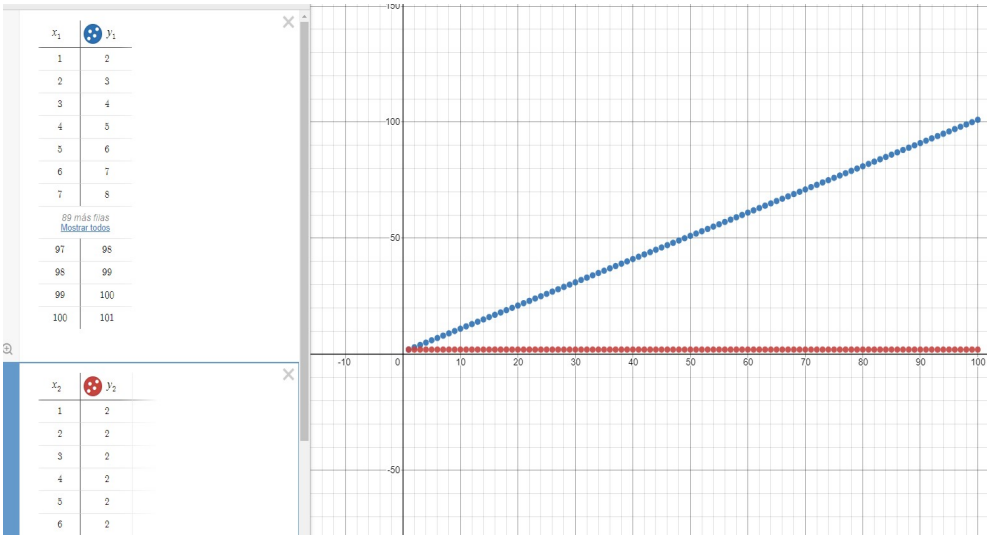


Figura 7. Análisis a posteriori de Division3.

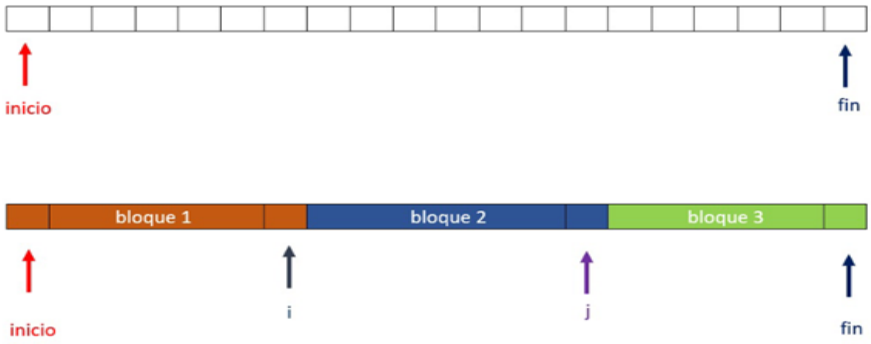
En la **Figura 7** se observa claramente el comportamiento del mejor caso (color azul) y mejor caso (color rojo) de maneja que el análisis a priori concuerda con la gráfica, y realmente observamos que tienen el mismo orden de complejidad que el primer algoritmo.

3.2 Segunda Parte

Del segundo programa se debió de realizar un buscador de un numero dentro de un arreglo de tamaño n, siendo que este arreglo fuera dividido en tres subarreglos para poder buscar dentro de estos si el numero se encontraba en el.

Figura 8. Subdivisión de arreglo con índices

Como se muestra en la **Figura 8** se debió hacer uso de un arreglo como se dijo, además de cuatro variables, o tres dependiendo de la cantidad de memoria que se desee ahorrar.



Para la implementación del arreglo se debió de timar en cuanta la plantación del programa, donde dentro de los números **Naturales** no existe ningún numero para que dividido por tres de como resultado otro numero **Natural**, por ello no podía dividirse el arreglo de manera libre, y al mismo tiempo que cada bloque de este arreglo quedara dividido de la misma manera, por lo cual se opto por el uso de un arreglo de múltiplos de 9 para tener una división exacta y poder dividir el arreglo en tres bloques iguales. Implementación de la división del arreglo **Figura 9** y **Figura 10**.

```
        tam+=9; /*Valor del aunmento para el arreglo y
        alamcenar sus respectivos valores, notando que el aumento se va dando de 9 en 9*/
    }
    fclose(onion);
}

/*Esta es una funcion que genera numeros aleatorios para el arreglo*/

void generaArreglo(int *A, int n){
    srand(time(NULL));
    /*Generacion de valores para el arreglo donde el valor
    total para los valores ingresados sismpre va aumentar dependiendo de n que recibe la variable tam*/
    int i;
    for(i=0; i<=n; i++)
        A[i]=i+5; /*Uso de valores que aumentaran cada vez de 5 eb 5, igual se pude usar una funcion de random.*/
}
```

Figura 9. Generación del Arreglo

```
int i = 0, j = 0, ini = 0, fin = n;
bool bus = false;

i = n/3; /*Division para el primer indice donde vamos a
ubicarlo a un punto ini + (ini+fin)/3*/
j = n-i; /*Ubicacion para el indice j donde su ubicacion
se da por el valor de i + i o el valor de i - fin por ser un valor
que esta mas cerca del final que otro*/

printf("%d %d\n",i,j);
```

Figura 10. División del arreglo

Como se muestra en la figuras anteriores esa seria la manera en la que se da la división del arreglo, se podría tomar una clase de caso diferente, ya que dependiendo del numero total de espacios que tenga el arreglo sea un numero para numero impar o solo múltiplos de 3 puede que sume o reste una casilla o solo se quede iguales como es este caso, por interpretación y facilidad se dejo solo como un caso donde son múltiplos de 3.

Ahora para las implementaciones se tiene lo siguiente. Donde el arreglo debía de ser recorrido de manera iterativa a lo largo de un espacio definido, esto se logra de forma de simulación recursiva, a que me refiero con esto, este algoritmo es mas sencillo interpretarlo de manera recursiva siendo que el sistema empieza en un numero n y se desea buscar otro número, pero al ser que encontrara el numero nos ostros debíamos de tener en cuanta en que momento debía detenerse el sistema para ellos se hizo uso de una variable del tipo **Booleana** donde el sistema al encontrar el numero simplemente cambia de estado hace la validación de la iteración una vez mas y al notar que no se cumple se sale de la iteración. Implementación Figura 11.

Figura 11. Implementación Iterativa del Buscador

```
while(bus == false){ //Inicio de la iteracion donde se detrenda cuando la condicion deje de ser falsa

    //Inicia de las comparaciones donde el sistema pide comprara si el numero esta en la pocision dada por los indices
    if((A[ini] == num)){

        printf("Numero %d en pocision %d", num, ini+1);
        bus = true; /*De encontrara el numero en el arreglo se cambia el estado de la variable booleana para detener el
        ciclo, una vez que se detiene se regresan los valores del contador para el analisis a porteriori */

    }
    if((A[fin] == num)){

        printf("Numero %d en pocision %d", num, fin+1);
        bus = true;

    }
    if(A[j] == num){

        printf("Numero %d en pocision %d", num, j+1);
        bus = true;

    }
    if(A[i] == num){

        printf("Numero %d en pocision %d", num, i+1);
        bus=true;

    }
    else {
        /*De ser que el numero no se encuientre en el arreglo se resta un 1 a los indices para hacer la siguienet
        busqueda dentro del arreglo y seguir con el sistema*/
        i--;
        j--;
        fin--;

        if(i==ini){ /*En caso de eestar en una estado en el que el indice ya regreso a la posicion 0 se termina el ciclo ya
        que los indices estan en la pocision sucesiva menor al origen de su indice menor siendo que el numero no due encontrado*/
            bus = true;
            printf("Numero no encontrado %d", num);
        }
    }
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
}

return cont; //Retorno del contador para el analisis a priori
```

Ahora para lo que es la implementación recursiva en el caso del buscador se debía de realizar lo siguiente que es el la misma definición para las comparaciones, siendo todo eso los mismo, pero solo que en ese caso es mas sencillo llevarlo al cabo ya que tenemos que al momento de detener el sistema este puede ser a través de un return, y en el mismo podremos realizar las operaciones necesarias para el cambio de los índices. En una vista general parece lo mismo pero se comporta de una forma totalmente diferente a ala que se nota para el caso iterativo. Implementación Figura 12.

```
if((A[0] == num)){ /*Inicio de la compraciones para encontrar el numero dentro del arreglo denotar
que este caso tenemos que la primera compracion no ponemos unvalor inicial, solo vrificamos si el numero
que deseamos no esta en el inidice 0 y que la iteracion ya no se encuentra*/

        printf("Numero %d en pocision 1", num);
        return cont; /*En el caso dado en que encontremos el numero en alguno de los indices se hace un
        return a la funcion y detenemos el sistema ya con el valor correspondiente encontrado*/
    }
    if((A[fin] == num)){

        printf("Numero %d en pocision %d", num, fin+1);
        return cont;

    }
    if(A[j] == num){

        printf("Numero %d en pocision %d", num, j+1);
        return cont;

    }
    if(A[i] == num){

        printf("Numero %d en pocision %d", num, i+1);
        return cont;

    }
    else {

        if((i-1)==0){/*Este es el caso de que no encontraramos el numero solo devolvemos lo que esta en
        nustro contador y salimos de la funcion*/
            printf("Numero no encontrado %d", num);
            return cont;
        }

        return busquedaR(A, i-1, j-1, fin-1, cont, num);/*Despues de la verificacion de que el arreglo no se ha completado
        tenemos la sentencia que llama de nuevo a la funcion y le mandamos los indices correspoendiente menos uno para
        seguir con la busqueda junto con el contadory el numero que seguimos buscando*/
    }

return cont;
```

Figura 12. Implementación Recursiva del Buscador

En el análisis a priori se tuvo lo siguiente:

Algorithm 3 Algoritmo Iterativo Búsqueda

```
function ITERATIVO(a ← A, n ← n)
    while bus = 0 do ———————— O(log n) y Ω(c)
        if n = a[0] then ———————————— O(1)
            bus ← 1 ———————— O(1)
            return a[0] + 1 ———————————— O(1)
        end if
        if a[fin] = n then ———————————— O(1)
            bus ← 1 ———————— O(1)
            return fin + 1 ———————————— O(1)
        end if
        if a[i] = n then ———————————— O(1)
            bus ← 1 ———————— O(1)
            return i + 1 ———————————— O(1)
        end if
        if a[j] = n then ———————————— O(1)
            bus ← 1 ———————— O(1)
            return j + 1
        else ———————————— O(1)
            i ← i − 1 ———————— O(1)
            j ← j − 1 ———————— O(1)
            fin ← fin − 1 ———————— O(1)
            if a[i] = 0 then ———————————— O(1)
                bus ← 1 ———————————— O(1)
                return −1 ———————————— O(1)
            end if
        end if
    end while
end function
```

Algorithm 4 Algoritmo Recursivo Búsqueda

```
function RECURSIVA(a ← A, n ← n, i, j, fin)
    if n = a[0] then ———————————— O(1)
        return a[0] + 1————————————O(1)
    end if
    if a[fin] = n then ———————————— O(1)
        return fin + 1————————————O(1)
    end if
    if a[i] = n then ———————————— O(1)
        return i + 1————————————O(1)
    end if
    if a[j] = n then ———————————— O(1)
        return j + 1
    else————————————O(1)
        if a[i] = 0 then————————————O(1)
            return −1————————————O(1)
        else————————————O(1)
            return Recursiva(a, n, i − 1, j − 1, fin − 1)——T(n) = T(n/3) +
O(5)
        end if
    end if
end function
```

Para solucionar el orden da la función se tiene:

$$T(n) = T(n-1) + c$$
$$n = 3^K || (k = \log_3 n)$$
$$T(3^k) = T(3^{k-1}) + c$$

Aplicando sustitución hacia atrás

$$= [T(3^{k-2}) + c] + c$$
$$= T(3^{k-2}) + 2c$$
$$= T(3^{k-3}) + 3c$$
$$.$$
$$.$$
$$.$$
$$= T(3^{k-i}) + i * c$$
$$= T(1) + (k) * c$$

Regresando a la variable original

$$= \log_3 n * c$$
$$T(n) = \Theta(\log_3 n)$$

Para el análisis a posterior se tiene lo siguiente:

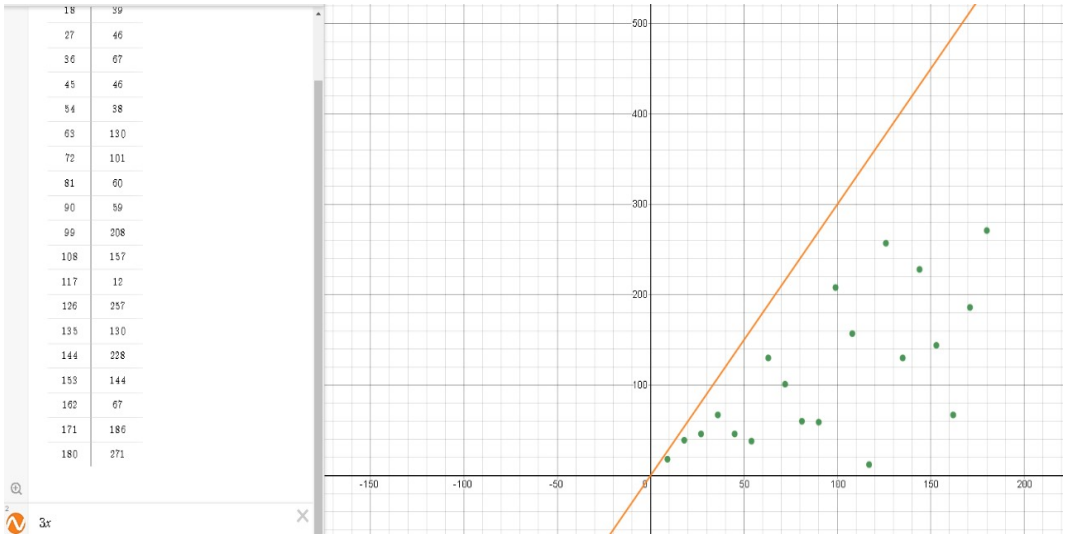


Figura 13. Análisis a posterior Búsqueda Recursiva

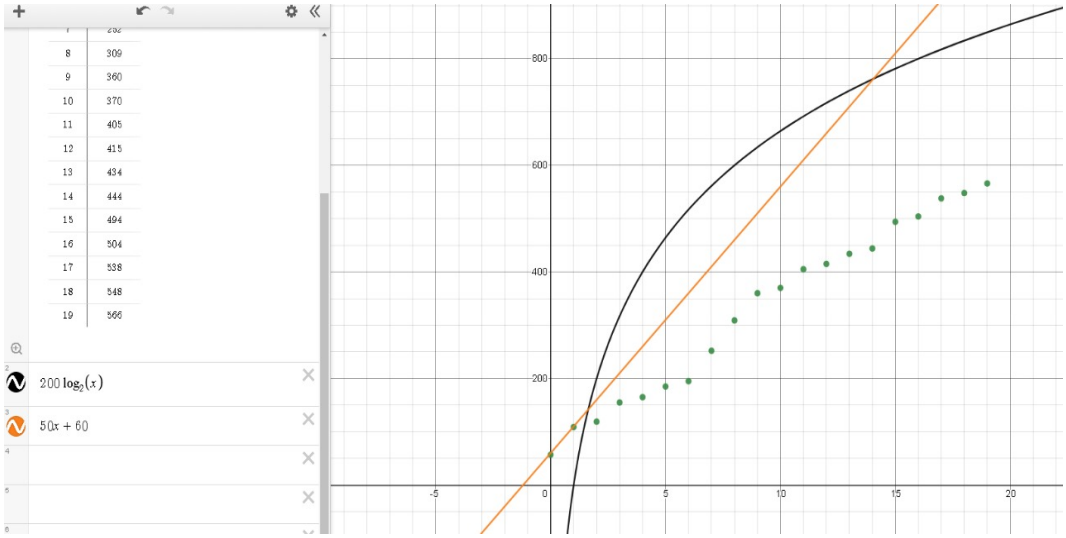


Figura 14. Análisis a posterior Búsqueda Iterativa

3.3 Características PC's

Características PC Venegas Garcia Andres
-Procesador Intel i5-10400f a 2.9 Ghz six core
-16 Gb de Ram
-Nvidia Geforce GTX 970

Características PC Vaquera Aguilera Ethan Emiliano
-Procesador ryzen 7-2700 a 3.2 Ghz octa core
-16 Gb de Ram
-AMD Radeon RX 5600XT

4 Conclusiones

Conclusión General

Para finalizar, en la practica se observo claramente que un algoritmo recursivo no es siempre mas complejo que un iterativo, por los visto en ambas partes de la practica, el algoritmo recursivo puede tener un mejor o peor orden de complejidad que un algoritmo iterativo, las técnica que se usen para diseñar un algoritmo depende del problema que uno quiera resolver, además el orden de complejidad puede variar dependiendo las linear de código y ciclos que uno utilice.

Conclusión Individual Vanegas García Andrés

De forma particular, tuve un poco de confusión al momento de calcular el orden de complejidad para cada algoritmo, ya que en clase siempre utilizamos como variable n y en este caso aparece otra letra, sin embargo la mecánica fue la misma. En cuestión de objetivo de la practica, a mi pareces se observo claramente que un algoritmo recursivo no es mejor que uno iterativo o viceversa, además de que hay distintas maneras y algoritmos de resolver un problema y cada uno tiene un orden de complejidad y comportamiento distinto sin importas las técnicas empleadas. Esta practica me sirvo para practicar lo aprendido en clase y observar como se comportan diferentes algoritmos que resuelven un mismo problema

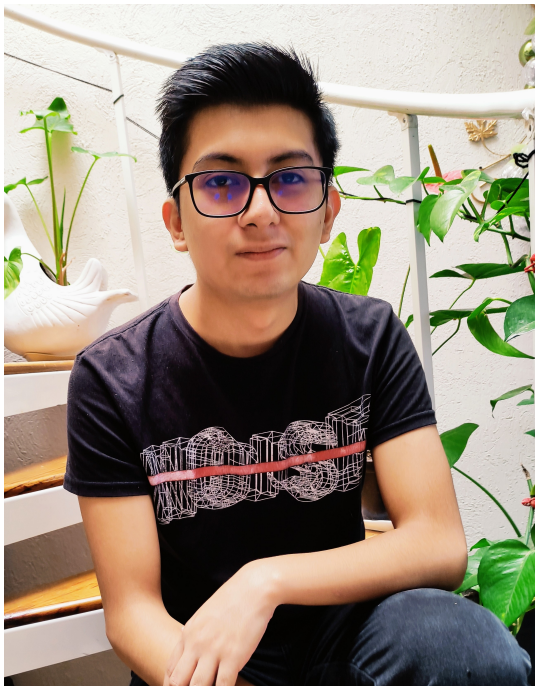


Figura 15. Vanegas García Andrés

Conclusión Individual Vaquera Aguilera Ethan Emiliano

Por mi lado la practica fue sencilla del lado de la abstracción del problema , aunque cabe la posibilidad que haya mal interpretado el problema a mi parecer llegue a una solución muy eficiente en términos de tiempo y memoria del

algoritmo, por otro lado el sistema que se planteo es de un modo muy sencillo de comprender y de usar, pero lo difícil de la practica fue el análisis a priori pero a pesar de eso se realizo de manera cautelosa y ordenada los cálculos puestos en el documento y se analizó detenidamente que estuvieran correctos. Por mi lado estuvo bien creo que el algoritmo es muy interesante y me gustaría poder programar mas algoritmos de ese tipo y aprender mas ejemplos de como hacer el análisis a priori para poder aplicarlos en el futuro cercano.

Figura 16. Vaquera Aguilera Ethan Emiliano

5 Bibliografía

programmerclick.com. (2021). Complejidad temporal y complejidad espacial. 2021, de programmerclick.com Sitio web: https://programmerclick.com/article/89671544092/

Agrawal, Manindra; Kayal, Neeraj; Saxena, Nitin: "PRIMES is in P". Annals of Mathematics 160 (2004), no. 2, pp. 781–793.

J.M.+Gimeno+y+J.L.+González. (2021). Recursividad. 2021, de web Sitio web: http://ocw.udl.cat/enginyeria-i-arquitectura/programacio-2/continguts-1/2-recursividad.pdf