Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos

Semestre: 2022-1

Grupo: 3CV11

Práctica 3

Fecha: 8 de octubre de 2021

Práctica 3: Funciones Recursivas vs iterativas.

Brayan Ramirez Benitez.
*bramirezb*1901@*alumno.ipn.mx*

Resumen: En esta práctica realizamos el análisis a priori y posteriori para tres algortimos que calculan el cociente de dos enteros positivos con el objetivo de encontrar el más eficiente. Por otro lado, implementaremos un algoritmo para encontrar un elemento en un arreglo de tamaño n, dividiendo el arreglo en tres bloques de manera iterativa y recursiva realizando el análisis a priori y posteriori en ambos algoritmos. Para esto, los algoritmos se implementarán mediante el uso del lenguaje de programación C, además llevamos a cabo el análisis a posteriori mediante el uso de una Calculadora Grafica del sitio Web Desmos.

Palabras Clave: Análisis Posteriori, Análisis Priori, Lenguaje C, Cociente, Busqueda, Algoritmos Recursivos, Algortimos Iterativos.

1 Introducción

Cuando necesitamos resolver un problema podemos utilizar una técnica que consiste en dividir este problema en algunos subproblemas de más fácil resolución, esto lo conocemos como divide y vencerás. De esta manera, cuando aplicamos esta técnica a un algoritmo, este recibe el nombre de algoritmo recursivo, donde el algoritmo se llama a si mismo en repetidas ocasiones, con la intención de simplificar el problema durante cada llamada, hasta conseguir una solución trivial o directa.

Tanto en matemáticas como en computación la **recursividad** es un concepto sumamente importante puesto que es una manera diferente para desarrollar estructuras de repetición, es decir ciclos. Una gran parte de los lenguajes de programación permiten implementar los **algoritmos recursivos**. Cuando implementamos un **algoritmo recursivo**, debemos definir cuando termina de llamarse a sí mismo, esto para impedir que lo haga indefinidamente, por lo tanto, definimos una condición llamada caso base. El caso base es una solución sencilla para un caso en particular, un **algoritmo recursivo** puede tener más de una solución particular y todos poseen al menos un caso base para garantizar que la **recursividad** no es infinita. Un caso recursivo es una solución que implica utilizar una llamada así mismo con parámetros que se acercan a un caso base.

Los algoritmos que tienen la característica de implementarse haciendo uso de ciclos reciben el nombre de **algoritmos iterativos**, los cuales nos permiten llevar a cabo tareas repetitivas. De igual manera que los **algoritmos recursivos**, una gran parte de los lenguajes de programación nos permiten implementar los **algoritmos iterativos** para los cuales existen palabras reservadas con el objetivo de realizar **iteraciones**. En general tenemos tres maneras para implementar un ciclo, esto mediante un **Do while, While** y **For** los cuales poseen una serie de instrucciones que se repiten siempre y cuando se cumpla una condición y en caso de no cumplirse esta condición el ciclo termina y continua con las demás instrucciones.

Una gran parte de los algoritmos pueden implementarse de manera recursiva e iterativa, pero si comparamos la **recursión** con la **iteración**, tenemos que una **iteración** tiene un clico explicito y este concluye cuando la condición del ciclo ya no se cumple, mientras que en la **recursión** en repetidas ocasiones se invoca a si mismo y este concluye cuando estas llamadas nos llevan a un caso base, por tanto necesitamos determinar que versión es mejor utilizar.

Esta práctica tiene como objetivo analizar la complejidad temporal para dos problemas que tienen varios algoritmos propuestos, tres de ellos consisten en calcular el cociente de dos enteros positivos implementados de manera recursiva e iterativa para comparar los resultados obtenidos, por otro lado,

dos algoritmos que determinan si un elemento se encuentra en un arreglo de tamaño **n**, dividiendo el arreglo en 3 bloques, un algortimo será implementado de manera recursiva y el otro de manera iterativa. Para cada algoritmo realizaremos varios experimentos con iteraciones y un análisis comparativo.

2 Conceptos Básicos

En esta sección mostraremos todos los conceptos necesarios para el desarrollo de la práctica.

De acuerdo a Cormen (2009) podemos definir un **algoritmo** como una secuencia de pasos computacionales que transforman una entrada a una salida, además podemos observarlo como una herramienta que resuelve un problema de cálculo bien definido.

Complejidad temporal o eficiencia en tiempo: Es el número de operaciones para resolver un problema con una entrada de tamaño **n**. Se denota por $\mathbf{T(n)}$.

Definición: Dada una función $\mathbf{g(n)}$. $\Theta(\mathbf{g(n)})$ denota el conjunto de funciones definidas como:
$\Theta(\mathbf{g(n)}) = \{\ \mathbf{f(n)}$: $\exists$n C_1, $C_2 > 0$ y $n_0 > 0$ Tal que $0 \leq C_1\ \mathbf{g(n)} \leq \mathbf{f(n)} \leq C_2\ \mathbf{g(n)}\ \forall$ n $\geq n_0\ \}$
Decimos que es un ajuste asintótico para $\mathbf{f(n)}$.

Definición: Dada una función $\mathbf{g(n)}$. $O(\mathbf{g(n)})$ denota el conjunto de funciones definidas como:
$O(\mathbf{g(n)}) = \{\ \mathbf{f(n)}$: $\exists$n $C > 0$ y $n_0 > 0$ constantes Tal que $0 \leq \mathbf{f(n)} \leq C\ \mathbf{g(n)}\ \forall$ n $\geq n_0\ \}$

Definición: Dada una función $\mathbf{g(n)}$. $\Omega(\mathbf{g(n)})$ denota el conjunto de funciones definidas como:
$\Omega(\mathbf{g(n)}) = \{\ \mathbf{f(n)}$: $\exists$n $C > 0$ y $n_0 > 0$ constantes Tal que $0 \leq C\ \mathbf{g(n)} \leq \mathbf{f(n)}\ \forall$ n $\geq n_0\ \}$

Recursividad: De acuerdo con Rodríguez, A (2016) la recursividad en programación se refiera a la técnica para resolver un problema mediante la sustitución por otros problemas más simples, pero de la misma categoría, entonces podemos decir que un algoritmo es recursivo si contiene de manera indirecta o directa una llamada a el mismo.

Iteracíon: Martins, J (2021) menciona que una iteración en matemáticas y computación consiste en una estructura de control que pertenece a un algoritmo y resuelve un problema determinado, mediante la ejecución repetida de un conjunto de instrucciones, hasta la ocurrencia de algunas condiciones lógicas.

Primer Algoritmo: El algoritmo consiste en devolver el cociente de dos enteros positivos mediante una función implementada de manera iterativa. Primero la función recibe tres números enteros positivos **n, div** y **res**, luego declaramos una variable **q** donde almacenaremos el valor del cociente, iniciamos un ciclo **while** que tendrá como condición **n** mayor o igual que **div**, posteriormente a **n** le restamos **div** y este resultado se lo asignamos a **n**, luego incrementamos **q** en uno, una vez que termine el ciclo **while** asignamos a **res** el valor de **n** y devolvemos **q**.

Algorithm 1 *Division1*

Require: Int n, Int div, Int res
Ensure: Cociente $\frac{n}{div}$

```
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
```

Segundo Algoritmo: El algoritmo consiste en devolver el cociente de dos enteros positivos mediante una función implementada de manera iterativa diferente al **primer algoritmo**. La función recibe tres números enteros positivos **n, div** y **res**, declaramos tres variables **dd, q** y **r**, a la variable **dd** le asginamos el valor de **div** y a la variable **r** le asignamos el valor de **n**, luego iniciamos un ciclo **while** con la condición **dd** menor igual a **n**, dentro del **while** incrementamos **dd** de dos en dos, posteriormente iniciamos otro ciclo **while** con la condición **dd** mayor que **div** y dentro del ciclo decrementamos a **dd** a la mitad, a **q** la incrementamos de dos en dos y colocamos una condición mediante un **if** que se ejecutara si **dd** menor o igual a **r**, dentro del **if** a **r** le restamos **dd** e incrementamos **q** en uno, por ultimo una vez que termine el ciclo **while** regresamos el valor de **q**.

Algorithm 2 *Division2*

Require: Int n, Int div, Int res
Ensure: Cociente $\frac{n}{div}$
1: $dd = div$
2: $q = 0$
3: $r = n$
4: **while** $dd \leq n$ **do**
5: $\quad dd = 2 \cdot dd$
6: **end while**
7: **while** $dd \succ div$ **do**
8: $\quad dd = \frac{dd}{2}$
9: $\quad q = 2 \cdot q$
10: $\quad$ **if** $dd <= r$ **then**
11: $\quad\quad r = r - dd$
12: $\quad\quad q = q + 1$
13: $\quad$ **end if**
14: **end while**
15: **return** q

Tercer Algoritmo: El algoritmo consiste en devolver el cociente de dos enteros positivos mediante una función implementada de manera recursiva. Primero la función recibe dos números enteros positivos **n** y **div**, colocamos un **if** con la condición **div** mayor que **n**, en caso de cumplirse regresa un cero, en caso de no cumplirse entra en el **else** y llama recursivamente a la función.

Algorithm 3 *Division3*

Require: Int n, Int div
Ensure: Cociente $\frac{n}{div}$
1: **if** $div \succ n$ **then**
2: $\quad$ return 0
3: **else**
4: $\quad$ return $1 + Division3(n - div, div)$
5: **end if**

Cuarto Algoritmo: El algoritmo consiste en devolver el índice de la posición del arreglo donde se encuentra el elemento que buscamos, implementado de manera iterativa. La función recibe un arreglo ordenado, el tamaño del arreglo **(tam)** y el elemento que buscamos **(x)**, comenzamos declarando cuatro variables **i, j, inicio** y **fin**. A la variable **inicio** le asignaremos el índice del primer elemento del arreglo mientras que a **fin** le asignamos el índice del último elemento del arreglo. Luego iniciamos un ciclo **while** que se ejecutara si el índice **inicio** es menor o igual que **fin**, si esto se cumple dividimos el arreglo en tres bloques e inicializamos a la variable **i** con el índice de la posición donde termina el primer bloque y **j** con el índice de la posición donde termina el segundo bloque. Posteriormente preguntamos mediante un **if** si el elemento **x** es igual al elemento del índice **i**, si esto se cumple devolvemos **i**, en caso de no cumplirse preguntamos si el elemento **(x)** es igual al elemento del índice **j**, si esto se cumple devolvemos **j**, si esto no se cumple preguntamos si el elemento **(x)** es menor al elemento del índice **i**, si esto se cumple asignamos al índice **fin** el valor de **i** menos uno, si esto no se cumple preguntamos si el elemento **(x)** es mayor al elemento del índice **i** pero menor al elemento del índice **j**, si esto se cumple asignamos al índice **inicio** el valor de **i** mas 1 y al índice **fin** el valor de **j** menos uno, si esto no se cumple preguntamos si el elemento **(x)** es mayor al elemento del índice **j**, si esto se cumple asignamos al índice inicio el valor de **j** mas 1, por último, una vez que termine el ciclo **while** devolvemos el valor de menos uno, es decir que el elemento **(x)** no se encuentra en el arreglo.

Algorithm 4 *Busqueda Iterativa*

Require: Int A[], Int tam, Int x
Ensure: Índice de la posición donde se encuentra x

```
1: i = 0
2: j = 0
3: inicio = 0
4: fin = tam − 1
5: while inicio ≤ fin do
6:     i = inicio + (fin−inicio)/3
7:     j = fin − (fin−inicio)/3
8:     if A[i] == x then
9:         return i
10:    end if
11:    if A[j] == x then
12:        return j
13:    else if x ≺ A[i] then
14:        fin = i − 1
15:    else if x ≻ A[i] & x ≺ A[j] then
16:        fin = j − 1
17:        inicio = i + 1
18:    else if x ≻ A[j] then
19:        inicio = j + 1
20:    end if
21: end while
22: return -1
```

Quinto Algoritmo: El algoritmo consiste en devolver el índice de la posición del arreglo donde se encuentra el elemento que buscamos implementado de manera recursiva. La función recibe un arreglo ordenado, el **inicio** del arreglo, el **fin** del arreglo y el elemento que buscamos **(x)**, comenzamos declarando dos variables **j** e **i**. Dividimos el arreglo en tres bloques e inicializamos a la variable **i** con el índice de la posición donde termina el primer bloque y **j** con el índice de la posición donde termina el segundo bloque. Luego preguntamos mediante un **if** si el índice **inicio** es mayor al índice **fin**, si esto se cumple regresamos un menos uno puesto que el elemento no se encuentra en el arreglo, en caso de no cumplirse mediante un **if** preguntamos si el elemento **(x)** es igual al elemento del índice **i**, si esto se cumple devolvemos **i**, en caso de no cumplirse preguntamos si el elemento **(x)** es igual al elemento del índice **j**, si esto se cumple devolvemos **j**, si esto no se cumple preguntamos si el elemento **(x)** es menor al elemento del índice **i**, si esto se cumple llamamos recursivamente a la función con el mismo arreglo, el índice **inicio**, el índice **fin** igual a **i** menos 1 y el elemento que buscamos, si esto no se cumple preguntamos si

el elemento **(x)** es mayor al elemento del índice **i** pero menor al elemento del índice **j**, si esto se cumple llamamos recursivamente a la función con el mismo arreglo, el índice **inicio** igual a **i** mas uno, el índice **fin** igual a **j** menos 1 y el elemento que buscamos, por último, si esto no se cumple preguntamos si el elemento **(x)** es mayor al elemento del índice **j**, si esto se cumple llamamos recursivamente a la función con el mismo arreglo, el índice inicio igual a **j** mas uno, el índice **fin** y el elemento que buscamos.

Algorithm 5 *Busqueda Recursiva*

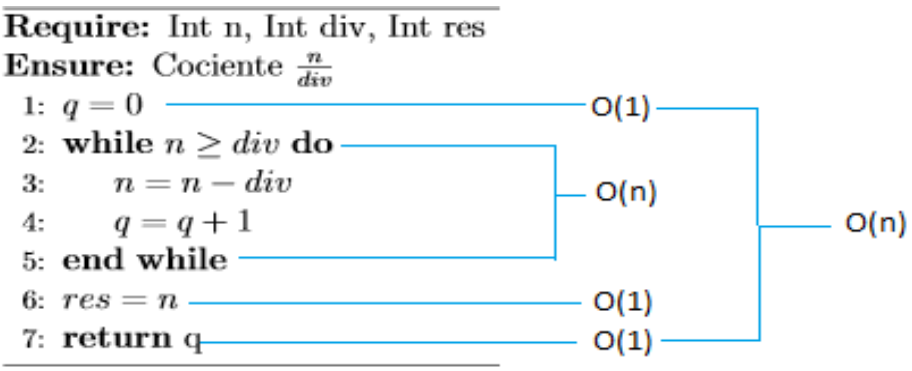
Require: Int A[], Int inicio, Int fin, Int x
Ensure: Índice de la posición donde se encuentra x

1: $i = inicio + \frac{fin-inicio}{3}$
2: $j = fin - \frac{fin-inicio}{3}$
3: **if** $inicio \succ fin$ **then**
4: return -1
5: **end if**
6: **if** $A[i] == x$ **then**
7: return i
8: **end if**
9: **if** $A[j] == x$ **then**
10: return j
11: **else if** $x \prec A[i]$ **then**
12: return BR(A, inicio, $i - 1$, x)
13: **else if** $x \succ A[i]$ & $x \prec A[j]$ **then**
14: return BR(A, $i + 1$, $j - 1$, x)
15: **else if** $x \succ A[j]$ **then**
16: return BR(A, $j + 1$, fin, x)
17: **end if**

3 Experimentación y Resultados

Análisis a priori para el primer algoritmo: Para el algoritmo **division1** llevaremos a cabo el análisis mediante segmentos de código. Si observamos el pseudo-código podemos determinar que el peor caso ocurre cuando **n** es mayor que **1** y **div** es **1**.

Se tiene que



De lo anterior concluimos que

La función **division1** $\in O(n)$

Análisis a priori para el segundo algoritmo: Para el algoritmo **division2** llevaremos a cabo el análisis mediante segmentos de código. Si observamos el pseudo-código podemos determinar que el peor caso ocurre cuando **n** es mayor que **1** y **div** es **1**.

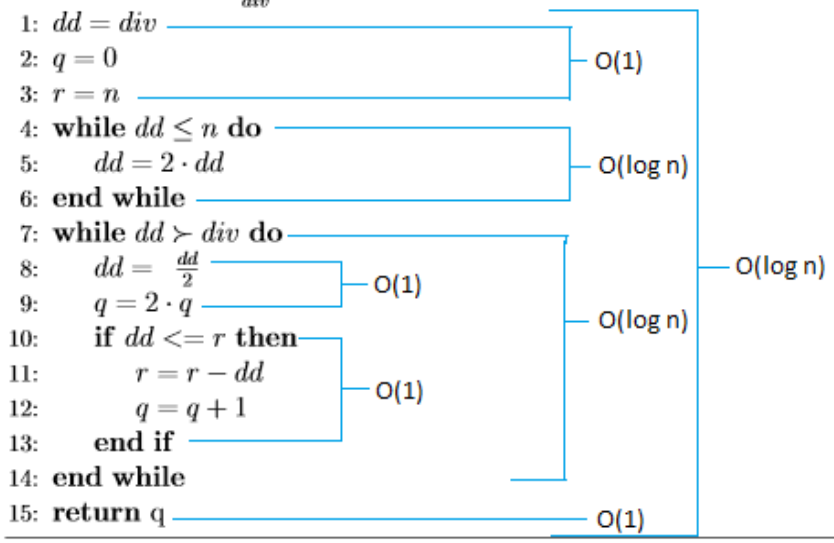
Se tiene que

Algorithm 2 *Division2*

Require: Int n, Int div, Int res

Ensure: Cociente $\frac{n}{div}$

```
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
```



De lo anterior concluimos que

La función **division2** $\in O(\log n)$

Análisis a priori para el tercer algoritmo: Para el algoritmo **division3** llevaremos a cabo el análisis mediante segmentos de código. Si observamos el pseudo-código podemos determinar que el peor caso ocurre de acuerdo a la ecuación de recurrencia **T(n)**.

Tenemos que

Algorithm 3 *Division3*

Require: Int n, Int div

Ensure: Cociente $\frac{n}{div}$

1: **if** $div \succ n$ **then**

2: return 0

3: **else**

4: return $1 + Division3(n - div, div)$

5: **end if**

O(1)

T(n - 1) + O(1)

de lo anterior

$$T(n) = \begin{cases} O(1), & div > n_1 \\ T(n-1) + O(1), & div \leq n_1 \end{cases}$$

Sea $n = div - n_1$

$\Rightarrow$

$$T(n) = \begin{cases} c, & n > 0 \\ T(n-1) + c, & n \leq 0 \end{cases}$$

Resolviendo la ecuación de recurrencia mediante el método de sustitución haceia atrás, se tiene:

$T(n) = T(n-1) + c$

$= T(n-2) + 2c$

$= T(n-3) + 3c$

$\vdots$

$(i) = T(n-i) + ic$

$\vdots$

$= T(1) + nc$

$= c + nc$

$= c(1 + n)$

$\therefore T(n) \in O(n)$

$\therefore$ La función **division3** $\in O(n)$

Análisis a priori para el cuarto algoritmo: Para el algoritmo **BI** llevaremos a cabo el análisis mediante segmentos de código. Si observamos el pseudo-código podemos determinar que el mejor caso ocurre cuando **x** se encuentra en la posición de los índices **i o j** y el peor caso ocurre cuando **x** no se encuentra en el arreglo.

Tenemos que

Algorithm 4 *Busqueda Iterativa*

Require: Int A[], Int tam, Int x
Ensure: Índice de la posición donde se encuentra x

```
1:  i = 0                                                          O(1)
2:  j = 0                                                          O(1)
3:  inicio = 0                                                     O(1)
4:  fin = tam − 1                                                  O(1)
5:  while inicio ≤ fin do                                          O(log n)
6:      i = inicio + (fin−inicio)/3                                O(1)
7:      j = fin − (fin−inicio)/3                                   O(1)
8:      if A[i] == x then                                          O(1)
9:          return i                                               O(1)
10:     end if                                                     O(1)
11:     if A[j] == x then                                          O(1)
12:         return j                                               O(1)
13:     else if x ≺ A[i] then                                      O(1)
14:         fin = i − 1                                            O(1)
15:     else if x ≻ A[i] & x ≺ A[j] then                           O(1)
16:         fin = j − 1                                            O(1)
17:         inicio = i + 1                                         O(1)
18:     else if x ≻ A[j] then                                      O(1)
19:         inicio = j + 1                                         O(1)
20:     end if                                                     O(1)
21: end while                                                      O(log n)
22: return -1                                                      O(1)
```

de lo anterior

La función **BI** $\in O(\log n)$

Análisis a priori para el quinto algoritmo: Para el algoritmo **BR** llevaremos a cabo el análisis mediante segmentos de código. Si observamos el pseudo-código podemos determinar que el mejor caso ocurre cuando **x** se encuentra en la posición de los índices **i o j** y el peor caso ocurre de acuerdo a la ecuación de recurrencia **T(n)**.

Tenemos que

Algorithm 5 *Busqueda Recursiva*

Require: Int A[], Int inicio, Int fin, Int x
Ensure: Índice de la posición donde se encuentra x

```
1: i = inicio + (fin−inicio)/3
2: j = fin − (fin−inicio)/3                                   O(1)
3: if inicio ≻ fin then
4:     return -1
5: end if                                                     O(1)
6: if A[i] == x then
7:     return i
8: end if                                                     O(1)
9: if A[j] == x then
10:    return j                                               O(1)
11: else if x ≺ A[i] then
12:    return BR(A, inicio, i − 1, x)                         T(n/3)
13: else if x ≻ A[i] & x ≺ A[j] then
14:    return BR(A, i + 1, j − 1, x)                          T(n/3)
15: else if x ≻ A[j] then
16:    return BR(A, j + 1, fin, x)                            T(n/3)
17: end if
```

de lo anterior

$$T(n) = \begin{cases} O(1), & n = 0 \\ T(n/3) + O(1), & n > 0 \end{cases}$$

Sea $n = 3^k$ con $(k = \log_3 n)$, se tiene que

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

como $n = 3^k$

$T(3^k) = T(\frac{3^k}{3}) + O(1)$

$T(3^k) = T(3^{k-1}) + O(1)$

entonces

$$T(2^k) = \begin{cases} O(1), & k = 0 \\ T(3^{k-1}) + O(1), & k > 0 \end{cases}$$

$\Rightarrow$

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

Resolviendo la ecuación de recurrencia mediante el método de sustitución haceia atrás, se tiene:

$T(3^k) = T(3^{k-1}) + c$

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

$= T(3^{k-2}) + 2c$

$= T(3^{k-3}) + 3c$

$\vdots$

$(i) = T(3^{k-i}) + kc$

$\vdots$

$= T(3^0) + kc = T(1) + kc$

$= T(3^{k-i}) + kc$

$= c + kc$

$= c(1 + k)$

$= c(1 + \log_3 n)$

$\therefore T(n) \in O(\log_3 n)$

$\therefore$ La función $\mathbf{BR} \in O(\log_3 n)$

Análisis a posteriori para el primer algoritmo:

Comenzamos implementando el algoritmo como una función llamada **"division1"** con base en el pseudo-código del **Algorithm 1 Division1**, mediante el Lenguaje de Programación en C.

```c
int division1(int n, int div, int res){
    int q = 0;

    while(n >= div){
        n = n - div;
        q = q + 1;
    }
    res = n;

    return q;
}
```

Figura 1. Código para el **primer algoritmo** en lenguaje C.

Además, utilizaremos la función **Principal** la cual nos servirá para ejecutar el algoritmo y obtener los datos para el análisis, donde enviaremos los resultados a un archivo.

```c
void principal(){
    FILE *pf = fopen("Muestra.csv","at");

    int ct = 0, pts = 900, i;
    srand(time(NULL));

    for(i = 0;i<pts;i++){
        ct = 0;
        int n = rand()%200000;
        int q = division1(n, 1, 0, &ct);

        printf("\nel cociente de %d y %d es %d\n", n, div, q);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 2. Código para la función **Principal** del **primer algoritmo** en lenguaje C.

Para realizar el análisis la función **division1 (Figura 1)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número ejecuciones de cada línea.

```
int division1(int n, int div, int res, int *ct){
    int q = 0;(*ct)++;

    while(n >= div){
        (*ct)++;(*ct)++;
        n = n - div;(*ct)++;
        q = q + 1;(*ct)++;
    }(*ct)++;
    res = n;(*ct)++;
    return q;(*ct)++;
}
```

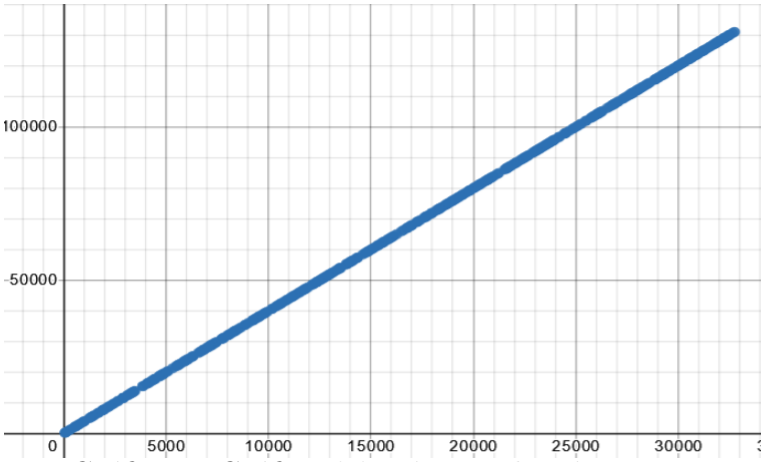
Figura 3. Código para el **primer algoritmo** con el parámetro **ct**.

Del análisis a priori sabemos qel peor caso ocurre cuando **div** es **1** y **n** es mayor que **1**. Entonces para el análisis a posteriori generamos valores aleatorios entre 2 y 200000 para **n** y le asignamos el valor de **1** a **div**, estos valores los enviaremos a la funcion **division1**. El resultado de esta ejecución arroja los siguientes valores.

n	Ct
21247	84992
28353	113416
31670	126684
23957	95832
24945	99784
20067	80272
3255	130224
18060	72244
18190	72764
5367	21472

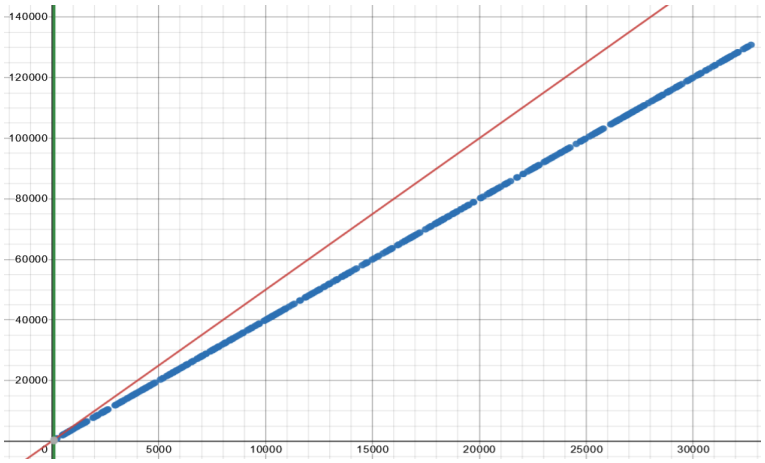
Tabla 1. Valores de la ejecución para el **primer algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es lineal $O(n)$.



Gráfica 1. Gráfica del **primer algoritmo**.

Ahora proponemos la función $\mathbf{g(n)} = 5(n) \in O(n)$ con $n_o = 100$ para la cota superior.



Gráfica 2. Gráfica del **primer algoritmo** con la función propuesta.

Análisis a posteriori para el segundo algoritmo:

Comenzamos implementando el algoritmo como una función llamada **"division2"** con base en el pseudo-código del **Algorithm 2 Division2**, mediante el Lenguaje de Programación en C.

```c
int division2(int n, int div, int res){
    int dd = div;
    int q = 0;
    int r = n;

    while(dd <= n){
        dd = 2*dd;
    }
    while(dd > div){
        dd = dd/2;
        q = 2*q;
        if(dd <= r){
            r = r - dd;
            q = q + 1;
        }
    }
    return q;
}
```

Figura 4. Código para el **segundo algoritmo** en lenguaje C.

Además, utilizaremos la función **Principal** para el **segundo algoritmo** la cual nos servirá para ejecutar y obtener los datos para el análisis, donde enviaremos los resultados a un archivo.

```c
void principal(){
    FILE *pf = fopen("Muestra.csv","at");
    int ct = 0, pts = 900, i;
    srand(time(NULL));

    for(i = 0;i<pts;i++){
        ct = 0;
        int n = rand()%200000;
        int q = division2(n, 1, 0, &ct);

        printf("\nel cociente de %d y %d es %d\n", n, div, q);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 5. Código para la función **Principal** del **segundo algoritmo** en lenguaje C.

Para realizar el análisis la función **division2 (Figura 4)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número ejecuciones de cada línea.

```
int division2(int n, int div, int res, int *ct){
    int dd = div;(*ct)++;
    int q = 0;(*ct)++;
    int r = n;(*ct)++;

    while(dd <= n){
        (*ct)++;
        (*ct)++;
        dd = 2*dd;
    }(*ct)++;
    while(dd > div){
        (*ct)++;
        dd = dd/2;(*ct)++;
        q = 2*q;(*ct)++;
        if(dd <= r){
            (*ct)++;
            r = r - dd;(*ct)++;
            q = q + 1;(*ct)++;
        }
        (*ct)++;
    }(*ct)++;
    (*ct)++;
    return q;
}
```

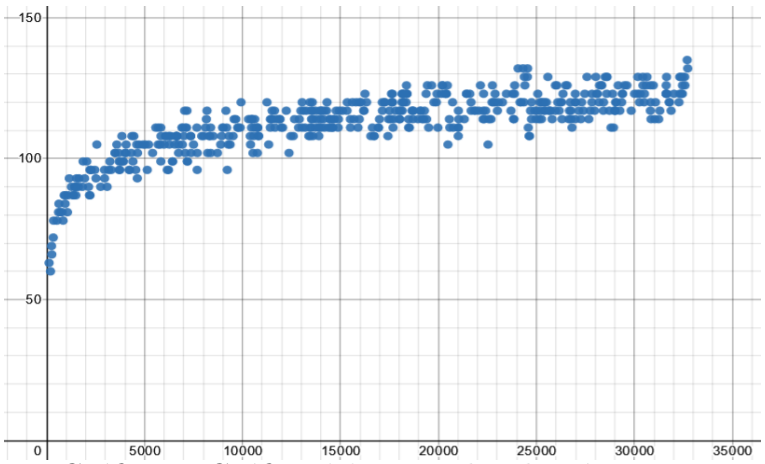
Figura 6. Código para el **segundo algoritmo** con el parámetro **ct**.

De manera similar para el análisis a priori sabemos qel peor caso ocurre cuando **div** es **1** y **n** es mayor que **1**. Entonces para el análisis a posteriori generamos valores aleatorios entre 2 y 200000 para **n** y le asignamos el valor de **1** a **div**, estos valores los enviaremos a la funcion **division2**. El resultado de esta ejecución arroja los siguientes valores.

n	Ct
26727	120
9011	111
25030	117
32021	123
31297	117
22751	126
12368	102
15795	120
13796	114
6334	108

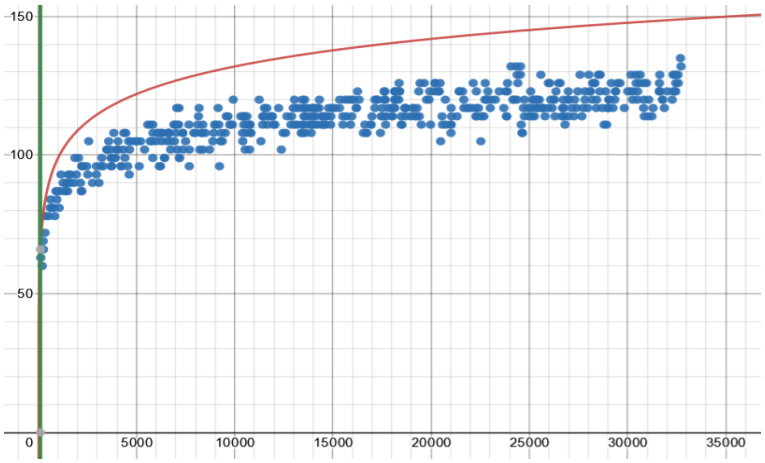
Tabla 2. Valores de la ejecución para el **segundo algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es logarítmica $O(\log n)$.



Gráfica 3. Gráfica del **segundo algoritmo**.

Ahora proponemos la función $\mathbf{g(n)} = 33 \log n$ con $n_o = 100$ para la cota superior.



Gráfica 4. Gráfica del **segundo algoritmo** con la función propuesta.

Análisis a posteriori para el algoritmo:

Comenzamos implementando el algoritmo como una función llamada **"division3"** con base en el pseudo-código del **Algorithm 3 Division3**, mediante el Lenguaje de Programación en C.

```c
int division3(int n, int div){

    if(div > n){
        return 0;
    }else{
        return 1 + division3(n - div, div);
    }
}
```

Figura 7. Código para el **tercer algoritmo** en lenguaje C.

Además, utilizaremos la función **Principal** la cual nos servirá para ejecutar el algoritmo y obtener los datos para el análisis, donde enviaremos los resultados a un archivo.

```c
void principal(){
    FILE *pf = fopen("Muestra.csv","at");
    int ct = 0, pts = 900, i;
    srand(time(NULL));

    for(i = 0;i<pts;i++){
        ct = 0;
        int n = rand()%200000;
        int q = division3(n, 1, &ct);

        printf("\nel cociente de %d y %d es %d\n", n, div, q);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 8. Código para la función **Principal** del **tercer algoritmo** en lenguaje C.

Para realizar el análisis la función **division3 (Figura 7)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número ejecuciones de cada línea.

```
int division3(int n, int div, int *ct){

    if(div > n){
        (*ct)++;
        (*ct)++;
        return 0;
    }else{
        (*ct)++;
        return 1 + division3(n - div, div, ct);
    }
    (*ct)++;
}
```

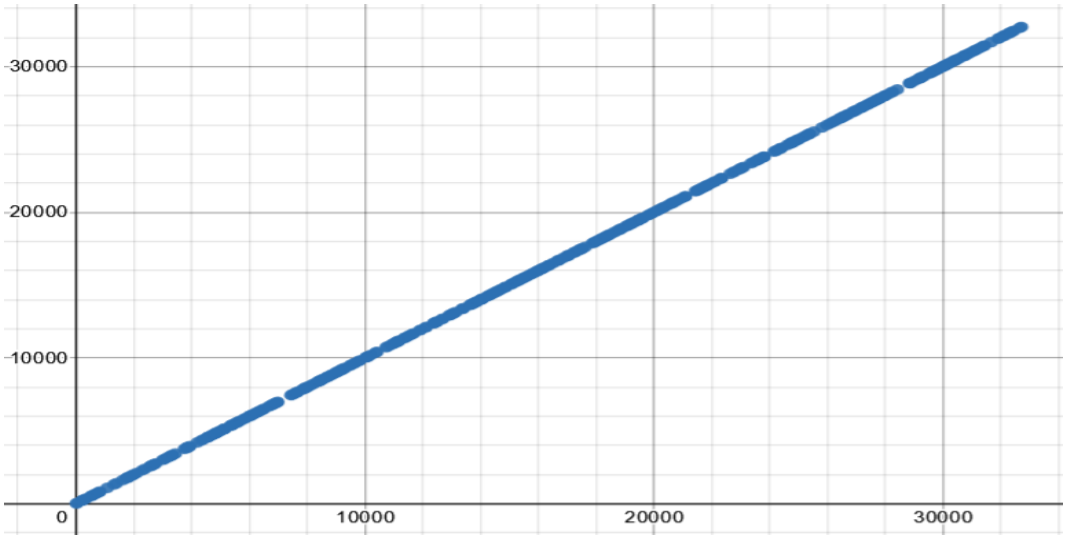
Figura 9. Código para el **tercer algoritmo** con el parámetro **ct**.

Para el análisis a priori es claro que el peor caso ocurre de acuerdo a la ecuación de recurrencia **T(n)**. Entonces para el análisis a posteriori generamos valores aleatorios entre 2 y 200000 para **n** y le asginamos el valor de **1** para **div**, estos valores los enviaremos a la funcion **division3**. El resultado de esta ejecución arroja los siguientes valores.

n	Ct
28118	28120
312	314
223	225
26609	26611
1626	1628
24370	24372
27552	27554
2761	2763
7642	7644
11147	11149

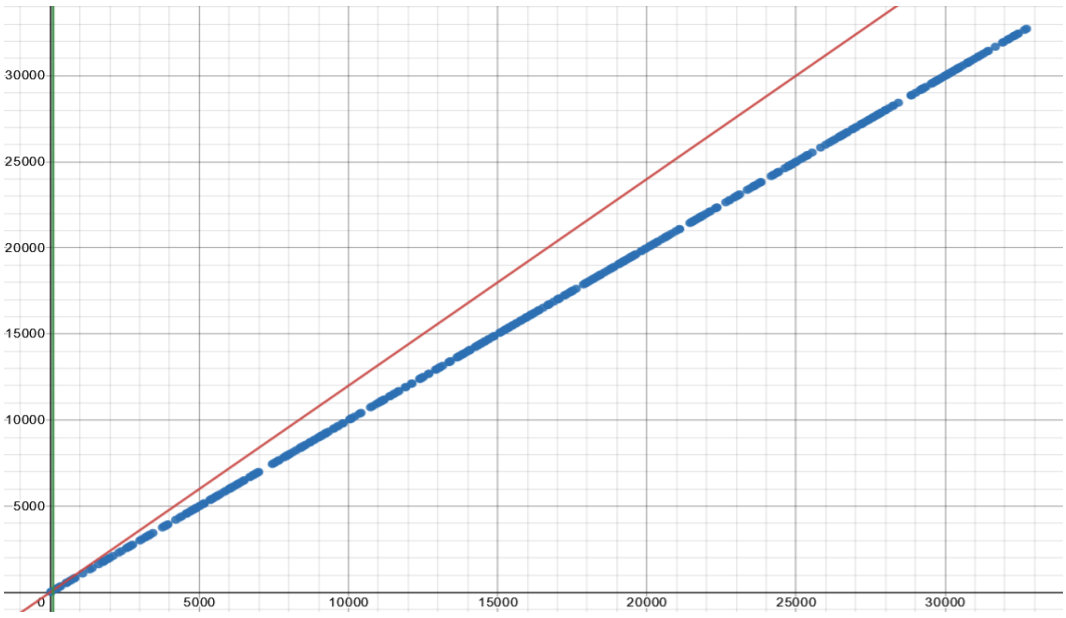
Tabla 3. Valores de la ejecución para el **tercer algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es lineal $O(n)$.



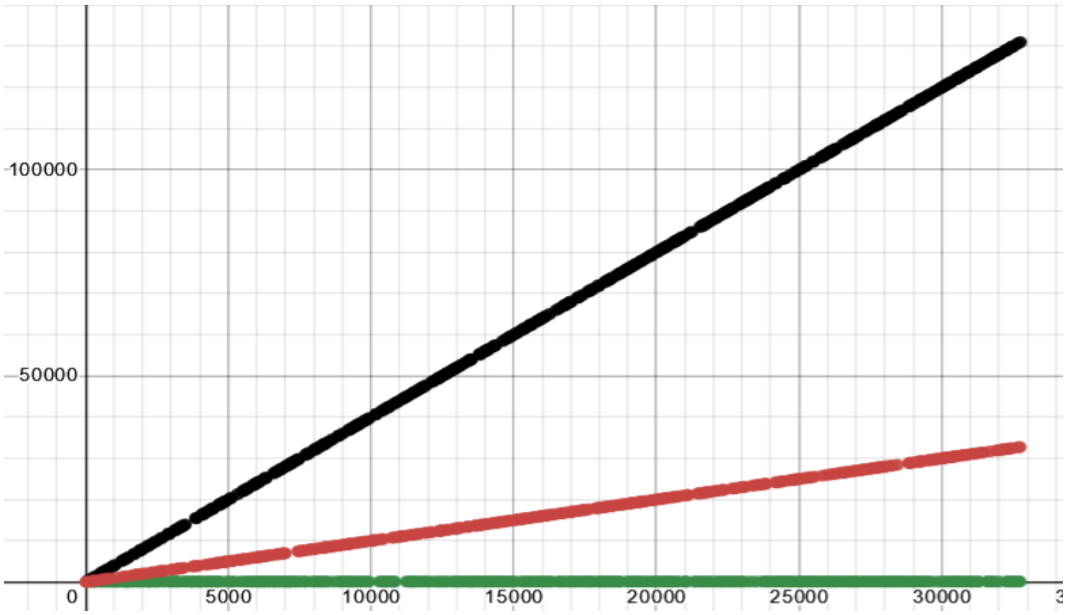
Gráfica 5. Gráfica del **tercer algoritmo**.

Ahora proponemos la función $\mathbf{g(n)} = \frac{6}{5}n \in O(n)$ con $n_o = 100$ para la cota superior.



Gráfica 6. Gráfica del **tercer algoritmo** con la función propuesta.

Comparamos los puntos obtenidos para cada algoritmo



Comparación 1. Gráficas para **division1, division2 y division3**.

Observamos la comparación para el **primer algoritmo - division1 (negro)**, **segundo algoritmo - division2 (verde)** y **tercer algoritmo - division3 (rojo)**.

Análisis a posteriori para el cuarto algoritmo:

Comenzamos implementando el algoritmo como una función llamada **"BI"** con base en el pseudo-código del **Algorithm 4 Busqueda Iterativa**, mediante el Lenguaje de Programación en C.

```c
int BI(int *A, int tam, int x){
    int i, j, inicio = 0, fin = tam-1;

    while(inicio <= fin){
        i = (inicio + ((fin-inicio)/3));
        j = (fin - ((fin-inicio)/3));

        if(A[i] == x){
            return i;
        }
        if(A[j] == x){
            return j;
        }
        else if(x < A[i]){
            fin = i - 1;
        }else if(x > A[i] && x < A[j]){
            inicio = i + 1;
            fin = j - 1;
        }else if(x > A[j]){
            inicio = j + 1;
        }
    }
    return -1;
}
```

Figura 10. Código para el **cuarto algoritmo** en lenguaje C.

Además, utilizaremos la función **Principal** para el **cuarto algoritmo**, la cual nos servirá para ejecutar y obtener los datos para el análisis, donde enviaremos los resultados a un archivo.

```c
void principal(){
    FILE *pf = fopen("Muestra.csv","at");

    int A[TAM], ct = 0, tam = 12, pts = 500, i, indice;
    srand(time(NULL));

    for(i = 0;i<pts;i++){
        ct = 0;
        int x = rand()%(tam);
        printf("\nx: %d\n", x);

        LLenaArreglo(A, tam);
        InsertionSort(A, tam);
        puts("");
        ImprimeArreglo(A, tam);

        indice = BI(A, tam, x, &ct);
        printf("\nIndice: %d\n", indice);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
        tam+=1;
    }
    fclose(pf);
}
```

Figura 11. Código para la función **Principal** del **cuarto algoritmo** en lenguaje C.

Para realizar el análisis la función **BI (Figura 10)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número ejecuciones de cada línea.

```
int BI(int *A, int tam, int x, int *ct){
    int i, j, inicio = 0, fin = tam-1;
    (*ct)++;(*ct)++;(*ct)++;(*ct)++;

    while(inicio <= fin){
        (*ct)++;(*ct)++;
        i = (inicio + ((fin-inicio)/3));(*ct)++;
        j = (fin - ((fin-inicio)/3));(*ct)++;

        if(A[i] == x){
            (*ct)++;
            return i;(*ct)++;
        }(*ct)++;
        if(A[j] == x){
            (*ct)++;
            return j;(*ct)++;
        }else if(x < A[i]){
            (*ct)++;
            fin = i - 1;(*ct)++;
        }else if(x > A[i] && x < A[j]){
            (*ct)++;(*ct)++;
            inicio = i + 1;(*ct)++;
            fin = j - 1;(*ct)++;
        }else if(x > A[j]){
            (*ct)++;
            inicio = j + 1;(*ct)++;
        }
        (*ct)++;(*ct)++;(*ct)++;(*ct)++;
    }(*ct)++;
    return -1;(*ct)++;
}
```

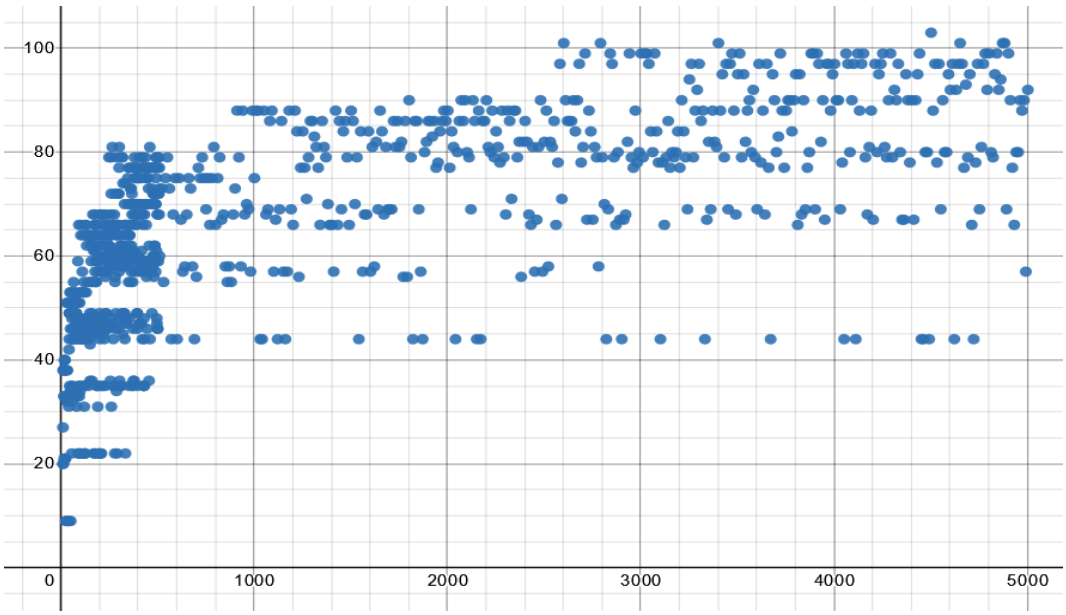
Figura 12. Código para el **cuarto algoritmo** con el parámetro **ct**.

Del análisis a priori tenemos que el mejor caso ocurre cuando el elemento **x** se encuentra en alguna de las posiciones del arreglo con los índices **j** o **i** y el peor caso ocurre cuando el elemento **x** no se encuentra en el arreglo. Entonces para el análisis a posteriori generamos arreglos con valores aleatorios, además valores entre 12 y 5000 para **tam** y valores que no se encuentren en el arreglo para **x**, estos valores los enviaremos a la funcion **BI**. El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
62	44
72	49
508	79
509	68
510	77
511	60
4972	88
4982	90
4992	57
5002	92

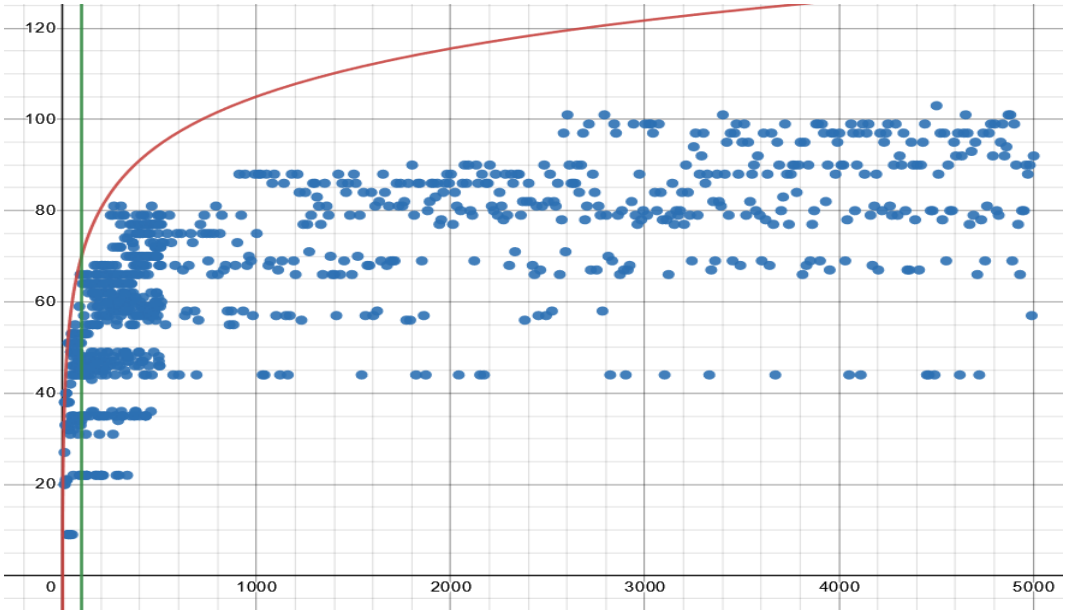
Tabla 4. Valores de la ejecución para el **cuarto algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es logaritmica $O(\log n)$.



Gráfica 7. Gráfica del **cuarto algoritmo**.

Ahora proponemos la función $\mathbf{g(n)} = 35 \log n \in O(\log n)$ con $n_o = 100$ para la cota superior.



Gráfica 8. Gráfica del **cuarto algoritmo** con la función propuesta.

Análisis a posteriori para el quinto algoritmo:

Comenzamos implementando el algoritmo como una función llamada **"BR"** con base en el pseudo-código del **Algorithm 5 Busqueda Recursiva**, mediante el Lenguaje de Programación en C.

```c
int BR(int *A, int inicio, int fin, int x){
    int i = (inicio + ((fin-inicio)/3));
    int j = (fin - ((fin-inicio)/3));

    if(inicio > fin){
        return -1;
    }
    if(A[i] == x){
        return i;
    }
    if(A[j] == x){
        return j;
    }
    else if(x < A[i]){
        return BR(A, inicio, i-1, x, ct);
    }else if(x > A[i] && x < A[j]){
        return BR(A, i+1, j-1, x, ct);
    }else if(x > A[j]){
        return BR(A, j+1, fin, x, ct);
    }
}
```

Figura 13. Código para el **quinto algoritmo** en lenguaje C.

Además, utilizaremos la función **Principal** para el **quinto algoritmo**, la cual nos servirá para ejecutar y obtener los datos para el análisis, donde enviaremos los resultados a un archivo.

```c
void principal(){
    FILE *pf = fopen("Muestra.csv","at");

    int A[TAM], ct = 0, tam = 12, pts = 500, i, indice;
    srand(time(NULL));

    for(i = 0;i<pts;i++){
        ct = 0;

        LLenaArreglo(A, tam);
        InsertionSort(A, tam);
        puts("");
        ImprimeArreglo(A, tam);
        puts("");

        int x = rand()%(tam);
        printf("\nx: %d\n", x);
        indice = BR(A, 0, tam - 1, x, &ct);
        printf("\nIndice: %d\n", indice);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
        tam+=1;
    }
    fclose(pf);
}
```

Figura 14. Código para la función **Principal** del **quinto algoritmo** en lenguaje C.

Para realizar el análisis la función **BR (Figura 13)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número ejecuciones de cada línea.

```c
int BR(int *A, int inicio, int fin, int x, int *ct){
    int i = (inicio + ((fin-inicio)/3));(*ct)++;
    int j = (fin - ((fin-inicio)/3));(*ct)++;

    if(inicio > fin){
        (*ct)++;(*ct)++;
        return -1;
    }(*ct)++;
    if(A[i] == x){
        (*ct)++;(*ct)++;
        return i;
    }
    (*ct)++;
    if(A[j] == x){
        (*ct)++;(*ct)++;
        return j;
    }
    else if(x < A[i]){
        (*ct)++;(*ct)++;
        return BR(A, inicio, i-1, x, ct);
    }else if(x > A[i] && x < A[j]){
        (*ct)++;(*ct)++;
        return BR(A, i+1, j-1, x, ct);
    }else if(x > A[j]){
        (*ct)++;(*ct)++;
        return BR(A, j+1, fin, x, ct);
    }
    (*ct)++;(*ct)++;(*ct)++;(*ct)++;
}
```

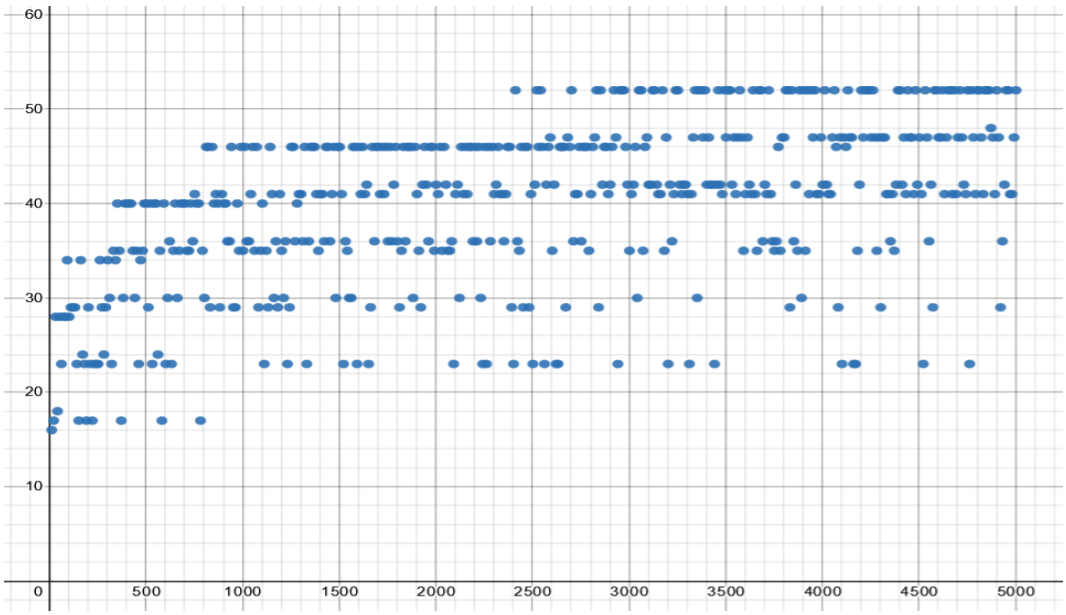
Figura 15. Código para el **quinto algoritmo** con el parámetro **ct**.

Del análisis a priori tenemos que el mejor caso ocurre cuando el elemento **x** se encuentra en alguna de las posiciones del arreglo con los índices **j** o **i** y el peor caso ocurre de acuerdo a la ecuación de recurrencia **T(N)**. Entonces para el análisis a posteriori generamos arreglos con valores aleatorios, además valores entre 12 y 5000 para **tam**, estos valores los enviaremos a la funcion **BR**. El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
232	23
272	29
282	24
2032	35
2052	42
2062	35
4962	52
4972	41
4982	47
4992	52

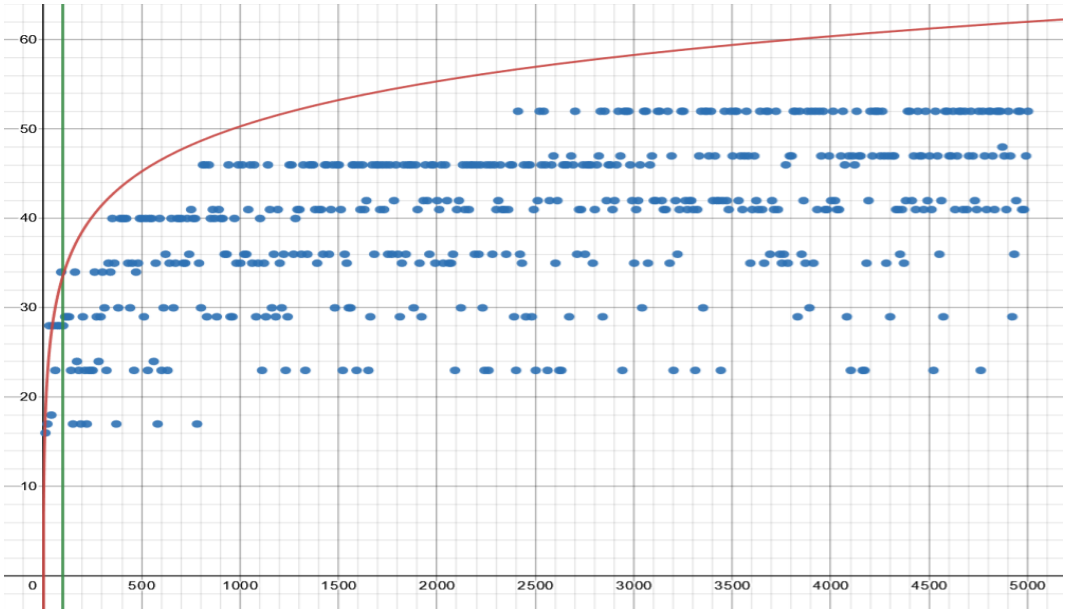
Tabla 5. Valores de la ejecución para el **quinto algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es logaritmica $O(\log_3 n)$.

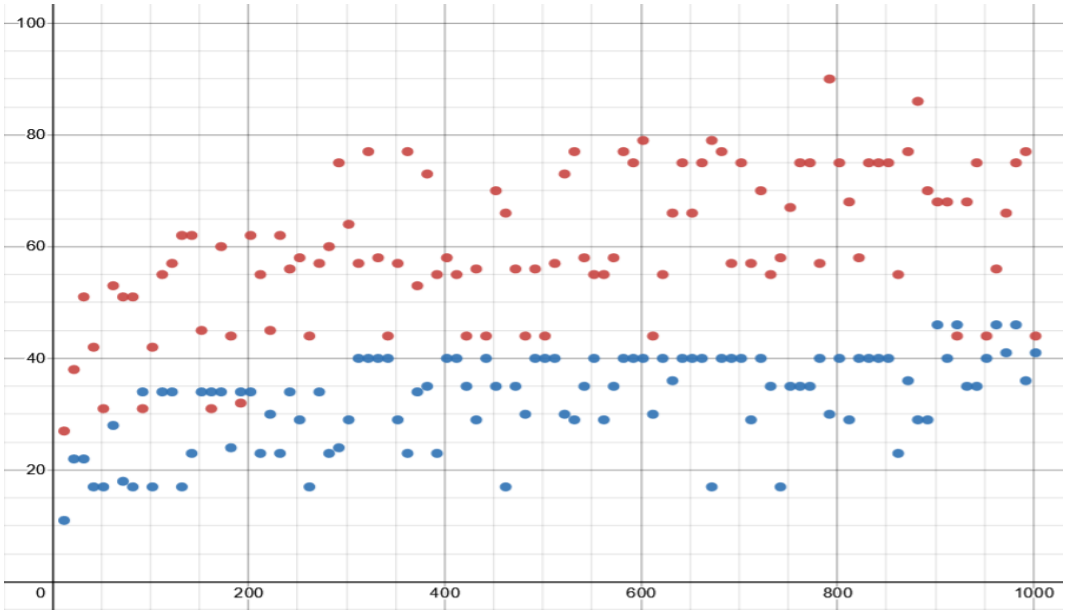


Gráfica 9. Gráfica del **quinto algoritmo**.

Ahora proponemos la función $\mathbf{g(n)} = 8\log_3 n \in O(\log_3 n)$ con $n_o = 100$ para la cota superior.



Gráfica 10. Gráfica del **quinto algoritmo** con la función propuesta.



Comparación 2. Gráficas para **BI y BR**.

Observamos la comparación para el **cuarto algoritmo - BI (rojo)** y **quinto algoritmo - BR (azul)**.

Características del equipo de cómputo:

- Procesador: Ryzen 5 1400
- Tarjeta gráfica: GTX 1050 ti de la marca PNY
- Tarjeta Madre: Asus A320M-K
- Memoria RAM: 8 Gb
- Disco duro: Disco Duro Interno 1tb Seagate

4 Conclusiones

Conclusiones generales: Durante el desarrollo de esta práctica ocurrieron algunos problemas, uno de ellos consistía en como dividir el arreglo en tres bloques y como asignar los índices **j e i**, ya que cuando necesitamos dividirlo más de una vez esto se complica, sin embargo, esto lo solucioné mediante las expresiones mostradas en los **algoritmo 4** y el **algoritmo 5**. Por otra parte, resulta complicado encontrar el peor caso para los algoritmos que determinan el cociente de dos enteros positivos. Otro problema ocurre cuando generamos más de 100 puntos con el **algoritmo 5**, lo cual dificultaba el análisis a posteriori, pero si ejecutamos el algoritmo con menos de 100 puntos y únicamente modificamos el tamaño del arreglo con cada ejecución, obtenemos los puntos necesarios para el análisis.

Conclusiones individuales (Brayan Ramirez Benitez):

Utilizamos la iteración y la recursión para ejecutar un conjunto de instrucciones de manera repetida mientras se cumpla o sea verdadera una condición en específico, sin embargo, existen algunas diferencias muy importantes entre ambas, utilizamos la recursividad para implementar algoritmos cortos y con menos código mientras que la iteración la utilizamos cuando nos presentan una forma que implica varios bucles, normalmente, el código que implementamos mediante recursividad es más eficiente que el código mediante iteración, sin embargo presenta un error llamado StackOverflow, lo cual en la iteración esto no ocurre, por lo tanto, considero que tenemos dos muy buenas maneras para resolver un problema, debemos conocer las ventajas, desventajas y su complejidad temporal para conseguir aplicar la mejor y más eficinete forma de resolver ese problema.

Brayan Ramirez Benitez.

5 Bibliografía

References

[1] Aguilar, I. (2019) *Introducción al análisis de algoritmos.* PDF. Recuperado de http://ri.uaemex.mx/bitstream /handle/20.500.11799/105198/Libro%20Complejidad.pdf? sequence=1&isAllowed=y

[2] Cormen, E. A. (2009) *Introduction to Algorithms.* 3Rd Ed. (3.a ed., Vol. 3). Phi.

[3] Martins, J. (2021, 20 marzo) *Iteración.* Recuperado de : https://conceptodefinicion.de/iteracion/

[4] Rodríguez, A. (2016) *Algoritmos recursivos.* PDF. Recuperado de : http://formacion.desarrollando.net/cursosfiles/formacion /curso_454/deda-03.pdf