Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos

Semestre: 2022-1

Grupo: 3CV11

Práctica 2

Fecha: 29 de septiembre de 2021

Práctica 2: Complejidades temporales polinomiales y no polinomiales

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

Resumen: En esta práctica realizamos el análisis a priori y posteriori para el problema de la suceción de **Fibonacci** en su versión iterativa, además, desarrollamos el análisis a posteriori para este mismo problema pero en su versión recursiva. Por otro lado, llevamos a cabo, el análisis a priori y posteriori para una función que determina si un **número es perfecto**, también, desarrollamos una función que muestra los **n** primeros **números perfectos** y realizamos el análisis a posteriori. 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, Fibonacci, Número perfecto.

1 Introducción

En computación, el análisis de algoritmos es fundamental, puesto que necesitamos encontrar el algoritmo más eficiente que resuelva un problema, sin embargo, como hemos visto un problema puede tener más de un solo algoritmo que lo resuelve, el objetivo del análisis consiste en encontrar el más eficiente y para encontrar el algoritmo más eficiente debemos comparar la complejidad temporal de cada uno de los algoritmos que resuelven un mismo problema.

La complejidad temporal es la cantidad de operaciones que emplea un algoritmo para concluir su tarea. El algoritmo que concluye su tarea con la menor cantidad de operaciones podemos considerarlo como el más eficiente. Es común que en un algoritmo la complejidad temporal sea igual para cualquier instancia de un tamaño n para un problema en específico, pero en cualquier otro caso la complejidad temporal para un algoritmo de tamaño n es diferente, esto implica analizar la complejidad temporal para un caso promedio, un mejor caso y un peor caso.

En la actualidad una gran parte de los algoritmos tienen una complejidad polinómica, es decir, la relación que existe entre el tamaño del problema y el tiempo de ejecución resulta polinómica. Entonces estos son problemas agrupados en la clase P, mientras que los problemas con costo no polinomial se encuentran agrupados en la clase NP.

Esta práctica tiene como objetivo analizar la complejidad temporal para cuatro algoritmos propuestos, dos de ellos consisten en mostrar el termino **n** de la sucesión de Fibonacci implementada de una manera iterativa y recursiva, por otro lado, un algoritmo que decide si un número es perfecto y mediante este algoritmo implementar otro algoritmo que muestre los **n** primeros números perfectos. 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 **T(n)**.

Definición: Dada una función **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 **f(n)**.

Definición: Dada una función **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 **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 $\geq n_0\ \}$

Sucesión de Fibonacci: Para Bravo, R. (2014) la sucesión de Fibonacci es una sucesión matemática infinita, que consiste en una serie de números naturales, los cuales se suman de dos en dos, comenzando en 0 y 1. En general, la sucesión de Fibonacci se consigue sumando los últimos dos números.

Como: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

Número perfecto: Zapata, F. (2020) señala que se considera perfecto a un número natural tal que es igual a la suma de sus divisores, sin incluir entre los divisores al propio número.

Primer Algoritmo: El algoritmo consiste en devolver lo que se encuentra en la posición de la sucesión de Fibonacci mediante una función implementada de manera iterativa. Primero la función recibe el número de la posición que se requiere conocer para la sucesión de Fibonacci en este caso **n**, luego declaramos tres variables **s** que nos servirá como valor siguiente de la sucesión, **a** representa el valor actual de la sucesión y **t** que nos servirá como auxiliar, además una variable **i** que funciona como iterador, luego iniciamos un ciclo **for** que tendrá como limite el número **n** que recibió la función, en este **for** guardamos el valor de **a** en nuestro auxiliar **t** posteriormente asignamos el valor de **s** a la variable **a** y finalmente sumamos las variables **s** y **t** y guardamos el resultado en **s**, una vez que concluya el ciclo **for** regresamos el valor de **a**.

Algorithm 1 *Fibonacci iterativo*

Require: Número entero
Ensure: Valor de la posición en la serie de Fibonacci

```
1: s = 1
2: a = 0
3: t = 0
4: i = 0
5: for i = 1; i ≤ n; i++ do
6:     t = a
7:     a = s
8:     s = s + t
9: end for
10: return a
```

Segundo Algoritmo: El algoritmo consiste en devolver lo que se encuentra en la posición de la sucesión de Fibonacci mediante una función implementada de manera recursiva. Primero la función recibe el número de la posición que se requiere conocer para la sucesión de Fibonacci en este caso **n**, con un **if** verificamos si el valor de **n** es menor que dos si esto se cumple regresara el valor de **n**, en cualquier otro caso llamamos recursivamente a la función hasta encontrar el valor deseado.

Algorithm 2 *Fibonacci recursivo*

Require: Número entero
Ensure: Valor de la posición en la serie de Fibonacci

```
1: if n ≺ 2 then
2:     return n
3: end if
4: return RFibonacci(n − 1) + RFibonacci(n − 2)
```

Tercer Algoritmo: El algoritmo consiste en decidir si un numero es perfecto o no. La función recibe un número **n**, luego iniciamos tres variables, la variable **i** nos servirá como iterador, la variable **s** representa la suma y **m** que representa la multiplicación, ahora iniciamos un ciclo **for** que tiene como limite el número que recibió la función, dentro del **for** asignamos a **m** el resultado de aplicar **n** modulo **i** y comparamos si es 0, en caso de ser 0 incrementamos la variable **s**, cuando termine el ciclo **for** verificamos si **s** es igual a **n**, en caso de ser iguales indica que el número es perfecto y en caso contrario el número no es perfecto.

Algorithm 3 *Número Perfecto*

Require: Número entero
Ensure: Es perfecto o no

```
1: s = 0
2: m = 0
3: i = 0
4: for i = 1; i ≤ n i++; do
5:     m = n%i
6:     if m == 0 then
7:         s+ = i
8:     end if
9: end for
10: if s == n then
11:     printf ("Es un número perfecto")
12: else
13:     printf ("No es un número perfecto")
14: end if
```

Cuarto Algoritmo: El algoritmo consiste en mostrar los **n** primeros números perfectos. Para esto modificaremos el tercer algoritmo de tal forma que devuelva un 1 o 0 en caso de que el número que recibe es perfecto o no. Ahora esta función recibe un número **n**, luego declaramos tres variables **i** que funciona como iterador, **p** que nos servirá para saber si un número es perfecto y **j** que nos servirá para almacenar los números, iniciaremos un ciclo **while** para buscar los **n** números, dentro del **while** asignaremos a **p** el resultado de determinar si **j** es un número perfecto y mediante un **if** preguntaremos si es perfecto, en caso de ser un número perfecto incrementaremos **i** y mostraremos en pantalla el número, por ultimo incrementaremos **j** para analizar el siguiente número.

Algorithm 4 *Mostrar Perfectos*

Require: Número entero
Ensure: **n** primeros números perfectos

```
1: p = 0
2: j = 1
3: i = 0
4: while i ≤ n do
5:     p = Perfecto(j)
6:     if p then
7:         printf ("j")
8:         i++
9:     end if
10:    j++
11: end while
```

3 Experimentación y Resultados

Análisis a priori para el primer algoritmo: El algoritmo para el problema de la sucesión de Fibonacci de manera iterativa. Si observamos el pseudocódigo podemos determinar que el mejor y peor caso son iguales puesto que no existen condiciones que puedan producir estos casos, por lo tanto, tiene un orden de complejidad Θ .

Entrada: Número entero
Salida: Valor de la posición en la serie de Fibonacci

Código	Costo	Número de ejecuciones
1. $s = 1$	C_1	1
2. $a = 0$	C_2	1
3. $t = 0$	C_3	1
4. $i = 0$	C_4	1
5. for $i = 1;\ i \leq n;\ i++$	C_5	n
6. $t = a$	C_6	n-1
7. $a = s$	C_7	n-1
8. $s = s + t$	C_8	n-1
9. return a	C_9	1

De lo anterior tenemos que

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5(n) + C_6(n-1) + C_7(n-1) + C_8(n-1) + C_9$

$T(n) = (C_1 + C_2 + C_3 + C_4 + C_9) + C_5(n) + C_6 n - C_6 + C_7(n) - C_7 + C_8(n) - C_8$

$T(n) = (C_1 + C_2 + C_3 + C_4 + C_9 - C_6 - C_7 - C_8) + (C_5 + C_6 + C_7 + C_8)n$

Entonces

$b = C_1 + C_2 + C_3 + C_4 + C_9 - C_6 - C_7 - C_8$ y $a = C_5 + C_6 + C_7 + C_8$

$T(n) = a(n) + b$

por lo tanto

$T(n) = a(n) + b \in \Theta(n)$

Análisis a priori para el tercer algoritmo: El algoritmo para el problema de decidir si un número es perfecto o no. Si observamos el pseudo-código podemos determinar que el mejor y peor caso son iguales, por lo tanto, tiene un orden de complejidad Θ . Sin pérdida de generalidad supongamos que **n** es un número perfecto.

Entrada: Número entero
Salida: Es perfecto o no

Código	Costo	Número de ejecuciones
1. $s = 0$	C_1	1
2. $m = 0$	C_2	1
3. $i = 0$	C_3	1
4. for $i = 1;\ i \leq n;\ i++$	C_4	n + 1
5. $m = n\%i$	C_5	n
6. $ifm == 0then$	C_6	n
7. $s+ = i$	C_7	n
8. $ifs == nthen$	C_8	1
9. $printf("Esnumeroperfecto")$	C_9	1
10. $else$	C_{10}	0
11. $printf("Noesnumeroperfecto")$	C_{11}	0

De lo anterior tenemos que

$T(n) = C_1 + C_2 + C_3 + C_4(n+1) + C_5(n) + C_6(n) + C_7(n) + C_8 + C_9 + C_{10}(0) + C_{11}(0)$

$T(n) = C_1 + C_2 + C_3 + C_8 + C_9 + C_4(n) + C_4 + C_5(n) + C_6(n) + C_7(n)$

$T(n) = (C_1 + C_2 + C_3 + C_8 + C_9 + C_4) + (C_4(n) + C_5(n) + C_6(n) + C_7(n))$

$T(n) = (C_1 + C_2 + C_3 + C_8 + C_9 + C_4) + (C_4 + C_5 + C_6 + C_7)n$

Entonces

$b = C_1 + C_2 + C_3 + C_8 + C_9 + C_4$ y $a = C_4 + C_5 + C_6 + C_7$

$T(n) = a(n) + b$

por lo tanto

$T(n) = a(n) + b \in \Theta(n)$

Análisis a posteriori para el primer algoritmo:

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

```c
unsigned long IFibonacci(int n){
    unsigned long s = 1, a = 0, t = 0;
    int i = 0;

    for(i = 1;i <= n;i++){
        t = a;
        a = s;
        s = s + t;
    }
    return a;
}
```

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, tam = 0, pts = 900, i;
    unsigned long f1 = 0;

    for(i = 0;i<pts;i++){
        ct = 0;
        tam = rand()%(40);

        f1 = IFibonacci(tam, &ct);

        printf("\n fibonacci de %d es %d \n", tam, f1);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
    }
    fclose(pf);
}
```

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

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

```
unsigned long IFibonacci(int n, int *ct){
    unsigned long s = 1, a = 0, t = 0;(*ct)++;(*ct)++;(*ct)++;
    int i = 0;(*ct)++;

    (*ct)++;
    for(i = 1;i <= n;i++){
        (*ct)++;
        t = a;(*ct)++;
        a = s;(*ct)++;
        s = s + t;(*ct)++;
        (*ct)++;
    }
    (*ct)++;
    (*ct)++;
    return a;
}
```

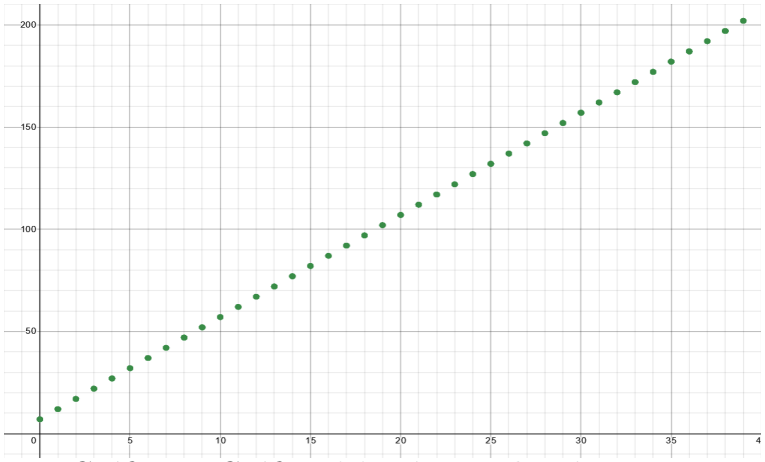
Figura 3. Código para el **primer algoritmo** con el parámetro **ct**.

Del análisis a priori sabemos que el mejor y peor caso son iguales puesto que no existen condiciones que puedan producir estos casos. Entonces para el análisis a posteriori generamos valores aleatorios entre 0 y 40 para la funcion **IFibonacci**. El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
1	12
27	142
14	77
20	107
9	52
4	27
38	197
2	17
24	127
25	132

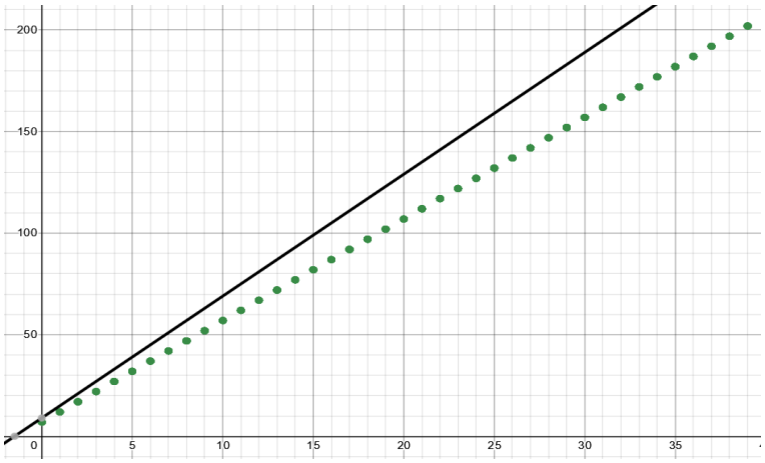
Tabla 1. Valores de la ejecución para el **primer algoritmo**.

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



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

Ahora proponemos la función $6(n) + 9 \in \Theta(n)$ con $n_o = 1$ 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 **"RFibonacci"** con base en el pseudo-código del **Algorithm 2 Fibonacci Recursivo**, mediante el Lenguaje de Programación en C.

```
unsigned long RFibonacci(int n){
    if (n < 2){
        return n;
    }
    return RFibonacci(n - 1, ct) + RFibonacci(n - 2, ct);
}
```

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.

```
void principal(){
    FILE *pf = fopen("Muestra.csv","at");
    int ct = 0, tam = 0, pts = 900, i;
    unsigned long f1 = 0;

    for(i = 0;i<pts;i++){
        ct = 0;
        tam = rand()%(40);

        f1 = RFibonacci(tam, &ct);

        printf("\n fibonacci de %d es %d \n", tam, f1);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
    }
    fclose(pf);
}
```

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

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

```
unsigned long RFibonacci(int n, int *ct){
    if (n < 2){
        (*ct)++;
        (*ct)++;
        return n;
    }
    (*ct)++;
    return RFibonacci(n - 1, ct) + RFibonacci(n - 2, ct);
}
```

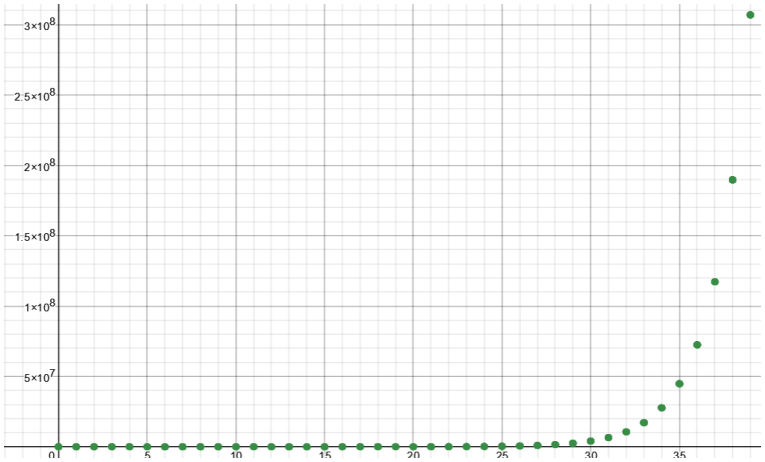
Figura 6. Código para el **segundo algoritmo** con el parámetro **ct**.

Analizando el pseudo-código podemos observar que el mejor y peor caso son iguales puesto que no existen condiciones que puedan producir estos casos. Entonces para el análisis a posteriori generamos valores aleatorios entre 0 y 40 para la funcion **RFibonacci**. El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
1	2
27	953432
14	1829
20	32837
9	164
4	14
38	189737957
2	5
24	225074
25	364178

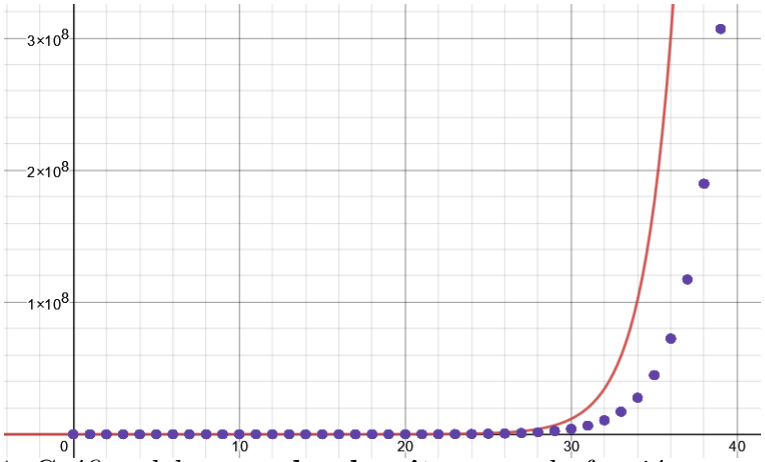
Tabla 2. Valores de la ejecución para el **segundo algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es exponencial.



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

Ahora proponemos la función $(\frac{1+\sqrt{5}}{2})^n$ con $n_o = 30$ para la cota superior.



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

Análisis a posteriori para el tercer algoritmo:

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

```c
void Perfecto(int n, int *ct){
    int i = 0, s = 0, m = 0;

    for(i = 1; i<n; i++){
        m = tam%i;
        if(m == 0){
            s+=i;
        }
    }

    if(s == n){
        printf("\n%d es un numero perfecto.\n", n);
    }else{
        printf("\n%d no es un numero perfecto.\n", n);
    }
}
```

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, tam = 0, pts = 900, i;

    for(i = 0;i<pts;i++){
        ct = 0;
        tam = rand()%(10000);

        Perfecto(tam, &ct);

        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
    }
    fclose(pf);
}
```

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

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

```
void Perfecto(int n, int *ct){
    int i = 0, s = 0, m = 0;(*ct)++;(*ct)++;(*ct)++;

    (*ct)++;
    for(i = 1; i<n; i++){
        (*ct)++;
        m = n%i;(*ct)++;
        if(m == 0){
            (*ct)++;
            (*ct)++;
            s+=i;
        }
        (*ct)++;
        (*ct)++;
    }
    (*ct)++;

    if(s == n){
        (*ct)++;
        (*ct)++;
        printf("\n%d es un numero perfecto.\n", n);
    }else{
        (*ct)++;
        (*ct)++;
        printf("\n%d no es un numero perfecto.\n", n);
    }
}
```

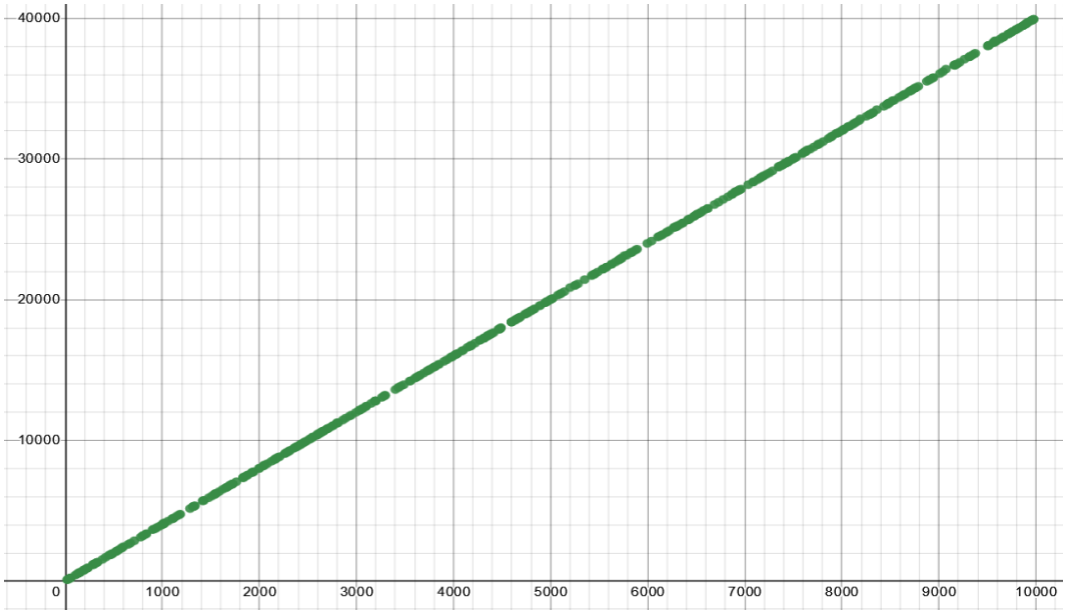
Figura 9. Código para el **tercer algoritmo** con el parámetro **ct**.

El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
41	169
8467	33873
6334	25345
6500	26049
9169	36685
5724	22945
1478	5921
9358	37441
6962	27861
4464	17917

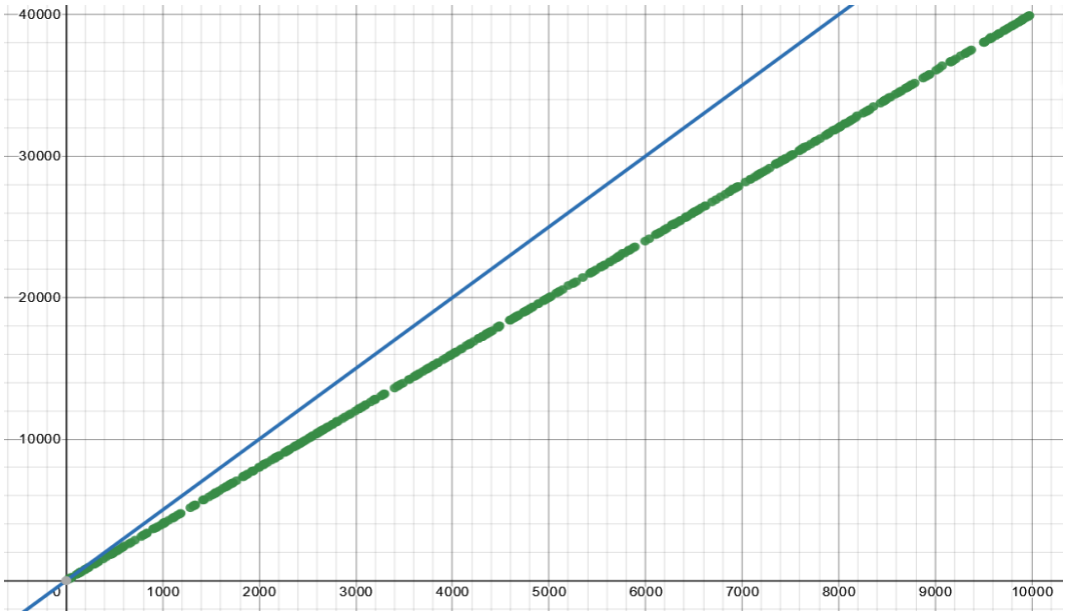
Tabla 3. Valores de la ejecución para el **tercer algoritmo**.

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



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

Ahora proponemos la función $5n \in \Theta(n)$ con $n_o = 30$ para la cota superior.



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

Análisis a posteriori para el cuarto algoritmo:

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

```c
void MostrarPerfectos(int n){
    int i = 0, p = 0, j = 1;

    while(i < n){
        p = Perfecto(j);
        if(p){
            printf(" %d ", j);
            i++;
        }
        j++;
    }
}
```

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 ct = 0, tam = 0, pts = 900, i;

    for(i = 0;i<pts;i++){
        ct = 0;
        tam = rand()%(4);

        MostrarPerfectos(tam, &ct);

        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
    }
    fclose(pf);
}
```

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

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

```
void MostrarPerfectos(int n, int *ct){
    int i = 0, p = 0, j = 1;(*ct)++;(*ct)++;(*ct)++;

    while(i < n){
        (*ct)++;
        (*ct)++;
        p = Perfecto(j);(*ct)++;
        if(p){
            (*ct)++;
            (*ct)++;
            printf(" %d ", j);
            i++;(*ct)++;
        }
        (*ct)++;
        j++;(*ct)++;
    }
    (*ct)++;
    (*ct)++;
}
```

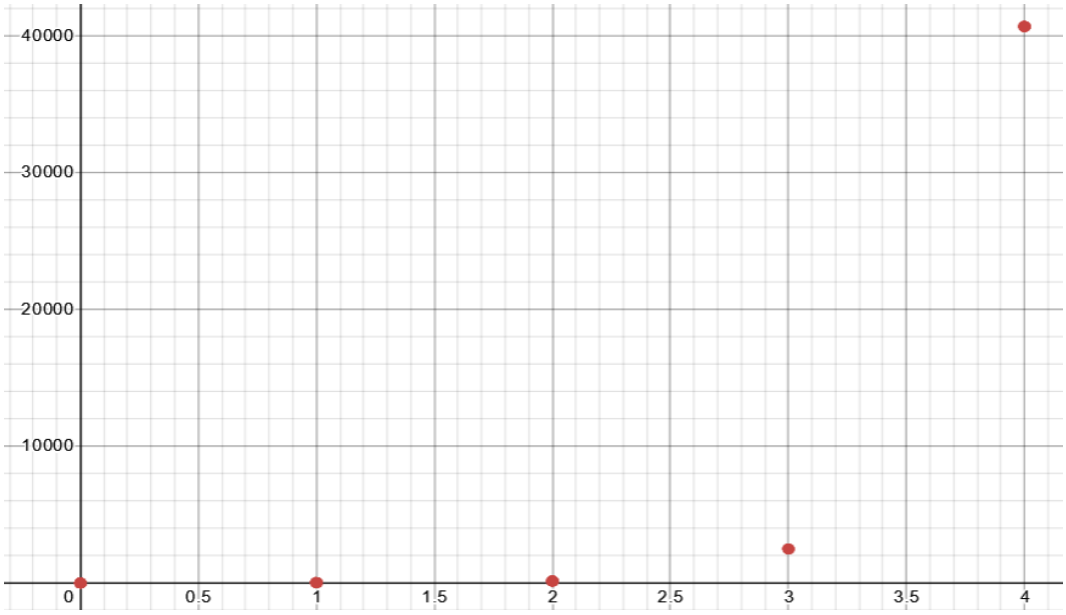
Figura 12. Código para el **cuarto algoritmo** con el parámetro **ct**.

El resultado de esta ejecución arroja los siguientes valores.

Tam	Ct
1	38
2	151
4	40657
0	5
3	2494
1	38
4	40657
3	2494
4	40657
2	151

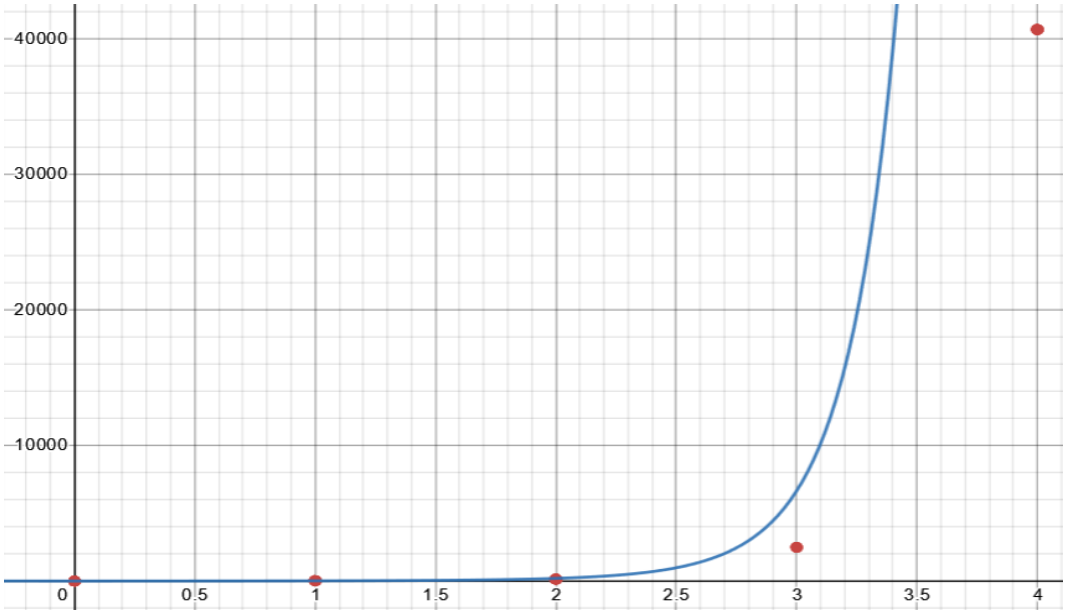
Tabla 4. Valores de la ejecución para el **cuarto algoritmo**.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que es exponencial $\Theta(2^{n^2})$.



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

Ahora proponemos la función $13 \cdot 2^{n^2} \in \Theta(2^{n^2})$ con $n_o = 2$ para la cota superior.



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

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 que al implementar el segundo algoritmo no entendiamos como enviarle el parametro por referencia (ct) a la función recursiva, este problema fue solucionado despues de observar un ejemplo mostrado en clase, de esta manera modificamos la función principal y RFibonacci para enviar el parametro ct. Por otro lado, al implementar la función MostrarPerfectos surgieron algunas dudas acerca de como implementarla puesto que cuando ejecutábamos el programa con un número mayor a 4 no se detenia.

Conclusiones individuales (Brayan Ramirez Benitez):

Es de gran importancia siempre que estemos trabajando con un algoritmo, realizar un análisis de la complejidad para el algoritmo, sin embargo, no todos los algoritmos son fáciles de analizar, normalmente para algunos algoritmos es sencillo determinar el orden de complejidad del mismo. Si conseguimos determinar la complejidad del algoritmo con el que estamos trabajando nos puede dar una idea muy clara de cómo se va a comportar, de esta forma con base en este análisis y los tamaños de la entrada podríamos ser capaces de determinar si el algoritmo será el más eficiente o no.

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] Bravo, R. (2014) *La sucesión de Fibonacci en el diseño.* Recuperado de https://www.eade.es/blog/186-la-sucesion-de-fibonacci-en-el-diseno

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

[4] Doyata, H. (2017) *Algoritmos.* Importancia. Recuperado de https://www.importancia.org/algoritmos.php

[5] Zapata, F. (2020) *Números perfectos: cómo identificarlos y ejemplos.* Recuperado de https://www.lifeder.com/numeros -perfectos/