Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos

Semestre: 2022-1

Grupo: 3CV11

Práctica 1

Fecha: 7 de septiembre de 2021

Práctica 1: Determinación experimental de la complejidad temporal de un algoritmo

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

Resumen: En esta práctica realizamos el análisis a posteriori para dos algoritmos, particularmente un algoritmo que convierte en decimal el resultado de aplicar el operador binario (&) a dos arreglos de tamaño n donde analizaremos el peor y mejor caso, por otro lado, el algoritmo de Euclides el cual nos permite encontrar el máximo común divisor (MCD) para dos números enteros donde analizaremos el peor caso, el cual se encuentra relacionado con la sucesión de Fibonacci. Para esto, los algoritmos se implementarán mediante el uso del lenguaje de programación C, además realizaremos el análisis a posteriori mediante el uso de una Calculadora Grafica del sitio Web Desmos.

Palabras Clave: Lenguaje C, Euclides, Algoritmo, Análisis Posteriori.

1 Introducción

Tanto en ciencias de la computación como en matemáticas, un algoritmo se define como un conjunto de instrucciones que tienen la característica de estar ordenadas, definidas y que son finitas, las cuales nos permiten resolver un problema. Los algoritmos son muy importantes para las disciplinas antes mencionadas puesto que con ellos podemos resolver una enorme cantidad de problemas. Un algoritmo en computación no es un programa por si mismo, el programa es resultado de la implementación del algoritmo mediante un lenguaje de programación de alto nivel, de esta manera la maquina puede compilarlo. Es posible expresar un algoritmo de diferentes maneras una de ellas es mediante el lenguaje natural, ya que es posible describir de una manera clara y concisa el conjunto de instrucciones para completar una tarea, aunque esta no es una forma estandarizada, por otro lado, tenemos los diagramas de flujo los cuales si son una manera estandarizada para expresar un algoritmo de forma gráfica, que representen las acciones y decisiones que se llevan a cabo durante la ejecución de un algoritmo. Además, el pseudo-código funciona como una representación más formal en texto, con varias similitudes a la de un lenguaje de programación de alto nivel, el cual expresa un algoritmo y es más sencillo convertirlo en código para compilarlo.

El análisis de algoritmos es una parte fundamental para una búsqueda de algoritmos confiables y eficientes, es decir implica predecir los recursos que un algoritmo requiere para ejecutarse. Este análisis consiste en un proceso para evaluar el desempeño en tiempo de ejecución para llegar a la solución, además nos brinda estimaciones teóricas sobre que recursos son necesarios para que un algoritmo sea capaz de resolver un problema. Así que, cuando un algoritmo es implementado en un lenguaje de programación de alto nivel, necesita de una depuración y pruebas para garantizar que este libre de cualquier tipo de error y cumpla su objetivo de manera eficaz. Además, como resultado del análisis de algoritmos obtenemos un conocimiento más profundo sobre los problemas, así como para las posibles soluciones. Por lo tanto, los resultados del análisis de algoritmos son de gran importancia para la elección de soluciones para un problema, haciendo énfasis en el contexto y de esta manera poder apreciar cuando no exista un algoritmo para resolver este problema. Normalmente, si analizamos varios algoritmos para un problema en común, es posible encontrar el más eficiente entre ellos.

Esta práctica tiene como objetivo analizar la complejidad temporal y eficacia de los dos algoritmos propuestos, un algoritmo para mostrar en decimal el resultado de utilizar un operador binario en dos arreglos y el algoritmo de Euclides para encontrar el máximo común divisor (MCD) de dos números enteros. Para cada algoritmo realizaremos varios experimentos con iteraciones y un análisis comparativo.

2 Conceptos Básicos

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 $\geq n_0\ \}$

Primer Algoritmo: El algoritmo consiste en devolver un numero decimal del resultado de aplicar el operador binario and (&) a dos arreglos de tamaño n. Iniciaremos declarando tres variables, la variable decimal que iniciara en 0, un iterador que de igual manera comenzara en 0 y una variable para las potencias de 2 que corresponden a cada posición de los arreglos, luego iniciaremos un ciclo **for** para recorrer ambos arreglos de derecha a izquierda, posteriormente mediante un **if** comprobaremos si el resultado de aplicar el operador (&) a dos posiciones de los arreglos, si el resultado es uno aumentaremos el valor decimal con la potencia que le corresponde a la posición del arreglo, en caso contrario no incrementamos el valor decimal, luego aumentamos la potencia para que corresponda a las posiciones de los arreglos y por ultimo regresaremos el valor decimal.

```
proceso 1:

1   int decimal = 0
2   int i = 0
3   int potencia = 1
4   for (i = n-1 to i>=0){
5       if(A[i]&B[i])
6           decimal+=potencia
7       potencia = potencia*2
8   }
9   return decimal
```

Figura 1. Pseudo-código para el **primer algoritmo**.

Segundo Algoritmo: El algoritmo de **Euclides** consiste en encontrar el máximo común divisor (MCD) de una manera rápida para dos números enteros. Comenzaremos declarando una variable r que por el momento comenzara en 0, luego iniciaremos un ciclo **while** que se ejecutará siempre que n sea distinto de 0, dentro del **while** le asignaremos el valor de m modulo n, a m le asignaremos el valor de n y a n le asignaremos el valor de r, por último, cuando termine nos devolverá el valor de m que corresponde al **MCD** de m y n.

```
proceso 2:

1   int r = 0
2   while(n != 0){
3       r = m%n
4       m = n
5       n = r
6   }
7   return m
```

Figura 2. Pseudo-código para el **segundo algoritmo (Euclides)**.

3 Experimentación y Resultados

Para el primer algoritmo:

Comenzaremos implementando el algoritmo como una funció llamada **"ProgramaUno"** con base en el pseudo-código de la **Figura 1**, mediante el Lenguaje de Programación C.

```c
int ProgramaUno(int *A, int *B, int n){
    int decimal = 0;
    int i = 0;
    int potencia = 1;

    for(i = n-1; i>=0 ; i--){
        (*ct)++;
        if(A[i]&B[i]){
            decimal+=potencia;
        }
        potencia = potencia*2;
    }
    return decimal;
}
```

Figura 3. Código para el **primer algoritmo** en lenguaje C.

Además, utilizaremos las siguientes funciones:

- **ImprimeArreglo:** Esta función nos permite mostrar un arreglo en pantalla.

```c
void ImprimeArreglo(int *A, int n){
    int i;

    for(i = 0; i<n ; i++)
        printf("[ %d ]",  A[i]);
}
```

Figura 4. Código para la función **ImprimeArreglo** en lenguaje C.

- **CreaDosArreglo:** La función genera dos arreglos y los llena con valores aleatorios o constantes.

```c
void CreaDosArreglo(int *A, int *B, int n){
    srand(time(NULL));
    int i;

    for(i = 0; i<n ;i++)
        A[i] = rand()%2;

    for(i = 0; i<n ;i++)
        B[i] = rand()%2;
}
```

Figura 5. Código para la función **CreaDosArreglo** en lenguaje C.

- **Principal:** Nos servirá para ejecutar el algoritmo haciendo uso de las funciones anteriores para generar dos arreglos e imprimirlos, además de llamar a la función que contiene 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 A[TAM], B[TAM], tam = 10, ct = 0, i, pts = 20, decimal = 0;

    for(i = 0;i<pts;i++){
        ct = 0;
        CreaDosArreglo(A, B, tam);
        ImprimeArreglo(A, tam);
        puts("");
        ImprimeArreglo(B, tam);
        puts("");

        decimal = ProgramaUno(A, B, tam, &ct);

        printf("\n Numero en decimal: %d\n",decimal);
        printf("\n Numero de pasos: %d\n",ct);
        fprintf(pf,"%d,%d\n\n", tam, ct);
        tam+=1;
    }
    fclose(pf);
}
```

Figura 6. Código para la función **Principal** en lenguaje C.

Para realizar el análisis la función **ProgramaUno (Figura 3)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número de ocasiones en las cuales se ejecuta cada línea.

```
int ProgramaUno(int *A, int *B, int n, int *ct){
    int decimal = 0;(*ct)++;
    int i = 0;(*ct)++;
    int potencia = 1;(*ct)++;
    (*ct)++;
    for(i = n-1; i>=0 ; i--){
        (*ct)++;
        if(A[i]&B[i]){
            (*ct)++;
            (*ct)++;
            decimal+=potencia;
        }
        (*ct)++;
        (*ct)++;
        potencia = potencia*2;
    }
    (*ct)++;
    (*ct)++;
    return decimal;
}
```

Figura 7. Código para el **primer algoritmo** en lenguaje C con el parámetro **ct**.

Analizando el algoritmo podemos observar que el mejor y peor caso son iguales puesto que no importa que valores contienen, el algoritmo siempre revisará cada posición de ambos arreglos, pero si contienen únicamente 1 siempre se cumplirá el **if** y por lo tanto ejecutara lo que contiene para cada posición de los arreglos. Entonces para el peor caso modificaremos la función **CreaDosArreglo** asignándole un 1 a cada posición de ambos arreglos, mientras que para otros casos generamos valores aleatorios entre 0 y 1 para ambos arreglos.

Ejecución del programa para el peor caso:

```
void CreaDosArreglo(int *A, int *B, int n){
    srand(time(NULL));
    int i;

    for(i = 0; i<n ;i++)
        A[i] = 1;

    for(i = 0; i<n ;i++)
        B[i] = 1;
}
```

Figura 8. Código de la función **CreaDosArreglo** para el peor caso.

En la función **CreaDosArreglo** asignamos el valor 1 para cada posición de los arreglos.

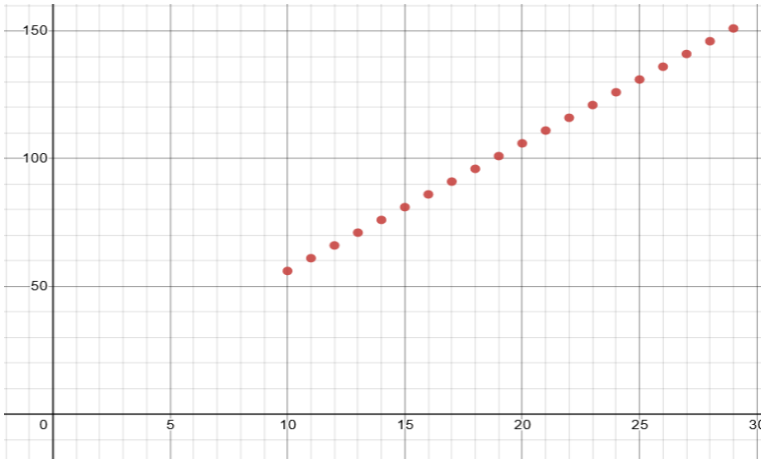


Figura 9. Gráfica del algoritmo para el peor caso.

Copiamos los valores que resultaron de la ejecución del programa para el peor caso y los graficamos mediante la calculadora Gráfica Desmos, observamos que es lineal.

Ejecución del programa para valores aleatorios:

```
void CreaDosArreglo(int *A, int *B, int n){
    srand(time(NULL));
    int i;

    for(i = 0; i<n ;i++)
        A[i] = rand()%2;

    for(i = 0; i<n ;i++)
        B[i] = rand()%2;
}
```

Figura 10. Código de la función **CreaDosArreglo** para valores aleatorios.

Ahora asignamos valores aleatorios entre 0 y 1 para cada posición de los arreglos.

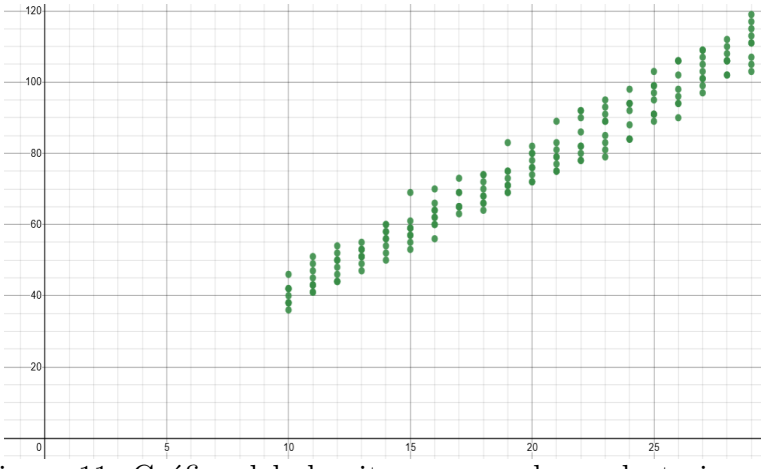


Figura 11. Gráfica del algoritmo para valores aleatorios.

Copiamos los valores que resultaron de la ejecución del programa para valores aleatorios y los graficamos mediante la calculadora Gráfica Desmos, observamos que estan por debajo del peor caso.

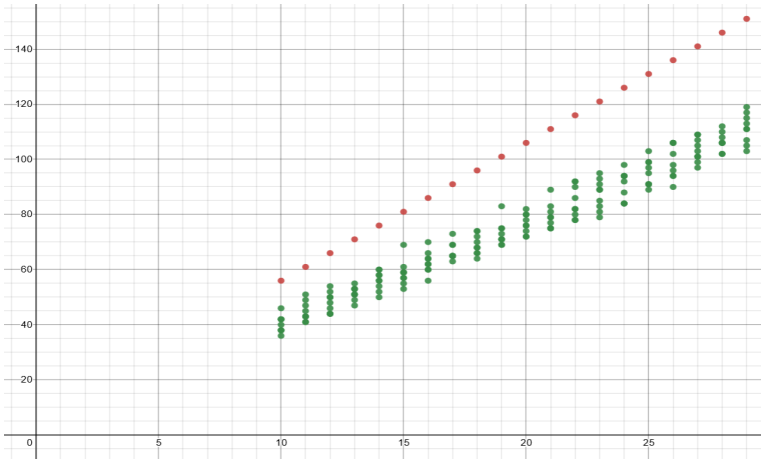


Figura 12. Gráfica del algoritmo para el peor caso y valores aleatorios.

Notamos que tiene orden lineal y proponemos la función **6x** para la cota superior.

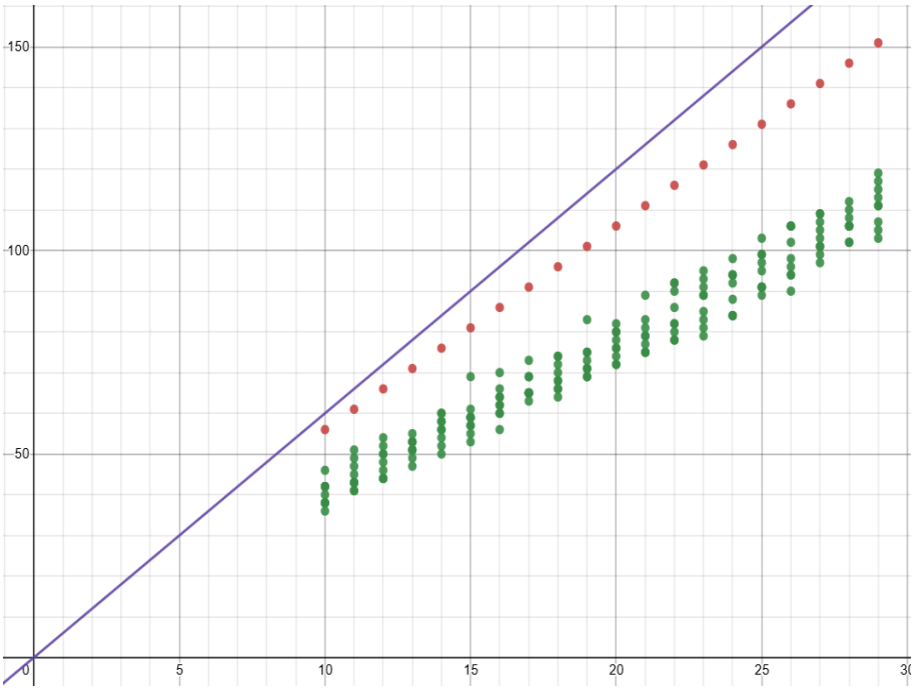


Figura 13. Gráfica con la función propuesta.

Para el segundo algoritmo:

Iniciamos implementando el algoritmo como la función **"Euclides"** con base en el pseudo-código de la **Figura 2**, mediante el Lenguaje de Programación en C.

```c
int Euclides(int m, int n){
    int r = 0;

    while(n != 0){
        r = m%n;
        m = n;
        n = r;
    }
    return m;
}
```

Figura 14. Código para el algoritmo de **Euclides** en lenguaje C.

De igual manera, para el algoritmo utilizaremos las siguientes funciones:

- **RellenaFib:** Esta función nos permite guardar en un arreglo los valores de la sucesión de Fibonacci.

```c
void RellenaFib(int *A, int n){
    int s, i, aux = 0, a = 1;

    A[0] = 1;
    for(i = 1; i<n ;i++){
        s = a + aux;
        A[i] = s;
        aux = a;
        a = s;
    }
}
```

Figura 15. Código para la función **RellenaFib** en lenguaje C.

- **Principal:** Nos servirá para ejecutar el algoritmo haciendo uso de las funciones anteriores para generar un arreglo y asignarle los valores de la sucesión de Fibonacci, además de llamar a la función Euclides que contiene el algoritmo y obtener los datos para el análisis, los cuales enviaremos a un archivo.

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

    int A[TAM], ct = 0, i = 0, m = 0, n = 0, pts = 30, mcd = 0, tam = 100;

    RellenaFib(A, tam);

    for(i = 0;i<pts;i++){
        ct = 0;
        m = A[i];// rand()%TAM;
        n = A[i+1];// rand()%TAM;

        mcd = Euclides(m, n, &ct);

        printf("\n MCD de los numeros %d y %d : %d\n", m, n, mcd);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 16. Código para la función **Principal** en lenguaje C.

Para realizar el análisis, la función **Euclides (Figura 14)** debe recibir un parámetro por referencia **(ct)** que obtendrá el número de ocasiones en las cuales se ejecuta cada línea.

```c
int Euclides(int m, int n, int *ct){
    int r = 0;(*ct)++;
    (*ct)++;
    while(n != 0){
        (*ct)++;
        r = m%n;(*ct)++;
        m = n;(*ct)++;
        n = r;(*ct)++;
    }
    (*ct)++;
    (*ct)++;
    return m;
}
```

Figura 17. Código para el algoritmo **Euclides** en lenguaje C con el parámetro **ct**.

Para este algoritmo únicamente analizaremos el peor caso, el cual está relacionado con los valores de la sucesión de Fibonacci. Entonces en la funciń **principal**, generaremos un arreglo para asignar los valores de la sucesión en cada una de las posiciones, haciendo uso de la función **RellenaFib**, posteriormente le asignaremos a m y n los valores que contiene el arreglo.

Ejecución del programa para el caso:

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

    int A[TAM], ct = 0, i = 0, m = 0, n = 0, pts = 30, mcd = 0, tam = 100;

    RellenaFib(A, tam);

    for(i = 0;i<pts;i++){
        ct = 0;
        m = A[i];
        n = A[i+1];

        mcd = Euclides(m, n, &ct);

        printf("\n MCD de los numeros %d y %d : %d\n", m, n, mcd);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 18. Código para el peor caso de la función **Principal** en lenguaje C.

Copiamos los valores que resultaron de la ejecución del programa para el peor caso y los graficamos mediante la calculadora Gráfica Desmos, observamos que tiene forma logarítmica.

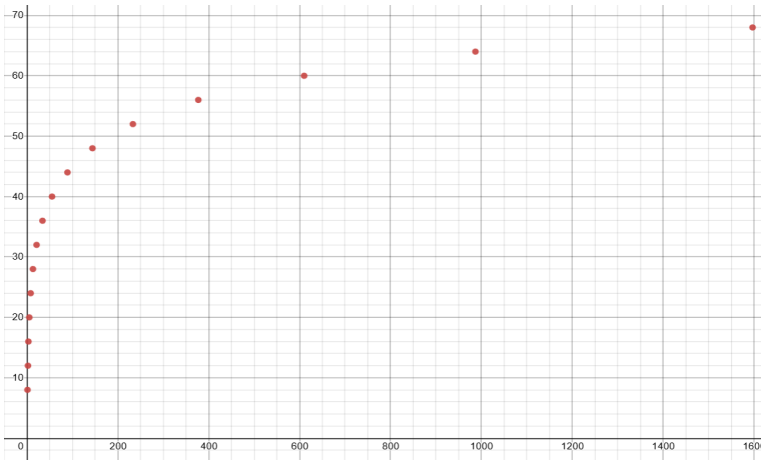


Figura 19. Gráfica del algoritmo para el peor caso.

Ejecución del programa para valores aleatorios:

Para los valores aleatorios modificamos la función **Principal** asignando a m y n valores generados aleatoriamente.

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

    int A[TAM], ct = 0, i = 0, m = 0, n = 0, pts = 30, mcd = 0, tam = 100;

    RellenaFib(A, tam);

    for(i = 0;i<pts;i++){
        ct = 0;
        m = rand()%TAM;
        n = rand()%TAM;

        mcd = Euclides(m, n, &ct);

        printf("\n MCD de los numeros %d y %d : %d\n", m, n, mcd);
        printf("\n Numero de pasos: %d\n\n",ct);
        fprintf(pf,"%d,%d\n\n", n, ct);
    }
    fclose(pf);
}
```

Figura 20. Código para valores aleatorios de la función **Principal** en lenguaje C.

Copiamos los valores que resultaron de la ejecución del programa y los graficamos mediante la calculadora Gráfica Desmos, observamos que estan por debajo del peor caso.

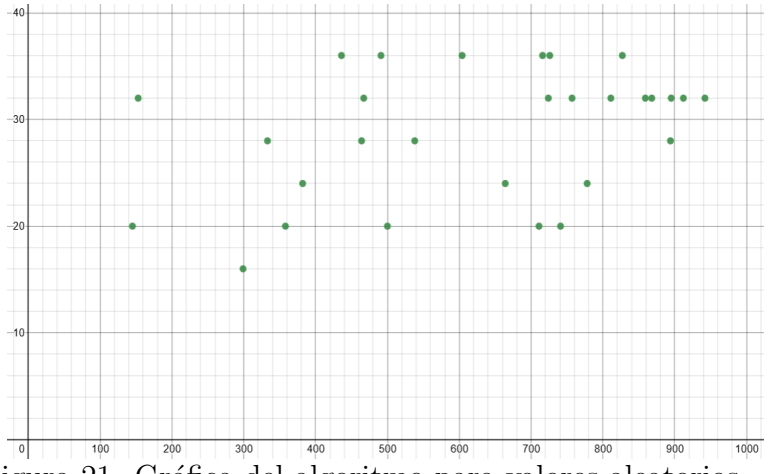


Figura 21. Gráfica del algoritmo para valores aleatorios.

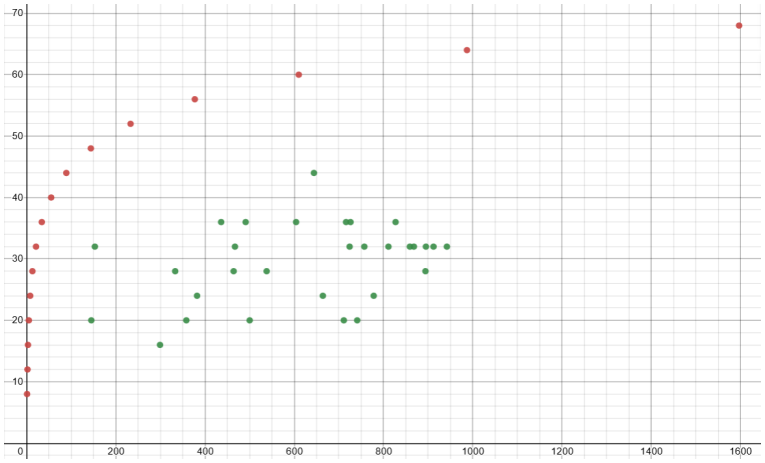


Figura 22. Gráfica del algoritmo para valores aleatorios y el peor caso.

Observamos que es de orden logarítmica y proponemos la función $\mathbf{7}\log_2 x$ para la cota superior.

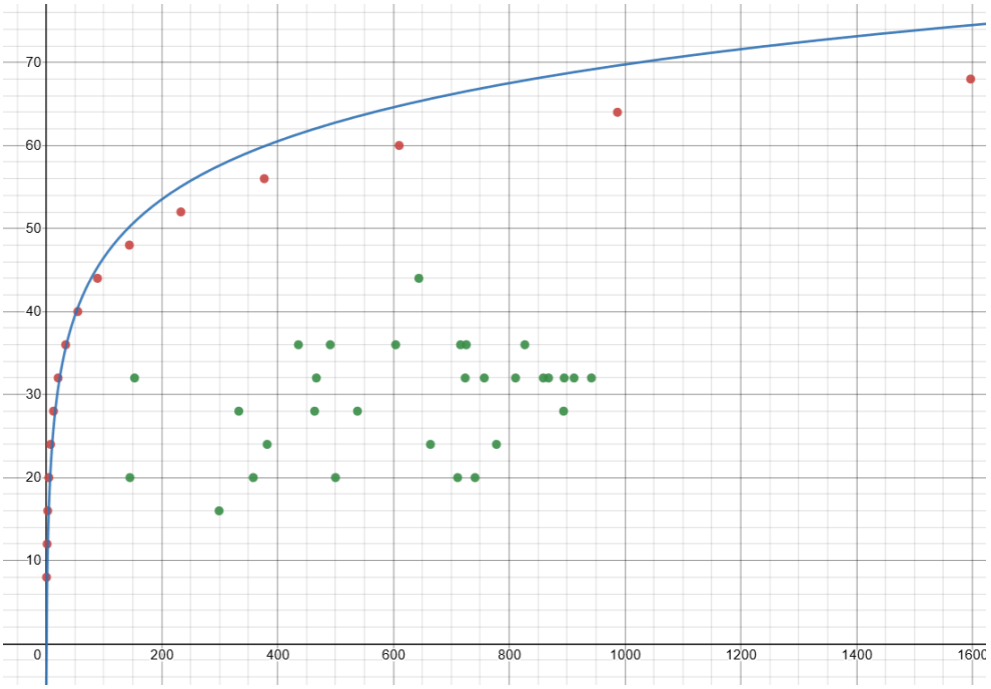
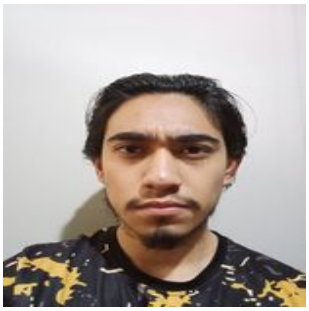


Figura 23. Gráfica con la función propuesta.

4 Conclusiones

Conclusiones generales: Durante el desarrollo de esta práctica ocurrieron algunos problemas, uno de ellos consistía en que al ejecutar el programa los algoritmos no generaban el archivo donde se guardan los resultados de la ejecución, este problema fue solucionado modificando algunos parámetros en la función principal. Por otra parte, al implementar el primer algoritmo surgieron varias posibles soluciones, una de ellas solo utilizaba una línea de código, pero resultaba difícil realizar el análisis puesto que generaba algunas dudas de cómo interpretarlo, entonces para implementar el algoritmo, fue elegida la solución más simple y comprensible, de esta manera es fácil identificar que hace el algoritmo y tener un análisis más sencillo. Por último, en el segundo algoritmo surgió un problema puesto que los valores de la serie Fibonacci, ya que encontrar una manera de utilizarlos para hacer el análisis a posteriori fue un poco confuso pero la solución fue crear una función que guarde los valores de la sucesión en un arreglo y posteriormente enviarlos a la función Euclides para encontrar el máximo común divisor.

Conclusiones individuales (Brayan Ramirez Benitez): Podemos considerar el diseño y análisis de algoritmos como una especie de arte, que nos ayuda en gran medida a mejorar nuestra inteligencia lógico matemática. Resulta bastante interesante encontrar tanto el mejor como el peor caso incluso estos pueden ser iguales para un algoritmo. La metodología y las definiciones mostradas en clase y una parte en esta práctica nos permiten comprender los elementos de un algoritmo y realizar un análisis para determinar su complejidad, de esta forma ser capaces de identificar cuáles son los más eficaces para resolver cualquier problema bien planteado.



Brayan Ramirez Benitez.

5 Bibliografía

References

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

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

[3] 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