Ingeniería en Inteligencia Artificial, Análisis y Diseño de Algoritmos
Sem: 2024-1, 3BV1, Práctica 2, 27 de septimebre de 2023

Práctica 2: Complejidades temporales polinomiales y no polinomiales.

Catonga Tecla Daniel Isaí 1, Olguin Castillo Victor Manuel 2.

*daniel*9513*importantes*@*gmail.com*$_1$, *manuelevansipn*@*gmail.com*$_2$

ResumenSe propusieron dos ejercicios, los cuales se programaron en el lenguaje C++ para hacer un análisis a priori del algoritmo iterativo de la sucesión de Fibonacci y el análisis a posterior del algoritmo recursivo de la sucesión de Fibonacci y el algoritmo del cálculo de números perfectos, para así determinar su complejidad algorítmica, parael análisis a priori se ocupó el conteo de proceso para determinar una ecuación que represente su complejidad, para hacer análisis a posteriori ocupamos un método que consiste en usar contadores que aumenta en 1 cada que se realiza una operación en nuestros algoritmos, para así poder comparar la tasa de crecimiento de las operaciones realizadas en un peor caso y en un mejor caso. Estos datos obtenidos se graficaron con la página web DESMOS para poder ver de manera gráfica el comportamiento del algoritmo.

Palabras Clave: Algoritmo, C++, big Θ, Complejidad Temporal

1. Introducción

Problemas como la serie de Fibonacci, que es una secuencia infinita de la suma de los dos números anteriores[**fibo**], o el problema de los números perfectos, que es la suma de los divisores de un número igual a ese mismo número entonces es perfecto[**key**], pueden resolverse parcialmente mediante el uso de algoritmos.

Los algoritmos son de suma importancia ya que los algoritmos ayudan a la resolución de problemas muy grandes o complejos como problemas matemáticos y científicos, por lo tanto, los algoritmos son fundamentales en varias áreas de las ciencias o ingeniería. En la actualidad existen muchos algoritmos para resolver problemas como lo son algoritmos de ordenamiento, búsqueda, entre muchos otros tipos de algoritmos.

El análisis de los algoritmos es otra parte fundamental por lo que los algoritmos tienen que ser lo más eficientes posibles en complejidad temporal y complejidad espacial. En cuestión de complejidad temporal algunos algoritmos pueden requerir años en resolver un solo problema, por lo que, no es eficiente para la práctica en una empresa, y se opta por otros algoritmos que resuelvan el problema que presenten de una forma más eficaz. Por otra parte, una maquina no cuenta con recursos infinitos de memoria o espacio de almacenamiento, por lo que, existen algoritmos que ocupan mucho espacio de memoria y esto puede ocasionar problemas para equipos que no cuentan con el espacio suficiente, entonces se tiene que analizar también lo que es la complejidad espacial para tratar de ocupar la menor cantidad de memoria posible. Ambas complejidades son importantes, pero es más importante la complejidad temporal ya que las empresas pueden comprar más memorias de almacenamiento, pero no pueden comprar tiempo, por lo que algunas empresas optan sacrificar complejidad espacial por una complejidad temporal más eficiente.

Los algoritmos de complejidad polinomial crecen de manera suave graficamente, mientras que los algortitmos de complejidad no polinomial crece de manera drástica, lo cual no favorece a la eficiencia de los algoritmos.

2. Conceptos Basicos

- **Algoritmo**. Un algoritmo es una secuencia de pasos lógicos que son precisos, ordenados y finitos que se ocupan para resolver un problema deseado[**concep1**].

- **Análisis de algoritmos** Es un proceso de evaluación donde conoceremos el rendimiento y la eficiencia de un algoritmo. Se evaluará el consumo de tiempo y de recursos computacionales que requiere el algoritmo para ser ejecutado con diversos datos de entrada, y esto determinará su complejidad[**concep1**].

- **Cota superior asintótica**. Es una función que delimita por la parte superior a otra función a medida que la entrada de la función delimitada crece.

- **Cota inferior asintótica**. Es una función que delimita por la parte inferior a otra función a medida que la entrada de la función delimitada crece.

- **Notacion O**. Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota superior asintótica en el peor caso de ejecución de un algoritmo[**concep1**].

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\ c \text{ constante positiva y } n_0 \notin \mathbb{N} :$$
$$f(n) \leq cg(n),\ \forall\ n \geq n_0\}$$

- **Notacion Tetha "Θ"** Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota superior y una cota inferior, esto nos da una idea más precisa del comportamiento y complejidad del mismo[**concep1**].

$$\Theta(g(n) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c_1, c_2 \text{ constantes positivas}, n_0\ :$$
$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall\, n \geq n_0\}$$

- **Notacion Omega "Ω"** Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota inferior asintótica en el peor caso de ejecución de un algoritmo[**concep1**].

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c \text{ constante positiva y } n_0\ :$$
$$0 < cg(n) \leq f(n), \forall\, n \geq n_0\}$$

- **Analisis a posteriori**. Es una evaluación que se realiza de forma empírica, donde los resultados se obtienen con la ejecución del algoritmo y la medición del tiempo y recursos computacionales que requiere[**concep1**].
- **Analisis a priori**. Es una evaluacion que se realiza de forma teorica, donde los resultados se pueden obtener con el conteo de operaciones y/o analisis matematico que en base a sus formulas se obtiene su complegidad algoritmica[**concep1**].
- **Sucesion de Fibonacci**. Es una secuencia matematica infinita que incia con los numeros 0 y 1, y el numero siguiente sera la suma de los dos numeros anteriores a este[**fibo**].

El tercer número es $0 + 1 = 1$.
El cuarto número es $1 + 1 = 2$.
El quinto número es $1 + 2 = 3$.

- **Numero perfecto**.Se le considera numero perfecto a aquel que la suma de sus divisores propios positivos da por por resultado el mismo numero[**key**].

Los divisores positivos de 28 son 1, 2, 4, 7 y 14.
Si sumamos estos divisores: $1 + 2 + 4 + 7 + 14 = 28$.

- **Funciones recursivas**. En la programacion las funciones recursivas son aquellas que durante su proceso se invocan asi mismas.

```
Suma_Recursiva(n):
If n == 1
    return 1
Else
    return n + sumaRecursiva(n - 1)
```

- **Funciones iterativas**. En la programacion las funciones iterativas son aquellas se ejecuta en un ciclo n numero de veces y existe una condicion de validacion en cada iteracion que controla si se itera una ves mas o finaliza en ciclo

```
Suma_iterativa(n):
resultado = 0
for i = 0 to n do:
    resultado += i
return resultado
```

- **Pseudocodigo Sucesion de Fibonacci version iterativa**

```
fibonacci_iterativa(num_anterior, num_actual, n)
for i = 0 to n do
    print num_actual
    aux = num_actual;
    num_actual += num_anterior;
    num_anterior = aux;
```

- **Pseudocodigo Sucesion de Fibonacci version recursiva**

```
fibonacci_recursiva(num_anterior, num_actual, n)
    if n == 0
        return
    else
        print num_actual
        fibonacci_recursiva(num_actual, num_actual + num_anterior, n-1);
```

- **Pseudocódigo Perfecto**. El algoritmo de Perfecto dice si el número n es perfecto o no, haciendo la suma de sus divisores con un for, retorna un 0 si el número no es perfecto y retorna un 1 si es perfecto.

```
Perfecto(n):
contador = 0
for i = 1 to i = n-1 do
  if n%i == 0
    contador = contador + i
if contador == n
  retorna 1
retorna 0
```

- **Pseudocódigo MostrarPerfectos**. El algoritmo de MostrarPerfectos, muestra n números perfectos, para cuando $n = 2$ muestra dos números perfectos tal que el output es: $[6, 28]$. Utiliza la función Perfecto para calcular los números perfectos.

```
MostrarPerfectos(n):
contador = 0
for i = 1 to contador != n do
  if Perfecto(i) do
    imprime i
    contador = contador + i
```

3. Experimentación y Resultados

Ejercicio 2

Una vez escritas las funciones (iterativo y recursivo) de la sucesión de fibonacci en C++, se procedió a hacer el análisis a priori de la función iterativa, por lo que se enumeraron el número de procesos que contiene la función (Imagen.1) y se creó la (tabla.1) donde se colocó el costo de cada proceso y el número de pasos ejecutados.

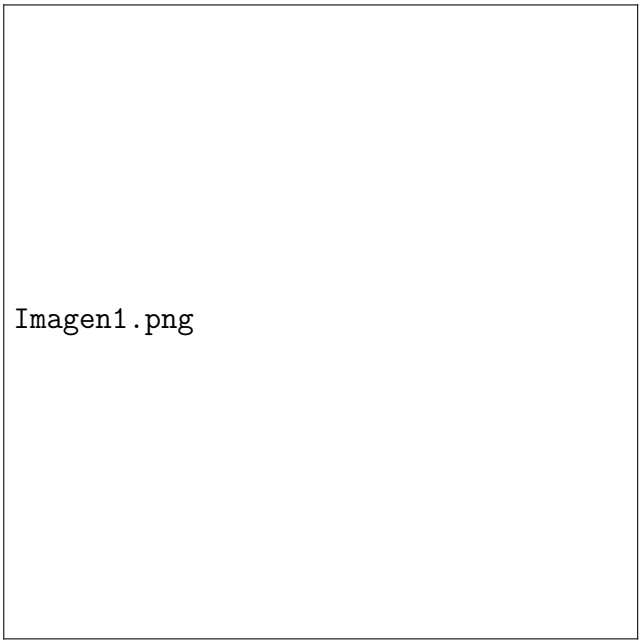


Imagen 1:Conteo de procesos

Indice Fibonacci	# Pasos ejecutados
C1	n+1
C2	n
C3	n
C4	n

Cuadro 1: costo vs pasos

1) Se ejecuta n+1 veces, ya que el ciclo recorrerá de 0 hasta n-1 (n ejecuciones) y se ejecuta una última vez donde ya no se cumple la condición, lo que significa que se ejecuta n+1 veces.

2) Se ejecuta n veces, ya que este proceso se encuentra dentro del ciclo for que itera desde 0 hasta n-1 (n ejecuciones)

3) Se ejecuta n veces, ya que este proceso se encuentra dentro del ciclo for que itera desde 0 hasta n-1 (n ejecuciones)

4) Se ejecuta n veces, ya que este proceso se encuentra dentro del ciclo for que itera desde 0 hasta n-1 (n ejecuciones)

Se sabe que para calcular la complejidad algorítmica usando el análisis a priori se multiplica el costo computación de cada proceso por el número de pasos ejecutados, por lo que tendríamos la (ecuación.1), lo que nos indica que este algoritmo tiene una complejidad lineal

$$\begin{gathered}
T(n) = C1(n + 1) + C2n + C3n + C4n \\
T(n) = C1 + C1n + C2n + C3n + C4n \\
T(n) = (C1 + C2 + C3 + C4)n + C1 \\
a = (C1 + C2 + C3 + C4)\ ,\ b = C1 \\
T(n) = an + b \\
\therefore T(n) \in \theta\ (n)
\end{gathered}$$

Ahora, ya que conocemos la complejidad algorítmica con el análisis a priori, se verá cómo se comporta el análisis a posteriori, para eso se colocan contadores en cada proceso de nuestra función en C++ y se contará cuántos procesos se ejecutan en comparación a qué índice de la sucesión de Fibonacci se calcula, el programa mostró la salida mostrada en la (Imagen.2) y los resultados del conteo se muestran en la (tabla.2) y la gráfica de estos datos la podemos ver en (gráfica.1), en ambos análisis concuerda su complejidad, por lo que podemos decir que la versión iterativa de la sucesión de Fibonacci tiene complejidad lineal.

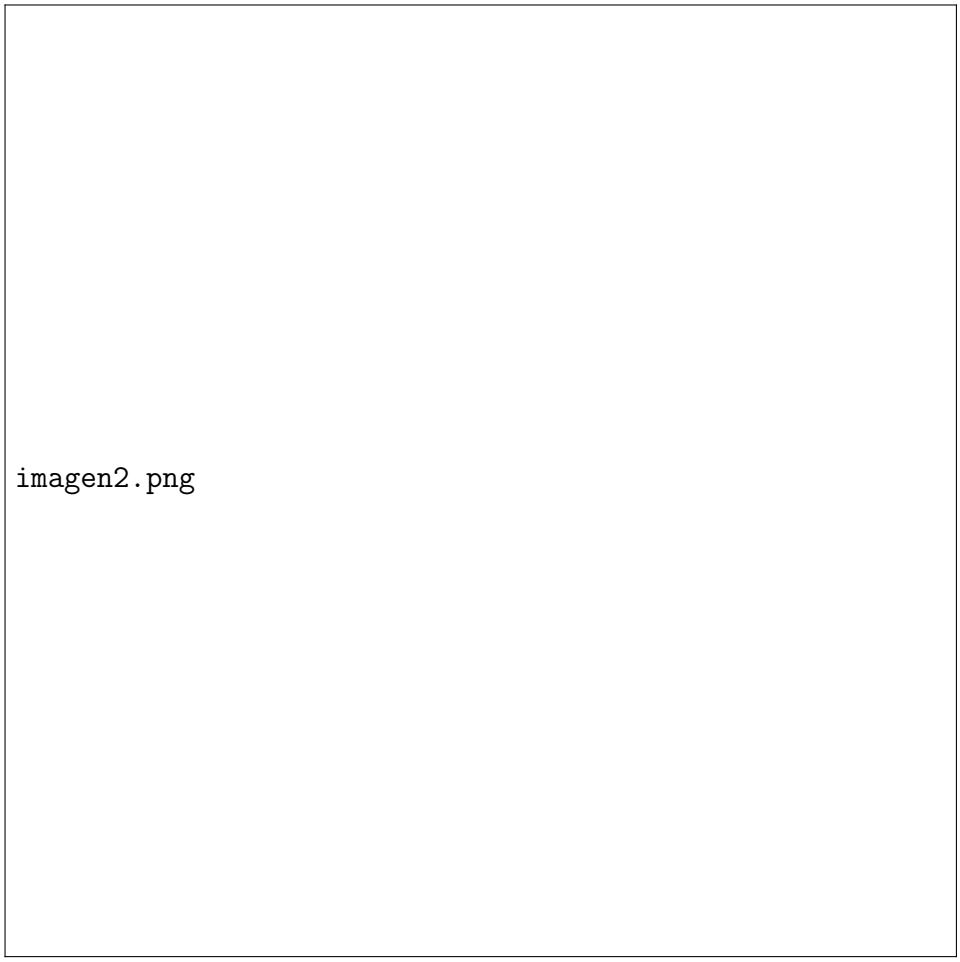


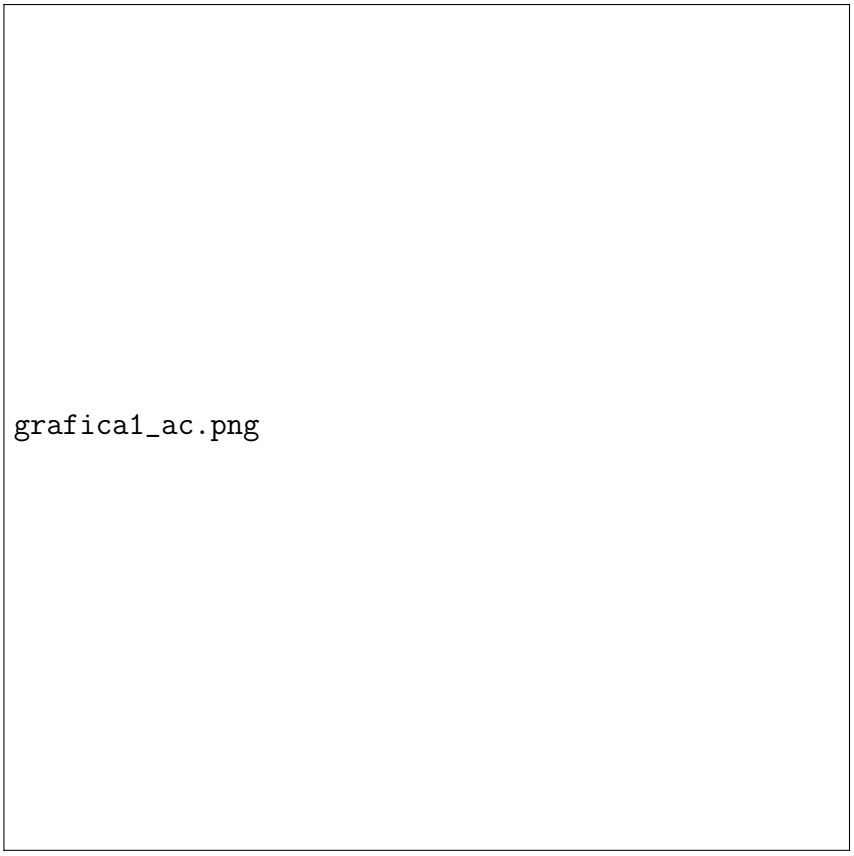
Imagen 2: Resultados de la ejecucion del algoritmo iterativo

Indice Fibonacci	**# Pasos ejecutados**
1	5
2	9
3	13
4	17
5	21
6	25
7	29
8	33
9	37
10	41
11	45
12	49
13	53
14	57
15	61
16	65
17	69
18	73
19	77
20	81
⋮	⋮

Cuadro 2: Analisis a priori algoritmo iterativo

grafica1.png

Una funcion que cree una cota superior es $g(n) = 4{,}5x$ y y una cota inferior es $z(n) = 3{,}5x$, y se muestra en la (grafica.2)



Grafica.2 (Indice vs # procesos)

El siguiente paso fue hacer el análisis a posteriri de la versión recursiva de la sucesión de Fibonacci, para esto se encontró dos maneras diferentes de realizar la función. La primera opción es sin memorización, en la que se realizan los mismos cálculos una y otra vez cada vez que se calcule un nuevo número de la sucesión.La segunda opción es con memorización en la que se almacenan en memoria los resultados calculados anteriormente para evitar recalcularlos, esto reduce demasiado la cantidad de cálculos.

Primero se hizo el análisis a posteriori de la función recursiva sin memoria, por lo que colocamos contadores en cada línea de código y se contará cuántos procesos se ejecutan en comparación a qué índice de la sucesión de fibonacci se calcula, el programa mostró la salida mostrada en la (imagen.3) y los resultados se muestran en la (tabla.3) y la gráfica de estos datos la podemos ver en (gráfica.3)

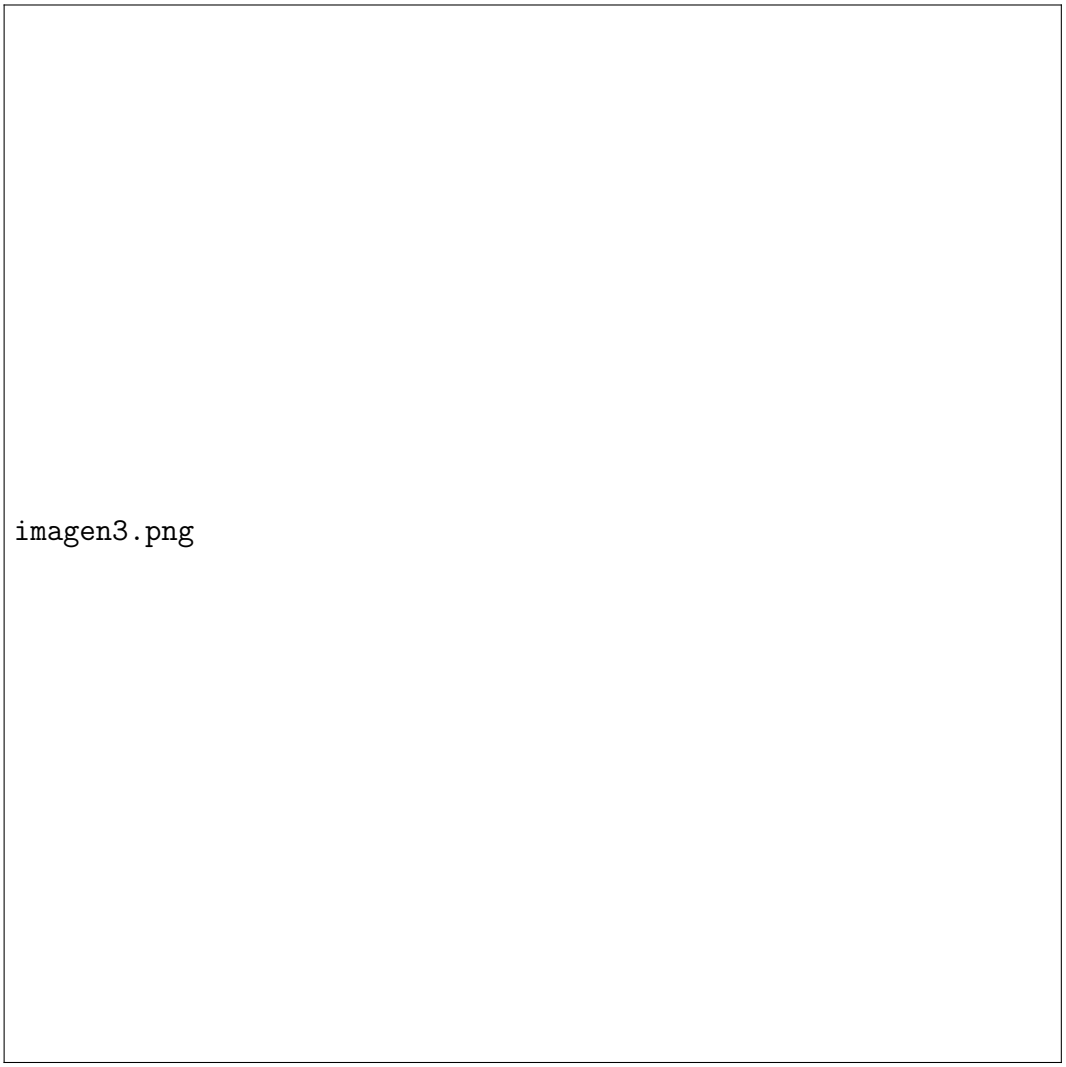


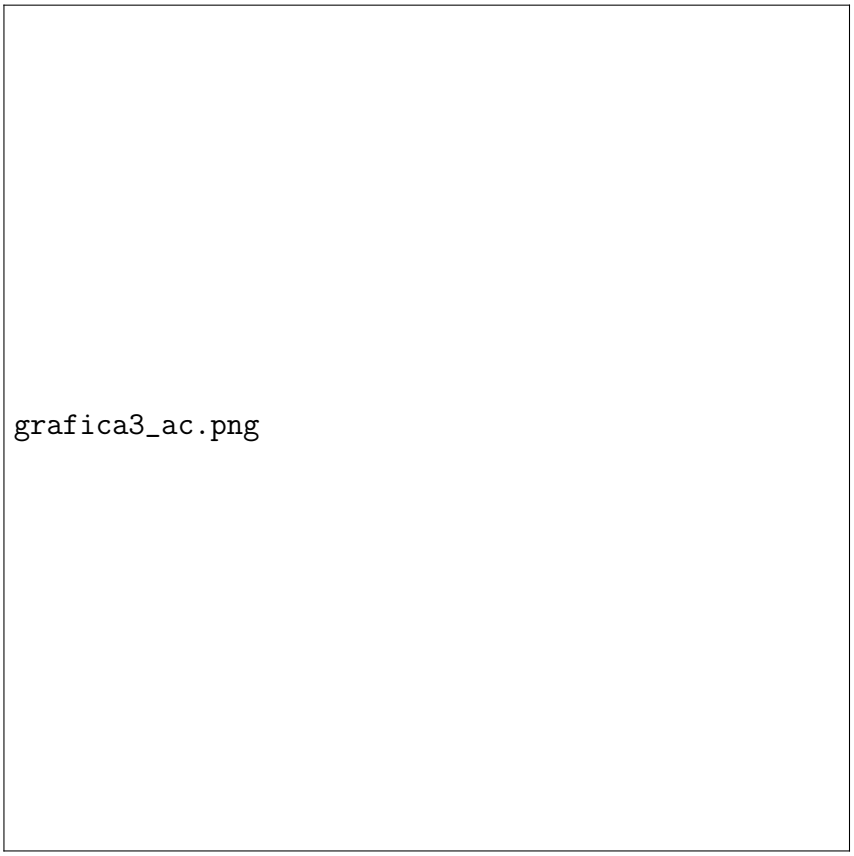
Imagen 3: Ejecucion del programa recursivo sin memoria

Indice Fibonacci	**# Pasos ejecutados**
1	2
2	2
3	6
4	10
5	18
6	30
7	50
8	82
9	134
10	218
11	354
12	574
13	930
14	1506
15	2438
⋮	⋮

Cuadro 3: Analisis a priori algoritmo recursivo sin memoria

Grafica.3 (Indice vs # procesos)

Una funcion que cree una cota superior es $g(n) = 2^{,73n}$ y y una cota inferior es $g(n) = 2^{,7n}$, y se muestra en la (grafica.4)



Grafica.4 (Indice vs # procesos)

Al ejecutar el programa e insertar un n>50 tardaba demasiado debido a la cantidad de procesos, lo que ocasiona una gran cantidad de cálculos redundantes y un tiempo de ejecución exponencial a medida que n aumenta, por lo que este método es demasiado ineficiente y provoca un desbordamiento de memoria con n demaciados grandes.

Después se procedió a hacer el análisis a posteriori de la función recursiva con memoria, por lo que colocamos contadores en cada línea de código y se contará cuántos procesos se ejecutan en comparación a qué índice de la sucesión de fibonacci se calcula, el programa mostró la salida mostrada en la (imagen.4) y los resultados se muestran en la (tabla.4) y la gráfica de estos datos la podemos ver en (gráfica.5)

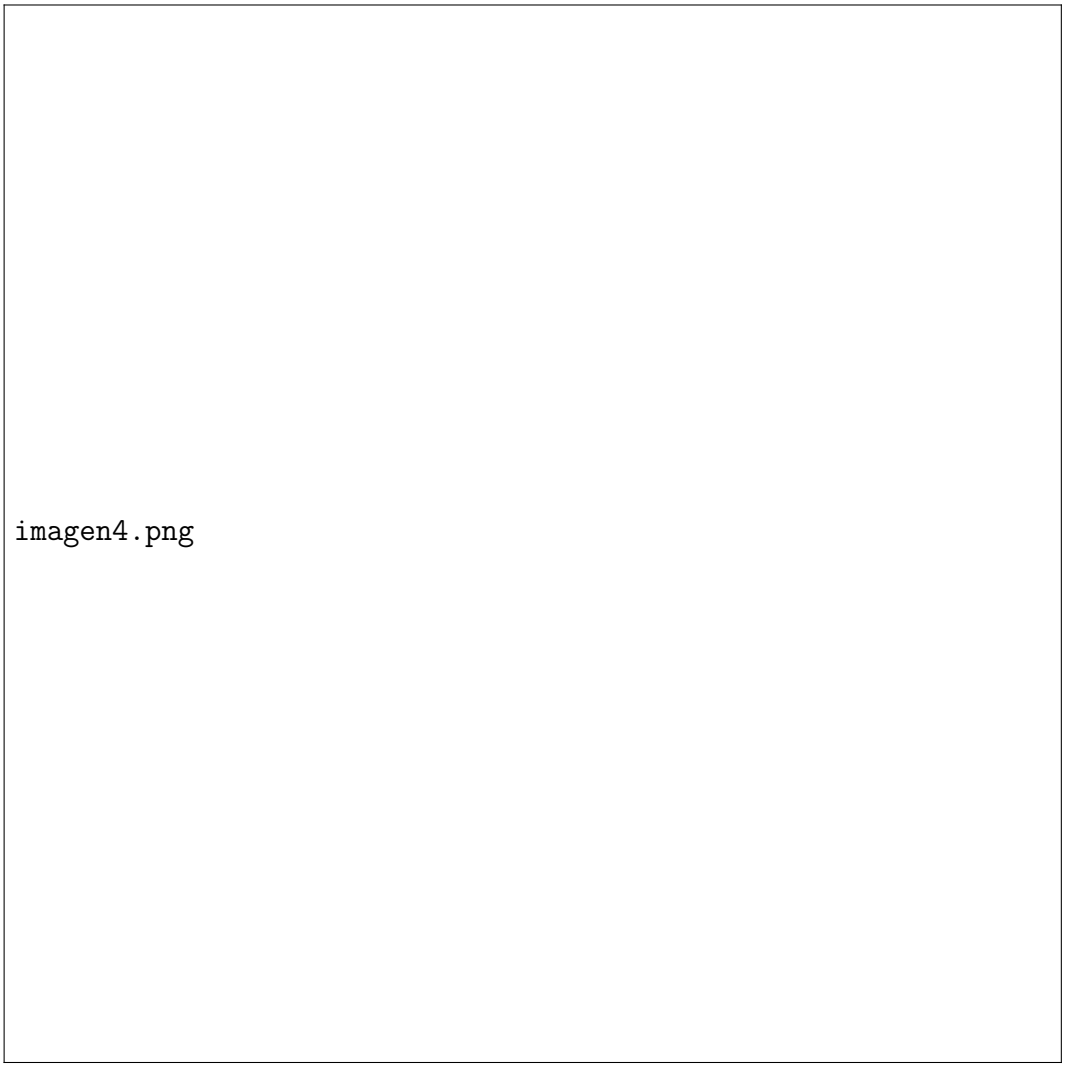


Imagen 4: Ejecucion del programa recursivo con memoria

Indice Fibonacci	**# Pasos ejecutados**
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	16
9	18
10	20
11	22
12	24
13	26
14	28
15	30
⋮	⋮

Cuadro 4: Analisis a priori algoritmo recursivo con memoria

grafica3.png

Grafica.5 (Indice vs # procesos)

Una funcion que cree una cota superior es $g(n) = 2^{,73n}$ y y una cota inferior es $g(n) = 2^{,7n}$, y se muestra en la (grafica.7)

Grafica. 7 (Indice vs # procesos)

Se observó que la función recursiva con memoria se comporta de manera lineal, por lo que esto resulta más eficiente en comparación a su versión sin memoria.

Algoritmo 2. Perfecto y MostrarPerfectos

Análisis a priori Perfecto. El análisis a priori del algoritmo 2 se basa con la tabla n que se muestra. La tabla 5 muestra el costo de número de pasos de cada instrucción del pseudocódigo el cual es calculado.

Algoritmo 2: Perfecto

```
Perfecto(n)
 1. contador = 0
 2. for i=1 to i=n-1 do
 3.    if n %i == 0 do
 4.        contador = contador + i
 5. if contado == n do
 6.    Retorna 1
 7. Retorna 0
```

Costo	**# de pasos**
C_1	1
C_2	n
C_3	$n-1$
C_4	1
C_5	1
C_6	1
C_7	1

Tabla 5. Tabla de análisis a priori.

El tiempo de ejecución es igual a la suma de los productos de costo del número de pasos ejecutados línea por línea. Por lo que:

$$T(n) = C_1 + C_2n + C_3(n-1) + C_4 + C_5 + C_6 + C_7$$
$$T(n) = C_2n + C_3n + C_1 - C_3 + C_4 + C_5 + C_6 + C_7$$
$$T(n) = n(C_2 + C_3) + (C_1 - C_3 + C_4 + C_5 + C_6 + C_7)$$

Para el algoritmo de Perfecto no hay peor caso o mejor caso, si el número es mayor entonces su crecimiento es lineal siempre, por lo que, no tiene un aumento cuadrático o de otra forma que no sea lineal. Por lo tanto, el agoritmo es el mismo en el peor y mejor caso, se denota como:

$$T(n) = n(C_2 + C_3) + (C_1 - C_3 + C_4 + C_5 + C_6 + C_7)$$
$$T(n) = an + b \text{ con } a = C_2 + C_3 \text{ y } \ b = C_1 - C_3 + C_4 + C_5 + C_6 + C_7$$
$$\therefore T(n) \in \Theta(n)$$

Análisis a posteriori Perfecto. El algoritmo en ejecución se muestra en la imagen 5 y el número de pasos se muestra en la tabla 6, el número de pasos es con respecto a la ejecución del algoritmo Perfecto. Los datos muestran un aumento lineal en la complejidad a medida que n crece. Se muestran variaciones en ciertos casos, pero no son variaciones significativas que sugieran un crecimiento cuadrático u otro tipo de complejidad.

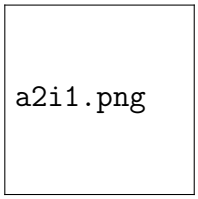
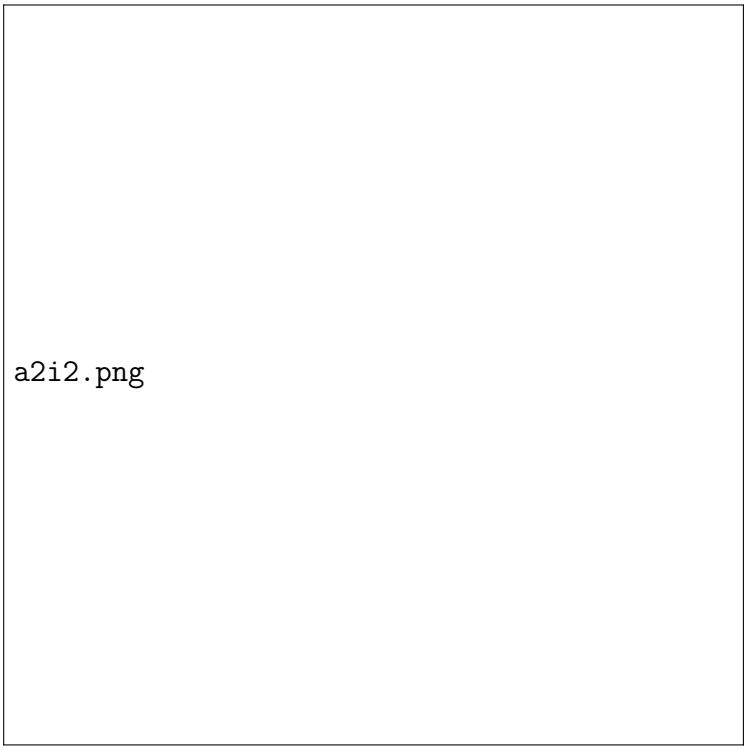


Imagen 5. Ejecución algoritmo Perfecto.

Valor de n	**# de pasos**
2	9
3	12
4	16
5	18
6	23
7	24
8	29
9	31
10	35
11	36
12	43
13	42
14	47
15	50
⋮	⋮

Tabla 6. Datos de algoritmo Perfecto.

Los datos de ejecución del algoritmo Perfecto se muestran en una gráfica en la gráfica 6. El algoritmo muestra un crecimiento lineal demostrando que para peor y mejor caso es el mismo, la gráfica es lineal por lo que esto se denota como $\Theta(n)$.



Gráfica 6. Grafica del comportamiento del algoritmo Perfecto.

En la gráfica 7 se muestra la función tal que $g(n) = \frac{10}{3}n$ acota por arriba al algoritmo y $z(n) = \frac{8}{3}n$ acota por abajo al algoritmo, y el ajuste asíntotico es para cuando $n_0 \geq 300$. En la gráfica 8 se muestra que $f(n)$ esta delimitado por $g(n)$ y $z(n)$ cuando $n \geq n_0$.

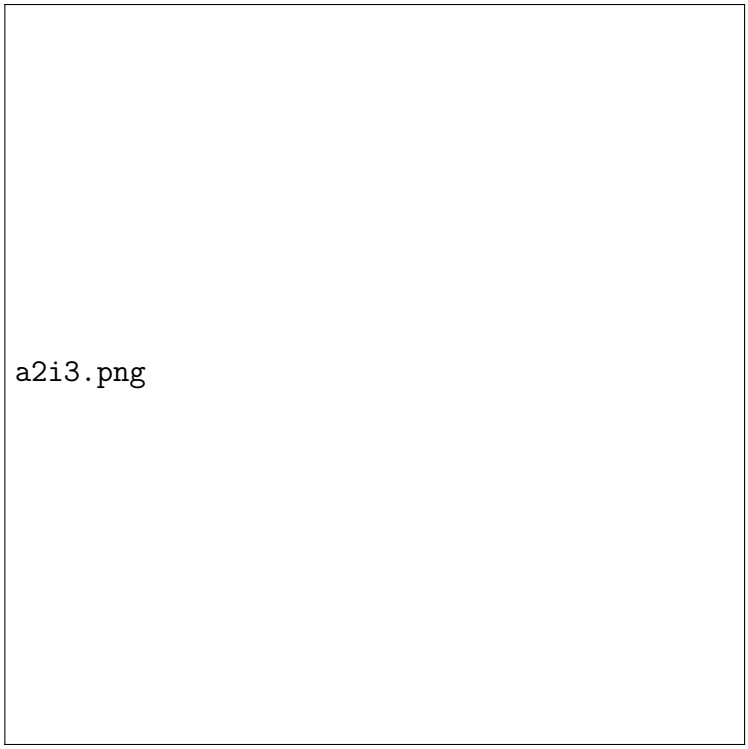


Figura 7. Gráfica con acotación para $f(n)$.

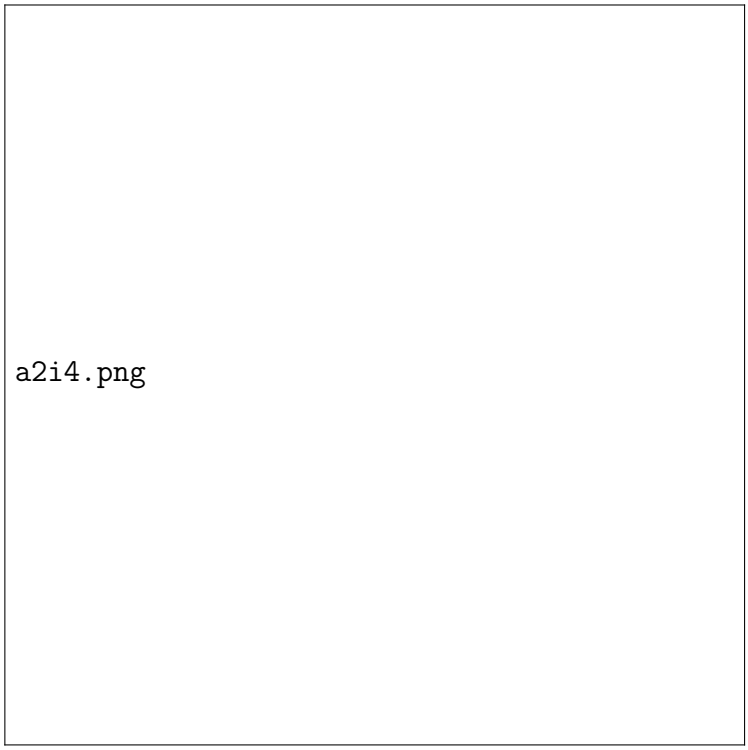


Figura 8. Gráfica para valor n_0.

Análisis a posteriori MostrarPerfectos. La ejecución del algoritmo 'MostrarPerfectos' se muestra en la imagen 6, con una entrada tal que $n = 5$. La particularidad del algoritmo es que no puede calcular el quinto número perfecto, el número de pasos para valores de n desde 1 hasta 4 se muestran en la tabla 7. Sin embargo, para valores de n mayores o iguales a 5, no se muestran en la tabla porque el algoritmo no lo puede calcular.

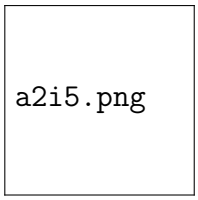


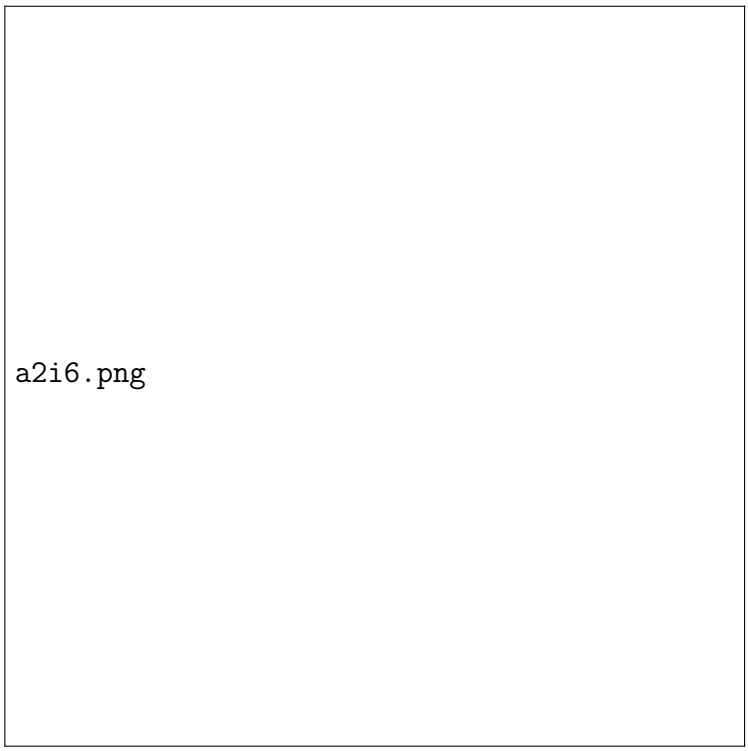
Imagen 6. Ejecución algoritmo 'MostrarPerefectos'.

Valor de n	**# de pasos**
1	98
2	1430
3	374917
4	992155729
5	sin valor
6	sin valor
⋮	⋮

Tabla 7. Datos de algoritmo 'MostrarPerfecto'.

La gráfica que genera los puntos de la tabla de n desde 1 hasta 4 se muestra en la gráfica 9. Dado que los valores son muy distintos entre sí, la escala de la gráfica no deja ver el comportamiento del algoritmo. Por lo tanto, en la gráfica

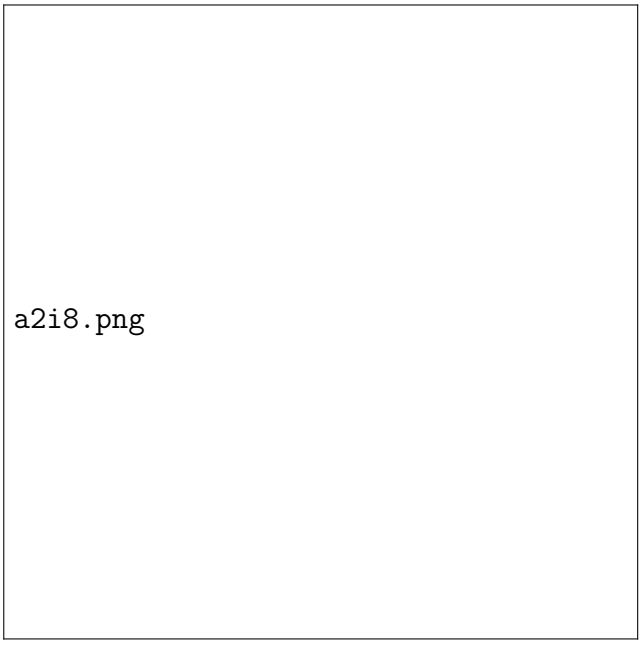
10 se muestran únicamente los dos primeros puntos, donde ya se puede observar el comportamiento de la gráfica y se muestra que aumenta significativamente del punto 2 con respecto del punto 1.



Gráfica 9. Gráfica del comportamiento de 'MostrarPerfectos'.

Gráfica 10. Gráfica de los primeros dos puntos de 'MostrarPerfectos'.

La función tal que $g(n) = 2^{7n}$ acota la gráfica por arriba y $z(n) = 2^{5n}$ acota por abajo a los dos primeros puntos se muestra en la gráfica 11, y también a los otros dos puntos restantes como se muestra en la gráfica 12. El ajuste asíntotico es para cuando $n_0 \geq 1$, se muestra en la gráfica 13 que $f(n)$ esta delimitado por $g(n)$ y $z(n)$ cuando $n \geq n_0$.



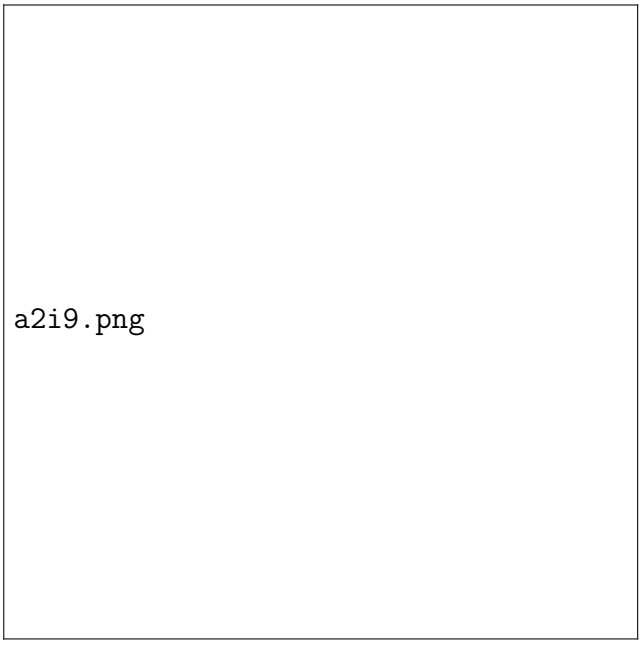
Gráfica 11. Gráfica primeros dos puntos con acotación para $f(n)$.

4. Conclusiones

El algoritmo de 'Perfecto' y 'Fibonacci' en su forma iterativa se pudieron resolver de forma eficiente, ambos algoritmos son big $\Theta(n)$, ya que ambos son lineales. La particularidad de los otros algoritmos es que el algoritmo 'Fibonacci' en su forma recursiva tienen un crecimiento exponencial al igual que el algoritmo de 'MostrarPerfectos', estos algoritmos tienen un crecimiento muy drástico a medida que n crece por lo que los algoritmos no son eficientes en lo práctico, ya que un algoritmo tiene que ser lo más eficiente posible, ambos algoritmos son $\Theta(2^n)$.

Conclusiones Catonga Tecla Daniel Isaí 1

Con los algoritmos que se desarrollaron en la práctica aprendí porque los algoritmos exponenciales no son eficientes para el uso de estos, por lo que, siempre se tiene que buscar la mejor forma en la que sea eficiente un algoritmo porque sino el algoritmo no podrá realizar el algoritmo para cuando n sea un número muy grande, en el caso de los algoritmos exponenciales. Por otra parte aprendí que los algoritmos no siempre resulven el problema que se tiene, es decir, se puede tener un problema y crear un algoritmo pero por limitaciones de una computadora, este podrá solucionar el problema parcialmente, es decir que solo obtendrá el resultado para un rango muy pequeño de lo que abarca el problema, por ejemplo el problema de mostrar n perfectos, se observó que el algoritmo no puede calcular el quinto número perfecto. En está práctica se

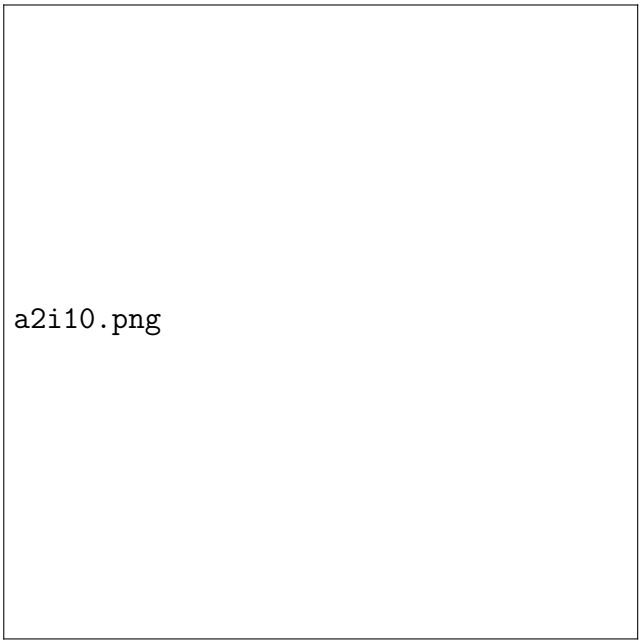


Gráfica 12. Gráfica dos puntos restantes con acotación para $f(n)$.

aprendieron cosas importantes sobre el análisis de algoritmos.

Conclusiones Olguin Castillo Victor Manuel 2

Al terminar esta practica reforce mis conociminetos sobre funciones recursivas, ya que sabia lo que eran y como funcionaban, pero muy pocas veces las habia implementado en un programa, ahora puedo saber como convertir una funcion iterativa en una recursiva; tambien aprendi que una funcion recursiva no es sinonimo de eficiencia el concepto de çon/sin memoriaïnfluye en la eficicnecia de este.



Gráfica 13. Gráfica para valor n_0.

5. Bibliografía