Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2022-2, 3CV11, Práctica 2, 14 de marzo de 2022

Práctica 2: Complejidades temporales polinomiales y no polinomiales

Luis Francisco Renteria Cedillo, Denzel Omar Vazquez Perez.

*lrenteriac*1400@*alumno.ipn.mx*, *dvazquezp*1600@*alumno.ipn.mx*

Resumen: En la presente documento se muestra el comportamiento de algoritmos iterativos contra los recursivos evaluando cual de estos es el más eficiente y determinando por su complejidad algorítmica ¿Cúal es le mejor a implementar?, por otra parte se habla de números perfectos y sus implementación para sus posterior evaluación ante un equipo de cómputo convencional.

Palabras Clave: Fibonacci, Numeros Perfectos, Recursividad, Complejidad, C++ .

1 Introducción

Hoy en día la recursión e iteratividad son usadas en tareas repetitivas hasta que se cumpla alguna condición, sin embargo al momento de codificar se ignora cual de estos estilos de programación es la conveniente para llegar al objetivo en el menor tiempo posible.

Es importante mencionar que se tiene la respuesta ante tal pregunta y es por medio de un análisis, los algoritmos tanto iterativos como recursivos deben de pasar por un análisis a **priori** tanto a **posteriori** donde en uno se busca el calculo de la complejidad algorítmica por medio de conceptos básicos aprendidos en la Unidad de aprendizaje, y el otro permite la colección de estadísticas del tiempo consumidos mientras se ejecuta el algoritmo, con el fin común de encontrar una función que acote el tiempo de ejecución de la tarea a realizar.

Para esta práctica se pondrá en comparación el tiempo de ejecución del algoritmo de la sucesión de Fibonacci tanto en su representación iterativa como recursiva con el fin de evaluar cual de estas implementaciones resulta más eficiente que la otra, como segunda parte del documento dará búsqueda de números perfectos y como la codficación y ejecución de este algoritmo puede

perjudicar directamente el rendimiento del equipo de cómputo con el que se trabaja.

2 Conceptos teóricos

2.1 Sucesión de Fibonacci

Los números de la sucesión de Fibonacci fueron definidos originalmente en el siglo XIII por el matemático italiano Fibonacci para modelar el crecimiento de las grandes multitudes de conejos, definió la relación de recurrencia como:

$$X_n \ = \ X_{n-1} \ + \ X_{n-2}$$

Los principales casos son $X_0 = 0$ y $X_1 = 1$, por tanto $X_2 = 1$, $X_3 = 2$ donde al continuar con la sucesión se tendrá $\{3, 5, 8, 13, 21, 34, 55, 89, 144, \cdots\}$, esta formula a la no ser buena para contar a la gran población de conejos, resulto tener una serie de propiedades, de las cuales tiene una estrecha relación con la sucesión de Lucas y el número áureo (φ) cuyo valor númerico es:

$$\varphi \ = \ \frac{1+\sqrt{5}}{2}$$

Es posible aproximarse a este ultimo al dividir dos términos consecutivos de la sucesión de Fibonacci, donde en cuanto más grandes son los términos, el resultado de los cocientes se acerca más a φ .
La solución de este problema es posible por medio de su implementación iterativa así como recursiva, cuyos pseudo-códigos son:
Algoritmo iterativo de la sucesión de Fibonacci

Algorithm 1 Sucecion_Fibonacci (*int n*)

Input: n
Output: x2

```
x1 = 1
x2 = 0
for i=1 to i≤ n do
    x2 = x1 + x2 ;
    x1 = x2 − x1 ;
end for
return x2
```

Algoritmo recursivo de la sucesión de Fibonacci

Algorithm 2 Sucecion_Fibonacci_R (*int n*)

Input: n
Output: Número natural
1: **if** n<2 **then**
2: **return** n
3: **else**
4: **return** (Sucecion_Fibonacci_R $(n-1)$ + Sucecion_Fibonacci_R$(n-2)$
5: **end if**

2.2 Números perfectos

Un número perfecto es un número que pertenece al dominio de los números enteros y cumple la condición que es igual a la suma de sus divisores. Como ejemplo, tenemos que el numero 6 tiene divisores 1, 2 y 3. Dado que la suma de 1+2+3=6, se cumple que 6 es un número perfecto.

En la antigüedad, los números perfectos eran considerados superiores a los demás y se les atribuían propiedades místicas. A pesar de que no se les ha encontrado utilidad para resolver problemas matemáticos o áreas como la criptografía, siguen siendo un misterio para los matemáticos de la actualidad.

Existen tres conjeturas de los números perfectos:

1. Todos los números perfectos son pares ya que tienen potencia de 2 como factor.
2. Todos los números perfectos finalizan en 6 o 8
3. Los números perfectos son infinitos

No obstante, ninguno de los enunciados anteriores ha sido demostrado.
Euclides propuso una formula para obtener los números perfectos:

$$(2^{n-1}) \cdot (2n-1) = Numero\,Perfecto$$

Donde n y $(2n-1)$ deben ser ambos primos. Más adelante, el matemático Euler realizó la demostración que todos los números perfectos se generan por la fórmula propuesta por Euclides.

2.2.1 Test de Primalidad de Fermat

La prueba o test de primalidad de Fermat es una prueba que utiliza el pequeño teorema de Fermat y establece que si un numero primo p y un numero coprimo a con p , entonces a^{p-1} es divisible por p. Se describe de la siguiente forma:

$$a^{p-1} \equiv 1 (mod\, p)$$

Sin embargo, existe la posibilidad de que el resultado sea un falso positivo, dado a que es un algoritmo probabilístico. Es por ello que debe ejecutarse determinado numero de veces para reducir la probabilidad de error.

Algorithm 3 testPrimalidadFermat($int\ n, int\ iteraciones$)

Input: n,iteraciones
Output: Primo si n es un posible primo, caso contrario regresa Compuesto

```
1: for i=1 to i≤ iteraciones do
2:     a = random_generator(1, n − 1)
3:     if a^{n−1} ≢ 1(mod n) then
4:         return Compuesto
5:     end if
6: end for
7: return Primo
```

3 Experimentación y resultados

3.1 Algoritmos para la sucesión de Fibonacci

Este primer problema tiene dos secciones donde se encuentra tanto su solución a partir de su relación iterativa como por su relación de recurrencia, presentando un análisis detallado para determinar su complejidad algorítmica en base a su codificación en lenguaje C.

3.1.1 Implementación iterativa de la Sucesión de Fibonacci

Para la implementación iterativa de los números de Fibonacci se hace uso del **Algorithm 1** visto en la sección de *Conocimientos teóricos*.

3.1.1.1 Análisis a priori

Para conocer la complejidad temporal que tiene el algoritmo sobre el espacio del tiempo, es fundamental hacer el calculo de este por medio de conceptos teóricos para tener como resultado una función que acote el tiempo de ejcución la solución al problema planteado. En seguida se muestra en la Tabla 1 el análisis linea por linea del algoritmo del la sucesión de Fibonacci con el fin de mostrar el comportamiento del mejor y peor caso.

Tabla 1: Análisis a priori linea por linea para el mejor y peor caso

No.	**Código**	**Costo**	**# Pasos ejecutables**	
	Sucesion_fibonacci (int n)		**Mejor caso**	**Peor caso**
1	x1 = 1;	C_1	1	1
2	x2 = 0;	C_2	1	1
3	for (i=1; i≤ n; i++){	C_3	n	n
4	x2 = x1 + x2;	C_4	n-1	n-1
5	x1 = x2 - x1; }	C_5	n-1	n-1
6	return x2;	C_6	1	1

Dado al número de pasos ejecutables para el mejor y peor caso de la Tabla 1, se identifica que no existe una diferencia para cualquiera de estos, puesto, que su crecimiento asintótico es igual. Así teniendo la información de la anterior Tabla se forma la siguiente ecuación para conocer el orden del complejidad del algoritmo expuesto:

$$T(n) = C_1\,(1) + C_2\,(1) + C_3\,(n) + C_4\,(n-1) + C_5\,(n-1) + C_6\,(1)$$
$$T(n) = C_1 + C_2 + nC_3 + (n-1)C_4 + (n-1)C_5 + C_6$$
$$T(n) = C_1 + C_2 + nC_3 + nC_4 + C_4 + nC_5 + C_5 + C_6$$
$$T(n) = nC_3 + nC_4 + nC_5 + C_1 + C_2 - C_4 - C_5 + C_6$$
$$T(n) = (C_3 + C_4 + C_5)(n) + (C_1 + C_2 - C_4 - C_5 + C_6)$$

Si $A,\ B \in \mathbb{Z}$, donde $A = C_3 + C_4 + C_5$ y $B = C_1 + C_2 - C_4 - C_5 + C_6$, entonces $T(n) = A(n) + B \therefore T(n) \in \Theta(n)$

3.1.1.2 Análisis a posteriori

Posteriormente al ejecutar función iterativa creada con el nombre *"sucesion_fibonacci"*, se tiene que el dato de entrada es un número natural (incluido el 0) n, por tanto la salida del programa da el número de operaciones hechas por cada iteración, los resultados registrados se muestran en la Tabla 2.

Tabla 2: Complejidad temporal de la sucesión de Fibonacci iterativa

n	**T(n)**
1	8
2	14
3	20
4	26
5	32

6	38
7	44
8	50
9	56
10	62
11	68
12	74

En la Figura 1 se muestra la gráfica obtenida para los valores de la Tabla 2. Dado el comportamiento de los puntos de muestra, se aprecia que este es

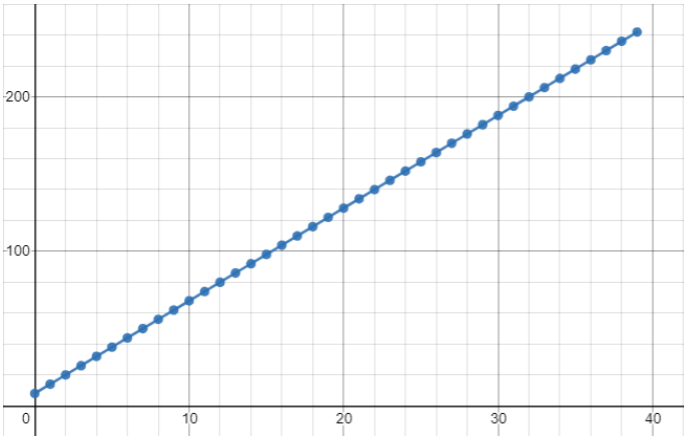

Figura 1: Gráfica del orden de complejidad del algoritmo iterativo.

lineal, no habiendo diferencia entre el mejor y peor caso, y conociendo que Θ se define como:

$$\Theta(g(n)) = \{f(n) \mid \exists_n \; C_1, \; C_2 > 0 \; \& \; n_0 > 0 \; tal \; que$$

$$0 \leq \; C_1 g(n) \; \leq \; f(n) \; \leq \; C_2 g(n) \; \forall \; n \geq \; n_0\}$$

Se propone $C_1 = 4$, $C_2 = 8$ y $n_0 = 6$, es posible obsevar en la Figura 2 que dados estos valores se acotan la parte superior como la inferior así como el punto de cruce en el que se cumple la inecuación, por tanto se puede decir que $T(n) \in \Theta(n)$.

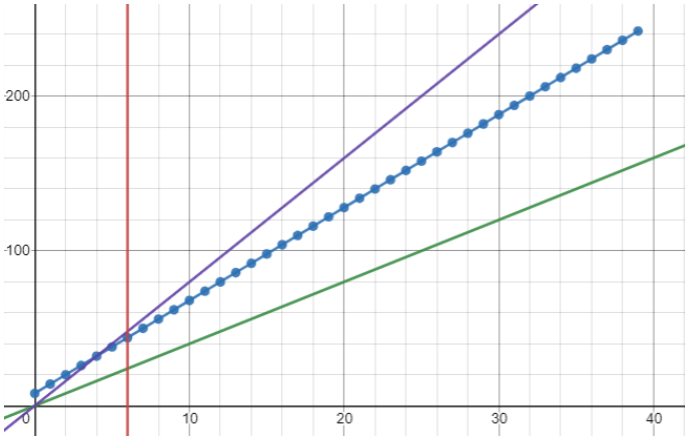

Figura 2: Gráfica de las curvas que acotan en la parte superior e inferior a los puntos muéstrales

3.1.2 Implementación recursiva de la Sucesión de Fibonacci

Posteriormente al ejecutar función recursiva creada con el nombre *"sucesion_fibonacci_R"*, se tiene que el dato de entrada es un número natural (incluido el 0) n, por tanto la salida del programa da el número de operaciones hechas, los resultados registrados se muestran en la Tabla 3.

Tabla 3: Complejidad temporal de la sucesión de Fibonacci recursiva

n	**T(n)**
1	2
2	2
3	8
4	14
5	26
6	44
7	74
8	122
9	200
10	326
11	530
12	860

Dados los tiempos de ejecución por la entrada n del algoritmo recursivo presentado se muestra que la construcción hacia la solución del problema es por medio de un árbol, simplemente se observa que a medida que se meten valores

más grandes las hojas de este crecen en un ritmo exponencial, donde el tiempo de ejecución esta determinado por el siguiente cociente $\frac{X_{n+1}}{X_n}$ lo cual recuerda la estrecha relación que tiene la sucesión de Fibonacci con el número áureo, por lo que para encontrar $sucesion\_fibonacci\_R(n)$ conlleva un tiempo mayor o igual que φ^n.
En la Figura 3 se muestran la cota superior e inferior donde al no a ver diferencias entre el mejor y el peor caso se hace usos de la definición de θ por lo que proponiendo $C_1 = 1$, $C_2 = 2$ y $n_0 = 5$ por tanto se puede decir que $T(n) \in \Theta(\varphi^n)$ por lo que el algoritmo presenta una complejidad exponencial.

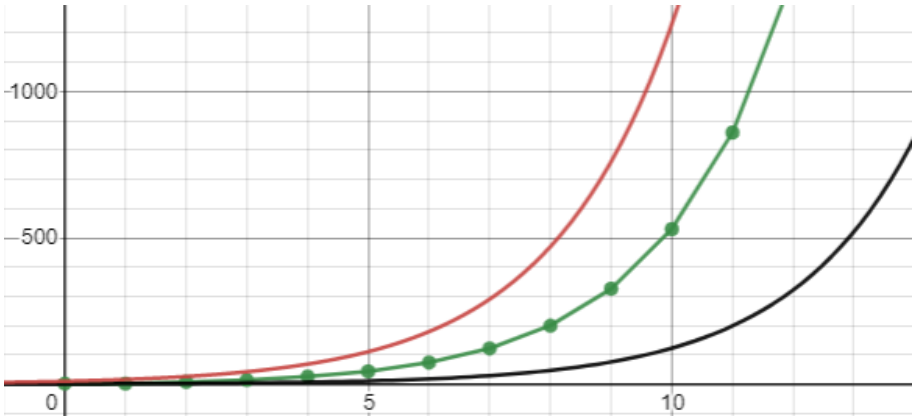

Figura 3: Gráfica de las curvas que acotan en la parte superior e inferior a los puntos muéstrales

3.2 Algoritmo para obtener los primeros n números perfectos

En cuanto al segundo problema, implementaremos el algoritmo de test de primalidad de Fermat, además, haremos uso del teorema de Euclides-Euler. A continuación mostraremos el análisis a priori y posteriori.

3.2.1 Análisis a priori

Para el análisis a priori, primero debemos calcular el orden de complejidad de cada linea de nuestro código. A continuación se mostraran las tablas de las funciones MostrarPerfectos(), Fermat() y Modulo().

Tabla 4: Análisis a priori linea por linea para el mejor y peor caso

No.	**Código**	**Costo**	**# Pasos ejecutables**	
	modulo(int base, int e, int mod)		**Mejor caso**	**Peor caso**
1	a = 1;	C_1	1	1
2	b = base;	C_2	1	1
3	while(e>0){	C_3	$log_2(2n)+1$	$log_2(2n)+1$
4	if(e%2 == 1):	C_4	$log_2(2n)$	$log_2(2n)$
5	a=(a*b)%mod }	C_5	$log_2(2n)$	$log_2(2n)$
6	b=(b*b)%mod	C_6	$log_2(2n)$	$log_2(2n)$
7	e=e/2}	C_7	$log_2(2n)$	$log_2(2n)$
8	return a%mod	C_8	1	1

$T(n) = C_1(1) + C_2(1) + C_3(log_2(2n)+1) + C_4(log_2(2n)) + C_5(log_2(2n)) + C_6(log_2(2n)) + C_7(log_2(2n)) + C_8(1)$

$T(n) = C_1 + C_2 + C_3(log_2(2n)) + C_3 + C_4(log_2(2n)) + C_5(log_2(2n)) + C_6(log_2(2n)) + C_7(log_2(2n)) + C_8$

$T(n) = C_1 + C_2 + C_3 + C_8 + C_3(log_2(2n)) + C_4(log_2(2n)) + C_5(log_2(2n)) + C_6(log_2(2n)) + C_7(log_2(2n))$

Si $A,\ B\ \in\ \mathbb{Z}$, donde $A = C_3 + C_4 + C_5 + C_6 + C_7$ y $B = C_1 + C_2 + C_3 + C_8$ y , entonces $T(n) = A(log_2(n)) + B\ \therefore\ T(n)\ \in\ \Theta(log_2(n))$

Tabla 5: Análisis a priori linea por linea para el mejor y peor caso de la función Fermat

No.	**Código**	**Costo**	**# Pasos ejecutables**	
	modulo(int base, int e, int mod)		**Mejor caso**	**Peor caso**
1	if($m == 1$)	C_1	1	1
2	return false	C_2	1	1
3	for($intj = 0; j < 20; j++$){	C_3	21	21
4	$x = rand()\%(m-1)+1$	C_4	$20log_2(2n)$	$20log_2(2n)$
5	if(modulo(x,m-1,m)}	C_5	$20log_2(2n)$	$20log_2(2n)$
6	return false}	C_6	$20log_2(2n)$	$20log_2(2n)$
7	return true	C_7	1	1

$T(n) = C_1(1) + C_2(1) + C_3(21) + C_4(20log_2(n)) + C_5(20log_2(n)) + C_6(20log_2(n)) + C_7(1)$
$T(n) = C_1 + C_2 + C_3 + C_4 + C_7 + C_4(log_2(n)) + C_5(log_2(n)) + C_6(log_2(n))$
Si $A,\ B \in \mathbb{Z}$, donde $A = C_1 + C_2 + C_3 + C_4 + C_7$ y $B = C_1 + C_4 + C_5 + C_6$
y , entonces $T(n) = A(log_2(n)) + B \therefore T(n) \in \Theta(log_2(n))$

Tabla 6: Análisis a priori linea por linea para el mejor y peor caso de la función Mostrar Perfectos

No.	**Código**	**Costo**	**# Pasos ejecutables**	
	mostrarPerfectos(int n)		**Mejor caso**	**Peor caso**
1	i = 1	C_1	1	1
2	while(n)	C_2	1	1
3	if ($Fermat(i) == 1$)	C_3	$log_2(2n)$	$log_2(2n)$
4	if ($Fermat(2^{i-1})$){	C_4	$log_2^2(2n)$	$log_2^2(2n)$
5	Mostrar $(2^{i-1}) \cdot (2^i - 1)$ es perfecto	C_5	$log_2^2(2n)$	$log_2^2(2n)$
6	n–}	C_6	$log_2^2(2n)$	$log_2^2(2n)$
7	i++	C_7	$log_2(2n)$	$log_2(2n)$

$T(n) = C_1(1) + C_2(1) + C_3(log_2(n)) + C_4(log_2^2(n)) + C_5(log_2^2(n)) + C_6(log_2^2(n)) + C_7(log_2(n))$
$T(n) = C_1 + C_2 + (C_3 + C_7) \cdot (log_2(2n) + (C_4 + C_5 + C_6) \cdot (log_2^2(2n))$
Si $A,\ B,\ C \in \mathbb{Z}$, donde $A = C_4 + C_5 + C_6$, $B = C_3 + C_7$ y $C = C1 + C2$
, entonces $T(n) = A(log_2^2(n)) + B(log_2(n) + C \therefore T(n) \in \Theta(log_2^2(n))$

3.2.2 Análisis a posteriori

En nuestro análisis a posteriori podremos identificar el numero de ejecuciones del código por cada incremento en n, siendo n los primeros números perfectos. A continuación se mostrará la tabla de los primeros ocho números perfectos y su respectivo numero de ejecuciones para obtenerlo.

Tabla 7: Complejidad temporal de MostrarPerfectos()

n	**T(n)**
1	478
2	1088
3	1936
4	2967
5	5067
6	7095
7	9212
8	13770

Como puede apreciarse en la figura 4, el comportamiento de nuestro algoritmo pertenece a una ecuación cuadrática que se aproxima a $220n^2$. Sabemos que no existe diferencia entre el mejor y peor caso, y conociendo que Θ se define como:

$$\Theta(g(n)) = \{f(n) \mid \exists_n\ C_1,\ C_2 > 0\ \&\ n_0 > 0\ tal\ que$$
$$0 \leq\ C_1 g(n)\ \leq\ f(n)\ \leq\ C_2 g(n)\ \forall\ n \geq\ n_0\}$$

Se propone $C_1 = 100$, $C_2 = 300$ y $n_0 = 4$, y con estos valores la cota superior es de $300n^2$, mientras que la cota inferior es de $100n^2$.Observamos que tanto las cotas como el punto de cruce cumplen la inecuación, por tanto se puede decir que $T(n) = 220n^2 \in \Theta(n^2)$.

La Figura 5 muestra la salida de la ejecución del código. Se alcanzaron hasta ocho números en un tiempo de 0.16 segundos, el cual es un tiempo muy razonable. Sin embargo el noveno numero tarda más de 2 minutos en encontrarlo. Podemos ver que a partir del cuarto numero, las cantidades se vuelven exponenciales, tan solo el octavo numero ya cuenta con 19 dígitos.

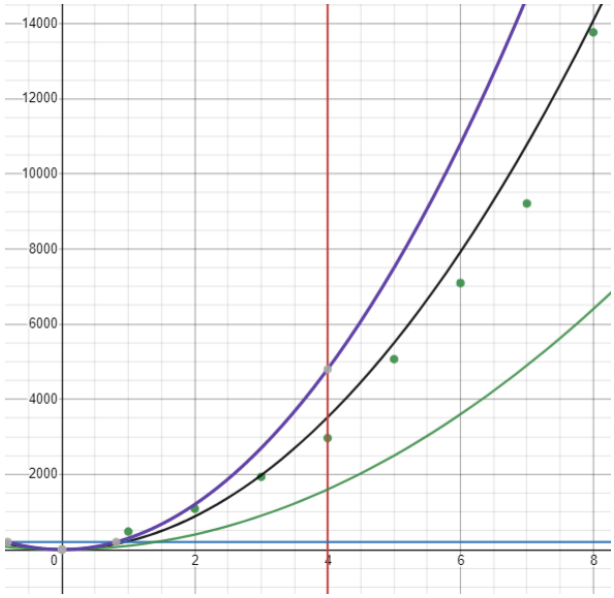

Figura 4: Gráfica de las curvas que acotan en la parte superior e inferior a los puntos muéstrales

```
Compilation time: 0.73 sec, absolute running time: 0.16 sec, cpu time: 0.01 sec.


6 es Perfecto
28 es Perfecto
496 es Perfecto
8128 es Perfecto
33550336 es Perfecto
8589869056 es Perfecto
137438691328 es Perfecto
2305843008139952128 es Perfecto
```

Figura 5: Salida de la ejecución del programa

4 Conclusiones

Luis Francisco Renteria Cedillo

En esta practica se pudo verificar la importancia de desarrollar algoritmos eficientes, ya que, en el segundo problema, el código hecho por fuerza bruta tardaba demasiado y no llegaba a mostrar más de cuatro números perfectos en un tiempo menor a treinta segundos. Investigando más sobre estos números, encontramos la formula de Euclides, y solo debíamos encontrar los primos que cumplían sus condiciones. Aquí fue cuando recordé el teorema de Fermat y su algoritmo de test de primalidad, el cual es probabilístico y debe ejecutarse varias veces para reducir su probabilidad de falsos positivos. Dicho algoritmo lo usamos en la asignatura de criptografía, en específico al desarrollar el sistema de encriptado RSA. Con esto se pudo lograr alcanzar los primeros ocho números perfectos.

Denzel Omar Vazquez Perez

Una de las cosas que deja de aprendizaje la practica es de nuevo que el estilo

de programación afecta directamente en los resultados que se espera obtener, esto se logra percatar en el el algoritmo de los números de Fibonacci donde la implementación iterativa como recursiva afecta directamente en el tiempo de ejecución de tal programa ya que la meter un n mas grande el proceso y los plazos de tiempo se hacen mas grandes esto es evidente en el algoritmo recursivo puesto que su complejidad es exponencial mientras que la del algoritmo iterativo es lineal.
Para el segundo problema la primer implementación se hizo en C sin embargo el tiempo de espera por la ejecución es inmenso por lo que se decidio codificar en C++, para obtener un resultado mas amplio a partir de conceptos matemáticos un poco mas avanzados, así se logro mostrar de manera eficiente los números perfectos que se solicitaban en la practica. Sin embargo es interesante observar que es importante el tener una noción del algoritmo a usar ya que dada su complejidad sera la forma mas rápida de resolver el problema.

5 Bibliografía

Wolfram Research (2022) *Perfect Numbers*, disponible en https://mathworld.wolfram.com/PerfectNumber.html, consultado el 14 de marzo del 2022.

Brassard, G. (1997). *Fundamentos de Algoritmia*. España: Ed. Prentice Hall.

Cormen, E. A. (2022). *Introduction To Algorithms*, 3Rd Ed. Phi.

Luna, Benjamín. *Fundamentos para el análisis de eficiencia algorítmica*. Escuela Superior de Computo, IPN. México. 14 de Marzo de 2022.

Sánchez, P. J. I. (2006). *Análisis y diseño de algoritmos: un enfoque teórico y práctico*. Servicio de Publicaciones y Divulgación Científica de la UMA.