Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 1, 7-11-2020

Práctica 3: Complejidades temporales polinomiales y no polinomiales.

De la Cruz Sierra Diana Paola
Raya Chávez Samuel Antonio
dianapaodcs@gmail.com, samiraya1323@gmail.com

Resumen. Primero se observará la diferencia de un algoritmo recursivo y uno iterativo con un programa que implemente la sucesión de Fibonacci. Luego se explicará sobre aquellos algoritmos que calculan un número perfecto.

Palabras clave. C++, Fibonacci, números perfectos, recursividad

1. Introducción

La teoría de la complejidad estudia la manera de clasificar problemas de acuerdo con la dificultad propia para resolverlos, basándose en los recursos necesarios y requeridos para establecer su grado de complejidad. Un cálculo resulta complejo si es difícil de realizar. El verdadero inconveniente es saber si existe una solución o no a una problemática presentada y qué tan compleja resultará su solución.

Para saber el grado de complejidad que puede tener un problema, nos apoyamos en el modelo computacional de la máquina de Turing,con el cual se obtiene una clasificación de los problemas con base en el grado de complejidad inherente para resolverlos.A través de este modelo se han detectado problemas intratables, clasificados como NP (nondeterministic polynomial time), que se piensa son imposibles de resolver en un tiempo razonable cuando el número de variables que los componen es una cantidad extremadamente grande.

Dentro de este grupo de problemas NP se encuentran dos subconjuntos de problemas: problemas P, para los cuales existe una máquina de Turing determinista que los puede resolver en tiempo polinómico.

Es importante aclarar que "tiempo polinomial"significa que la complejidad del algoritmo es O(p(x)), donde x es la medida de los datos del problema y p(x) es un polinomio en x, es decir, que su grado es constante con respecto a x.

Para los NP-completos. No existe una máquina de Turing determinista que pueda resolverlos en tiempo polinómico. En su lugar, se puede encontrar un valor próximo a la solución del problema acotando polinomialmente el tiempo. Estos problemas NP son aquellos cuya solución, hasta la fecha, no se ha resuelto de manera exacta por medio de algoritmos deterministas en tiempo polinomial.

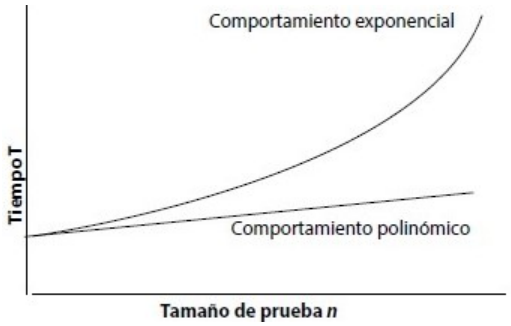


Figura 1: Tiempo polinomial vs exponencial.

en la figura 1 se muestra la diferencia entre una solución obtenida en tiempo polinomial y una en tiempo exponencial.

Acerca de por qué los problemas P están incluidos dentro de los considerados difíciles de resolver NP, esto es porque los algoritmos no determinísticos usados para los problemas NP y NP-completo, también pueden usarse en los problemas P; pero no es posible, en caso contrario, que un algoritmo determinístico que resuelva un problema P en tiempo polinomial, también pueda resolver un NP.

De acuerdo con la teoría de la complejidad, si el problema se puede clasificar como P, entonces se podría hacer uso, encontrar o desarrollar algoritmos eficientes que comprueben si obtienen la mejor solución para dicho problema. En caso de que el problema estuviera clasificado como NP o NP-completo, entonces se perdería el tiempo tratando de encontrar algoritmos eficientes.En lugar de eso, el camino más adecuado es trabajar con heurísticas computacionales, por lo que solo quedaría tratar de proponer nuevas heurísticas de baja complejidad.De ahí el uso de algoritmos heurísticos no determinísticos acotados en tiempo polinomial para este tipo de problemas.

2. Conceptos básicos

- **Serie de Fibonacci:** Es una sucesión de números en la que el número que continua a partir del tercero es el resultado de la suma de los dos anteriores. Es una sucesión matemática infinita, está elaborada a partir de una serie de números naturales iniciando por el cero, y sumándose de a dos empezando por 0 y 1.

  Por ejemplo: 0, 1, 1, 2, 3, 5, 8, 13… (Pedroza C, 2012)

- **Número perfecto:** Un número entero positivo es perfecto, si es igual a la suma de sus divisores menores.

  Por ejemplo, 8 no es un número perfecto ya que $8 \neq 1 + 2 + 4$, por otro lado, 6 si es un número perfecto, ya que $6 = 1 + 2 + 3$. (Benjamín, B, 2020)

- **Recursividad:** Es una técnica utilizada en programación que nos permite que un bloque de instrucciones se ejecute un cierto número de veces (el que nosotros determinemos). (Pérez, M., 2015)

2.1. Pseudo-códigos para el algoritmo Fibonacci

A continuación se mostraran 2 algoritmos para calcular hasta el n-ésimo término de la sucesión de Fibonacci, el primero lo muestra de forma iterativa, mientras que el segundo de forma recursiva.

Algorithm 1 Pseudocódigo: Fibonacci iterativo

```
1: procedure FIBONACCI-ITERATIVO(int n)
2:     x ← 0
3:     y ← 1
4:     for i ← 1, n do
5:         z ← x + y
6:         x ← y
7:         y ← z
8:         print: z
9:     end for
10: end procedure
```

Algorithm 2 Pseudocódigo: Fibonacci recursivo

```
1: procedure FIBO-RECURSIVO(int n)
2:     x ← 0
3:     y ← 1
4:     if n <= 1 then
5:         return n
6:     else
7:         return FiboRecursivo(n − 1) + FiboRecursivo(n − 2)
8:     end if
9: end procedure
```

2.2. Pseudo-códigos para el algoritmo de número perfecto

El pseudocódigo planteado para un algoritmo que encuentre si un número es perfecto es el siguiente:

Algorithm 3 Pseudocódigo: Verificación de que un número es perfecto

```
1: procedure PERFECTO(n)
2:     aux ← 1
3:     while aux ≠ n do
4:         if n%aux ← 0 then
5:             suma ← suma + aux
6:         end if
7:         aux + +
8:     end while
9:     if suma ← n then
10:        return 1
11:    else
12:        return 0
13:    end if
14: end procedure
```

Donde se retorna el valor 1 si el número n es perfecto.

Y para encontrar los primeros n números perfectos, se tiene el siguiente algoritmo:

Algorithm 4 Pseudocódigo: Mostrar los primeros n números perfectos

```
1: procedure MostrarPerfectos(n)
2:     aux ← 1
3:     while cont ≠ n do
4:         if perfecto(aux) ← 1 then
5:             cont + +
6:             print aux
7:         end if
8:         aux + +
9:     end while
10:    return 1
11: end procedure
```

3. Experimentación y resultados

3.1. Algoritmo Fibonacci

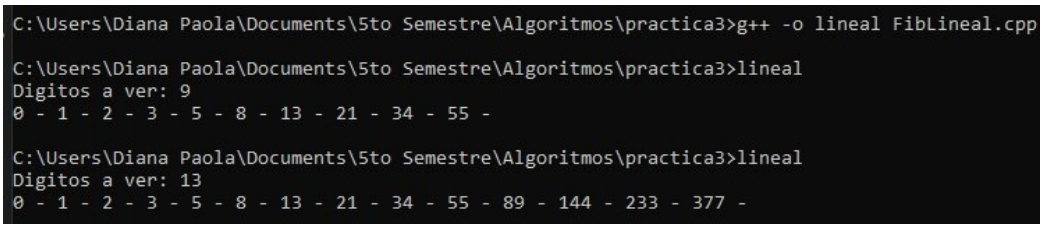


Figura 2: Pruebas de 2 ejemplos en ejecución del algoritmo1

En la figura 2 se observan 2 ejemplos del algoritmo 1, el cual es programado de manera iterativa.

Para este algoritmo tanto el mejor como el peor caso es el mismo, ya que se ejecutará el cuerpo del *for* exactamente n veces, a continuación en la tabla 1 se muestra el procedimiento de un análisis a priori para calcular la complejidad del algoritmo 1.

	Costo	**No. pasos ejecutados**
x = 0	c1	1
y = 1	c2	1
for i =1 to i=n	c3	n+1
z = x+y	c4	n
x = y	c5	n
y = z	c6	n
print z	c7	n

Cuadro 1: Análisis A priori

T(n) = c1 + c2 + c3 + c3*n + c4*n + c5*n + c6*n + c7*n = (c3+c4+c5+c6+c7)*n + (c1+c2+c3) = $an + b$

Con esto podemos observar que el orden de complejidad del algoritmo 1 es lineal. Si hacemos un análisis *aposteriori* vemos que las veces que el código se ejecuta es: T(n)=4+3n

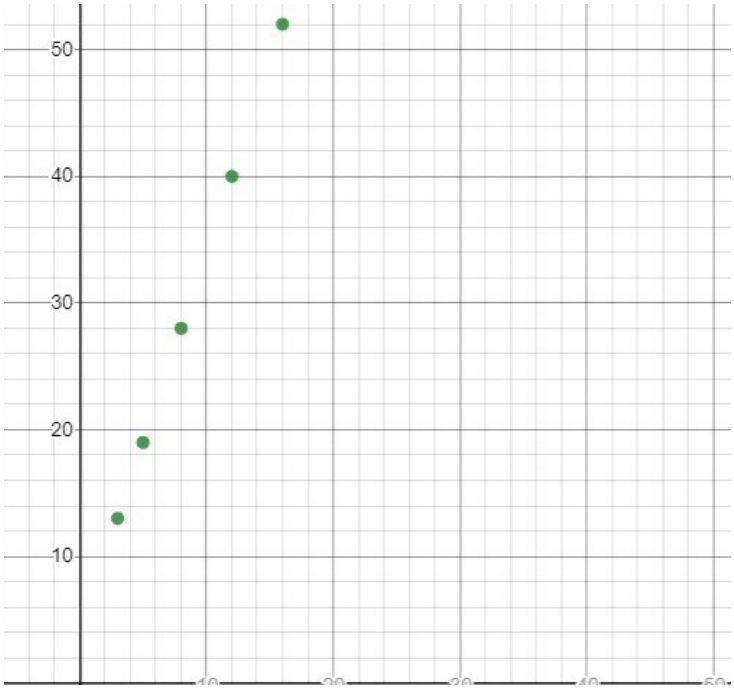


Figura 3: Puntos del análisis a posteriori del algoritmo 1

En la figura 3 se pueden observar los puntos graficados en el análisis a posteriori. Viendo la forma que toman, podría proponerse como una ecuación que se ajuste a estos puntos T(n) = 4n, cuya gráfica es mostrada en la figura 4.

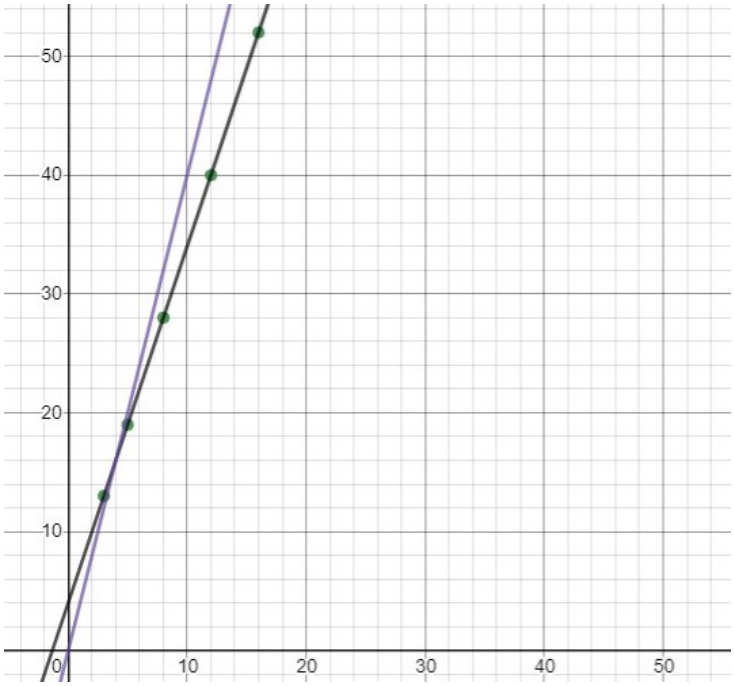


Figura 4: Ecuación propuesta y puntos del análisis a posteriori

Estas gráficas mostradas anteriormente comprueban lo dicho en el análisis a priori y se concluye que en el algoritmo 1 $T(n) \in O(n)$

```
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\practica3>g++ -o recursivo FibRecursivo.cpp

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\practica3>recursivo
Digitos a ver: 10
0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 -

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\practica3>recursivo
Digitos a ver: 15
0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 - 89 - 144 - 233 - 377 - 610 -
```

Figura 5: Pruebas de 2 ejemplos en ejecución del algoritmo2

En la figura 5 se observan 2 ejemplos del algoritmo 2, el cual es programado de manera recursiva.

vamos a hacer un análisis utilizando unos ejemplos:
por ejemplo para fibo (4)
fibo(n) = fibo(n-1)+fibo(n-2)
fibo(4) = fibo(3) + fibo(2) = [fibo(2)+fibo(1)] + [fibo(1)+fibo(0)]
= [(fibo(1)+fibo(0)) + 1] + [1 + 0]
Eso da un total de 9 llamadas a la función Siguiendo con este procedimiento de forma similar se observa en la tabla 2 el número de llamadas en la función de FibonacciRecursivo (n) con parametro n.

Fibo(n)	No. llamadas a la función
2	3
4	9
6	25
8	67

Cuadro 2: No de llamadas a la función en el algortimo recursivo 2

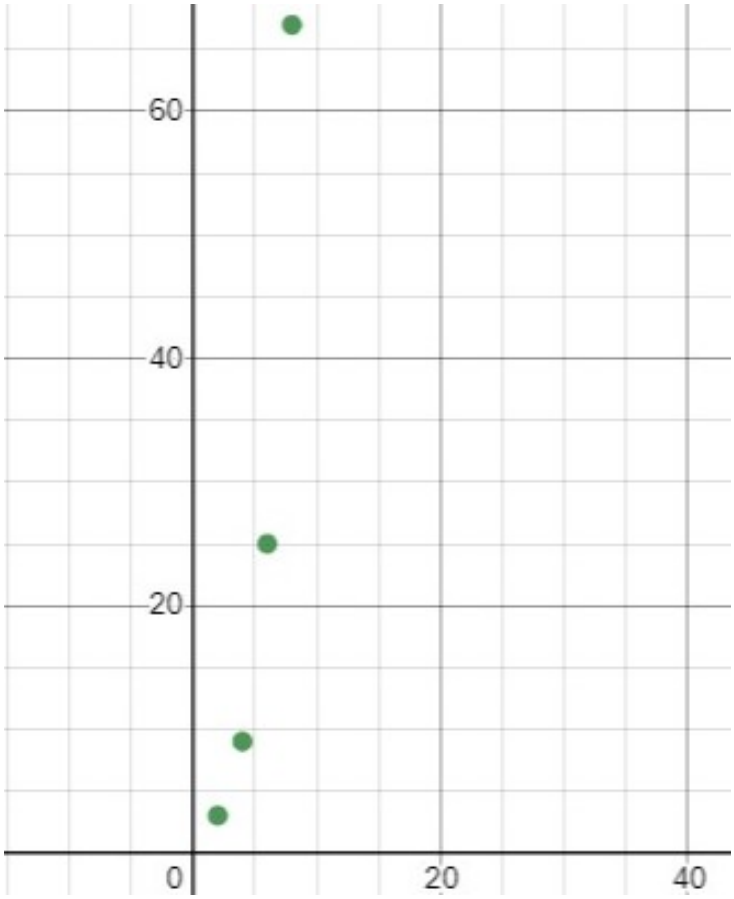


Figura 6: Puntos del análisis a posteriori del algoritmo 2

En la figura 6 se pueden observar los puntos graficados de la tabla 2 y en la figura 7 se puede observar la ecuación propuesta

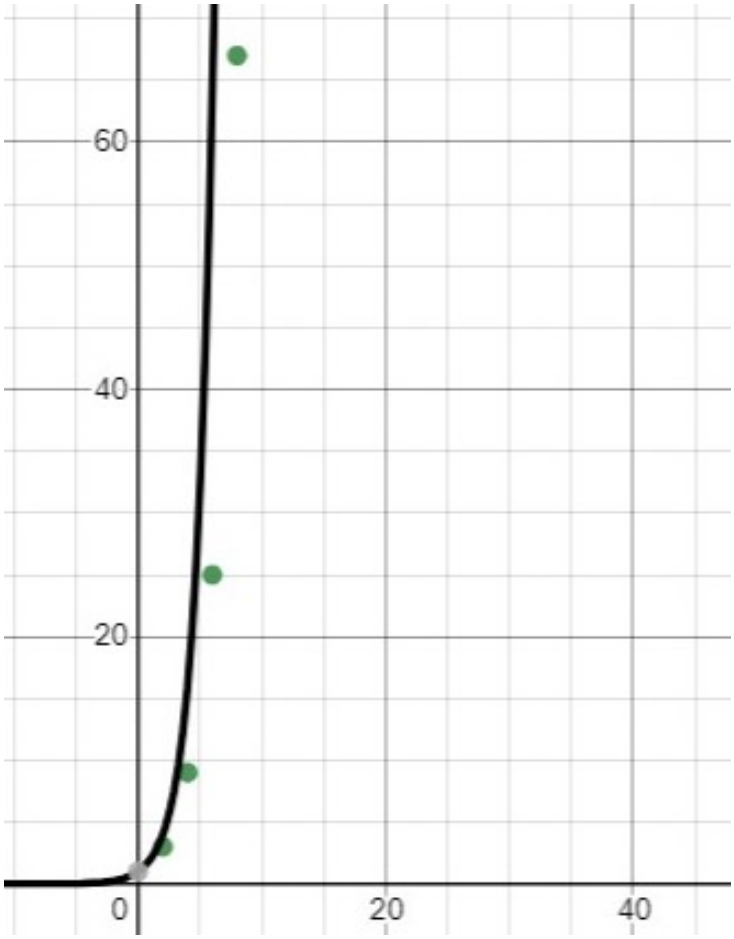


Figura 7: Ecuación propuesta vs puntos análisis a posteriori

podemos observar que la gráfica propuesta 2^n mostrada anteriormente en la figura 7 acota por arriba a los puntos gráficados. Por lo tanto podemos concluir que el algoritmo 2 Fibonacci-Recursivo $\in 2^n$

3.2. Algoritmo número perfecto

Mostrando los resultados que arroja el algoritmos perfecto(n), se tiene:

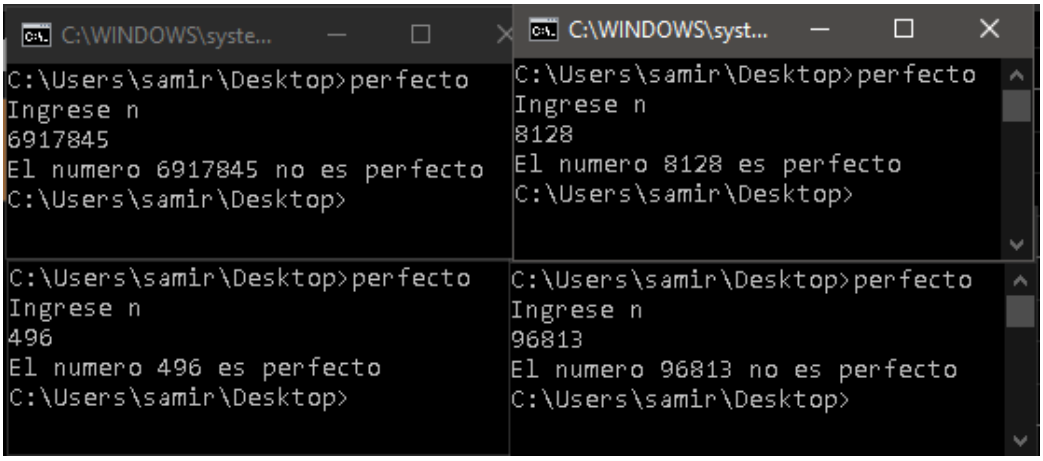


Figura 8: Resultados del algoritmo perfecto(n)

Haciendo un análisis del algoritmo 3 para calcular su orden de complejidad, en el peor de los casos que es cuando n es un número tan grande que se tendrán que hacer muchas operaciones, se plantea:

	Costo	**No. pasos ejecutados**
aux = 1	c1	1
while aux *neq* n	c2	n+1
if n % aux = 0	c3	n
suma = suma + aux	c4	$2^{p-1}(2^p\text{-}1)$
aux++	c5	n
if suma = n	c6	n
return 1	c7	1
else	c8	0
return 0	c9	0

Cuadro 3: Análisis A priori del algoritmo 3

De $2^{p-1}(2^p\text{-}1)$ se obtiene que es igual a log(p-1)*(log(p)-1) donde p es número primo, el orden de complejidad para obtener un número primo es de manera lineal (O(n)), lo que convierte la ecuación a log(O(n)-1)*(log(O(n))-1) y en cualquier caso, es menor o igual a n. Por lo que se obtiene:

T(n) = c1 + (n+1)*c2 + n*c3 + (log(p-1)*(log(p)-1)*c4 + n*c5 + n*c6 + c7 + 0*c8 + 0*c9

= (c2+c3+c5+c6)*n +c4*(log(p-1))*(log(p)-1) (c1+c2+c7)

$= an + b$

De aquí se observa que el algoritmo 3 tiene una complejidad lineal, ahora analizando con un análisis a Posteriori con t(n)=4n+(log(p-1))*(log(p)-1)+3, que son las veces que el código s ejecuta, se tiene:

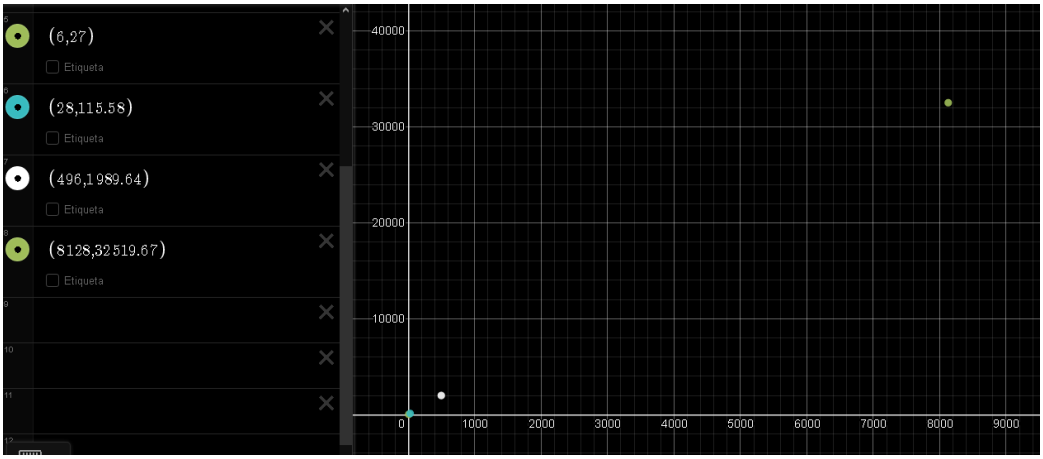


Figura 9: Puntos arbitrarios del resultado de perfecto(n), con x=n & y=T(n).

Uniendo los puntos y proponiendo una gráfica nos da:

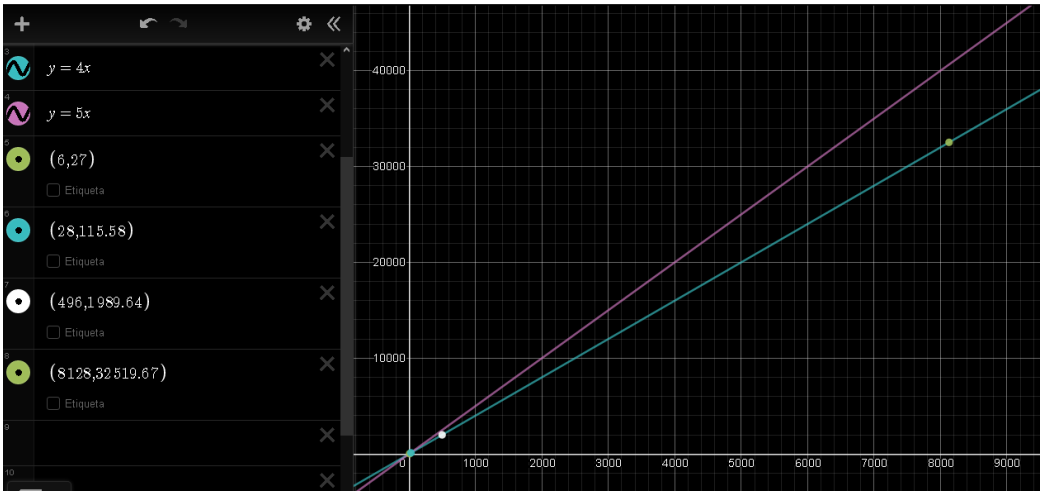


Figura 10: Gráficas de perfecto(n), con y=4x que pasa hasta donde n=8,128 & y=5x que es la recta superior a todos los puntos.

Con esto, queda demostrado que el algoritmo 3: perfecto(n) $\in$ O(n).

Y para MostrarPerfectos(n), se tienen los siguientes resultados:

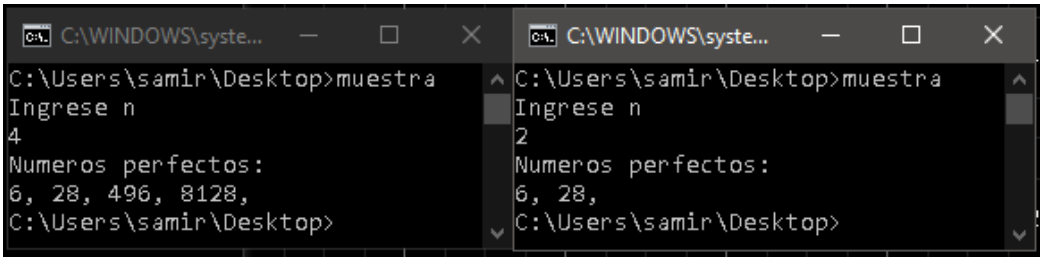


Figura 11: Resultados del algoritmo MostrarPerfectos(n)

Se considerará la complejidad de MostrarPerfectos(n) como
T(n) = n*(3+O(perfecto(aux)))+3. Y recordando que perfecto(n) tiene una complejidad lineal, por lo que haciendo unas gráficas experimentales se tiene:

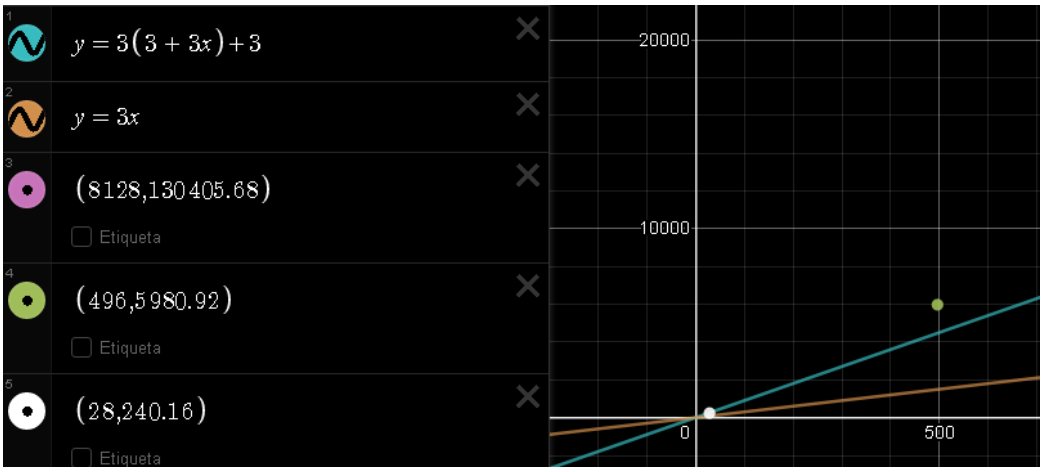


Figura 12: Gráfica propuesta del algoritmo MostrarPerfectos(n) 1

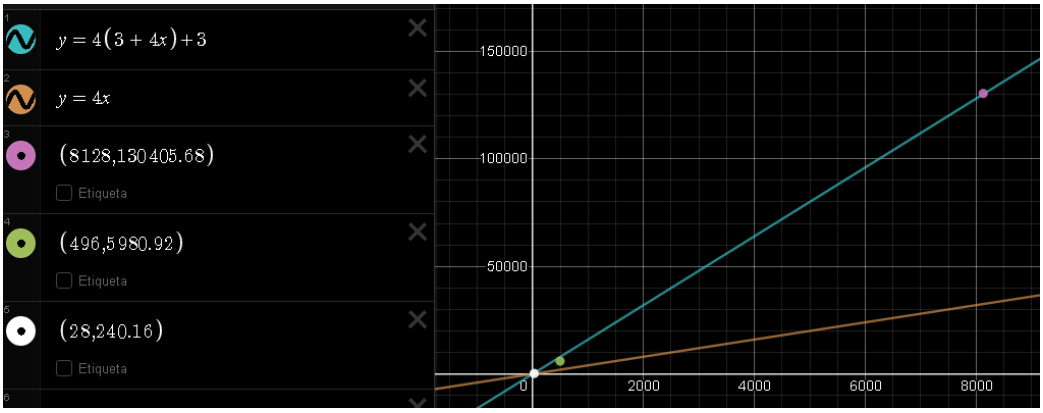


Figura 13: Gráfica propuesta del algoritmo MostrarPerfectos(n) 2

Donde se observan los puntos con x=n y y=T(n), se comparan ambas rectas de los programas, observando que el tiempo de respuesta de MostrarPerfectos(n) es mayor que Perfectos(n), y esta recta va aumentando de tamaño cuanto más grande sea n.

Ahora, mostrando algunos números perfectos:

Pos.	p	Número perfecto	Nº dígitos	Año	Descubridor
1	2	6	1	?	
2	3	28	2	?	
3	5	496	3	?	
4	7	8 128	4	?	
5	13	33 550 336	8	1456	*anónimo*
6	17	8 589 869 056	10	1588	Cataldi
7	19	137 438 691 328	12	1588	Cataldi
8	31	2 305 843 008 139 952 128	19	1772	Euler
9	61	265845599...953842176	37	1883	Pervushin
10	89	191561942...548169216	54	1911	Powers

Figura 14: 10 Números perfectos Benjamín, B. (2020).

Con las gráficas propuestas anteriormente y con los números perfectos dados en la tabla 14 se puede llegar a la conclusión de que MostrarPerfectos(N) $\in$ O(n).

Con respecto a la complejidad algorítmica del algoritmo de MostrarPerfectos(n) , depende del algoritmo que se ha llevado a cabo:

- La solución simple es ir a través de todos los números de 1 a n-1 y verificar si el número auxiliar es un divisor del número perfecto pedido, luego se mantiene un sumador de todos los divisores. Si la suma es igual a n, regresará verdadero, de lo contrario retornará falso.

De esta forma y como se vio anteriormente en este documento se tendrá una complejidad lineal, O(n)

- La solución eficiente es ir a través de números hasta la raíz cuadrada de n. Si un número 'i' divide a n, entonces se agrega 'i' y n/i a la suma total.

  Mientras que para esta forma se tiene una complejidad O($\sqrt{n}$). (Kumar, M., 2019)

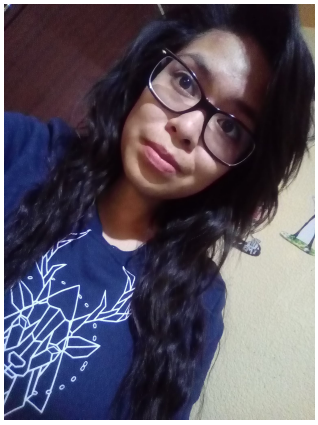
4. Conclusiones

- Conclusiones generales:

  Para el algoritmo Perfecto(n), se pensó desde un principio en hacer un ciclo que fuera de 1 en 1 unidades, pero al ir indagando un poco más sobre el tema y ver que existen 2 posibles formas de hacer este algoritmo siendo que la segunda forma lo hace en un tiempo mucho menor, da a pensar que hay que agilizar más la mente para encontrar un mejor algoritmo del que se tiene planeado. En este caso se mantuvo con la idea original puesto que fue con la que se ideo todo este desarrollo.
  En cuanto al algoritmo de fibonacci se observa que es más sencillo de implementar el algoritmo recursivo pero el costo computacional es mucho mayor, tan solo para la parte de calcular el n-ésimo término, ahora si se quieren imprimir todos costaría más tiempo debido a que en la función principal sería necesario el uso de un for para cada término

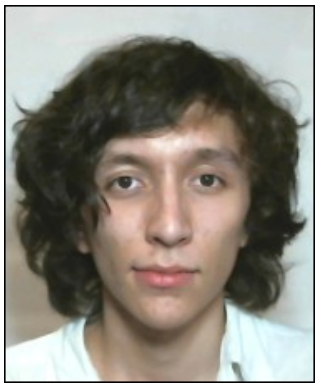
- Diana Paola De la Cruz Sierra:

  Haciendo un mejor análisis de los algoritmos desarrollados anteriormente se puede concluir que no siempre es buena idea desarrollar el algoritmo de forma recursiva, ya que tienden a gastar más memoria además de mayor tiempo de ejecución, podría ser buena idea solo si es muy difícil programarlo de manera iterativa, pero ya vimos que no hay garantía en que el tiempo sea el más óptimo.



- Raya Chávez Samuel Antonio:

  Con el desarrollo de esta práctica pude ver que no siempre el llamar a una función recursivamente va a ayudar a que el programa sea más rápido, si no que puede pasar como en este caso que MostrarPerfecto(n) se tardó más por la misma cuestión de estar llamando a perfecto(n) en cada ciclo del contador. En lo personal, me llamó y gustó mucho que dijeran que el 6 es un número perfecto ya que 6 fueron los días que Dios tardó en crear al mundo.



5. Fuentes de información

- Cardoso, R. (S.F). *Tratabilidad*. [PDF]. Disponible en: https://cursos.virtual.uniandes.edu.co/isis1105/wp-content/uploads/sites/80/2015/11/8-intratabilidad.pdf

- Cruz, M. (S.F). *Aplicación de la teoria de la complejidad en optimización combinatoria*. [PDF]. Disponible en:

https://www.google.com/url?sa=tyrct=jyq=yesrc=sysource=webycd=-cad=rja-uact=8-ved=2ahUKEwj2vKT0pPLsAhVNWs0KHc4aC0gQFjAAegQIBRAC-url=https-3A-2F-2Fdialnet.unirioja.es-2Fdescarga-2Farticulo-2F4733827.pdfyusg=AOvVaw TEYWv

- Pedroza C. (2012). *Serie Fibonacci.* Noviembre 08, 2020, de Matemáticas Modernas Sitio web: https://matematicasmodernas.com/serie-fibonacci/
- Benjamín, B. (2020). *Práctica 3, Complejidades temporales polinomiales y no polinomiales.* Escuela Superior de Cómputo, 1, 1. Noviembre 08, 2020, De Microsoft Teams Base de datos.
- Kumar, M. (2019). *Perfect Number.* Noviembre 08, 2020, de GeeksForGeeks Sitio web: https://www.geeksforgeeks.org/perfect-number/
- Pérez, M. (2015**). ¿Qué es la recursividad?**. Noviembre 09, 2020, de Geeky theory Sitio web: https://geekytheory.com/que-es-la-recursividad