



Instituto Politécnico Nacional
Escuela Superior de Cómputo



Análisis de Algoritmos, Sem.: 2022-1, 3CV11, Práctica 1, 27 septiembre 2021

PRÁCTICA 2: COMPLEJIDAD TEMPORALES POLINOMIALES Y NO POLINOMIALES

Vanegas García Andrés, Vaquera Aguilera Ethan Emiliano.

avanegasg1601@alumno.ipn.mx, evaqueraa@alumno.ipn.mx

Resumen: La practica consistió de manera resumida en la realización de dos programas, uno de los programas trataba de la serie de Fibonacci de manera iterativa, que consistía solo en el uso de un ciclo normal junto sus respectivos contadores y variables gradientes en aumento, terminado se realizo una solución para el mismo problema pero solo que ahora se iba a hacer de manera recursiva. Una vez finalizado eso se realizo la solución al problema numero dos, resolviendo una relación que consistía que un numero es perfecto si la suma de sus divisores daba como resultado el numero en cuestión, la resolución del problema no fue difícil el problema consistía mas que nada en el numero de operaciones iterativas que se tenían que hacer para obtener os numero, ambas computadoras de los integrantes del equipo solo pudieron generar cuatro números perfectos, ya que al parecer llevar al cabo la tarea dada tomaría demasiado coste de tiempo para la computadora ya que se debe de hacer cientos de miles de operaciones antes de llegar el numero en cuestión.

Palabras Clave

C: C es un lenguaje de programación estructurado, usado por ya poco más de 52 años, esto brindándole un amplio repertorio de bibliotecas, además de un gran soporte de la comunidad que lo utiliza. C al ser un lenguaje de programación estructurado comparte similitudes con otros lenguajes desarrollados en la época además de que es un lenguaje de alto nivel, que de igual manera pude ser utilizado a bajo nivel para programar en ensamblador y rutinas de memoria.

Complejidad de un Algoritmo: Este es el punto en el que se mide la complejidad de resolución de un algoritmo por medio de una computadora, poniendo énfasis en el uso de la memoria de este sistema además de, el tiempo que este toma para desarrollarlo. Esta desde el inicio de la computación es una problemática que ha ido evolucionando a modo de que hoy en día se utiliza

el análisis de algoritmos para su resolución y determinar la complejidad de un algoritmo en específico.

Análisis a priori y posterior: Según lo comprendido en clase. El análisis a priori en computación es analizar un algoritmo, de manera formalmente teórica y matemáticamente, primero haciendo un análisis de sus posibles pseudocódigos y después en base a esto realizar una formulación del comportamiento de este en base a una función matemática.

Por otro lado el análisis a posterior, es la recolección de datos por medio de la experimentación, que en el caso de un algoritmo es la programación de este para así tomar una serie de pruebas en base a las veces ejecutadas y la cantidad de veces que este algoritmo tarda para terminar.

Iteración: Una iteración es la forma de hacer una serie de pasos repetidas veces hasta llegar al resultado esperado en un rango esperado sin sobrepasar este, las iteraciones pueden ir de cualquier manera en la vida real siendo como realizar un ejercicio matemático de una raíz, una potencia numérica o un logaritmo, y en computación serían las sentencias de while, for, do-while, etc.

Recursividad: La recursividad consiste más que nada en la realización iterativa de un mismo lenguaje descriptivo, pero sin tener que mandar a llamar a una función iterativa. En palabras más simples se trata de el desarrollo de una función que se llama a sí misma, y después de llegar a un resultado esta regresa de vez en vez al inicio y finalmente otorgar el resultado final.

1 Introducción

La complejidad temporal es usada hoy en día por los informáticos para determinar el llamado tiempo de ejecución de un programa o algoritmo, para hacer posible saber el tiempo de ejecución de un programa lo que se debe de hacer es la solución en base al número de veces que se ejecuta cierta línea de código de un algoritmo, este número se le puede llamar frecuencia de tiempo, denotada como $\mathbf{T(n)}$, donde n es el tamaño máximo que va a tardar el algoritmo en ejecutarse. Otra manera de ver esto es cuando una $\mathbf{f(n)}$ hace que $\mathbf{T(n)}$, tienda a un número infinito diferente de cero produciendo que ambas sean de un orden igual, denotando se como $T(n) = O(f(n))$, donde $O(f(n))$ es la complejidad de transcurso de tiempo del algoritmo.

Un ejemplo puede ser el siguiente. $O(2n^2 + n + 4) = O(7n^2 + 5n)$, donde a pesar de ser funciones con factores diferentes, el orden de las funciones son

el mismo por lo que son iguales a $O(n^2)$, denotando se como el final de la operación para encontrar la complejidad de tiempo.

A aquellas funciones que se describen el tiempo de ejecución por medio de una polinomio $T = O(n^K)$ se le llama función polinomial, siendo que estos se ejecutan de manera eficaz y rápida por medio de esta función. La eficiencia de este tiempo de ejecución se determina por el numero de operaciones aritméticas de entrada o salida, además de un factor polinómico que determina el numero de veces de una operación que se debe de realizar.

La complejidad temporal no polinomial es aquella que prefiere hacer uso de la razón, lógica booleana y una serie de estructuras no deterministas, otra manera de describirse puede ser todos aquellos problemas que se resuelven de manera no deterministas por una maquina de Turing en un tiempo polinómico.

2 Conceptos Básicos

Θ : La sigla griega Teta (Θ) es la denotación que describe de manera clara, el comportamiento similar de las funciones matemáticas del peor y mejor caso, esto siendo una referencia clara para hacer distinguir de Ω y O , que tiene definiciones diferentes.

Definición formal: $\Theta(g(n)) = \{f(n) : \text{existen } c_1, c_2, > 0 \text{ y } n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$

O : Big O es la denotación usada para el conjunto de funciones que se acotan por encima de $f(n)$, describiendo de manera clara el comportamiento de la función del peor caso, y denota los puntos experimentales de esta también.

Definición Formal: $O(g(n)) = \{f(n) : \text{existen } c > 0 \text{ y } n_0 > 0 \text{ constantes} \mid 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$

Ω : La denotación Omega (Ω) es la denotación usada para acotar por abajo a la función $f(n)$, siendo que es usada para describir al conjunto de funciones que están por debajo de la función experimental $f(n)$, y de igual manera sirve para describir los puntos experimentales de igual manera.

Definición Formal: $\Omega(g(n)) = \{f(n) : \text{existen } c > 0 \text{ y } n_0 > 0 \text{ constantes} \mid cg(n) \leq f(n) \forall n \geq n_0\}$

Fibonacci: Este es un algoritmo usado para encontrar una sucesión numérica infinita tan solo partiendo de cero y uno, siendo que el numero anterior se suma con el siguiente esto provocando una suma de números enteros naturales infinita. El uso de esta sucesión es precisa para la generación de un arreglo que

va a ser de utilidad para la obtención de un numero m y otro numero n y que no tengan un mínimo común divisor así poder poder aplicar el algoritmo de Euclides para el peor caso.

Algorithm 1 Algoritmo serie de Fibonacci

```

 $a \leftarrow 1$ 
 $b \leftarrow 0$ 
 $i \leftarrow 0$ 
while  $i \neq n$  do
     $a = n + a$ 
     $b = b + a$ 
end while
  
```

Numero perfecto: Se le denomina numero perfecto a aquellos números que sus divisores predecesores, tienen la capacidad de al ser sumados cumplen con la propiedad de que su resultado es igual al numero que nosotros denominamos perfecto. Por ejemplo esta el seis que cumple lo siguiente: $1 + 2 + 3 = 6$, y un numero que no cumple con esto es el ocho observándose: $1 + 2 + 4 = 7 \parallel 7 \neq 8$, por lo tanto no se cumple la propiedad de los números perfectos. A día de hoy solo se conoce la existencia de 51 números perfectos, siendo que cada uno de estos se hace cada vez mas grande tanto en dígitos como en escala de dificultad para ser encontrados.

Figura 1. Lista de Los Números Perfectos

| Pcs. | 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 | 3 589 809 056 | 10 | 1508 | 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 | 1383 | Pervushin |
| 10 | 89 | 191561942...548169216 | 54 | 1911 | Powers |
| 11 | 107 | 131640364...783728128 | 65 | 1914 | Powers |
| 12 | 127 | 144740111...199152128 | 77 | 1376 | Lucas |
| 13 | 521 | 235627234...555646976 | 314 | 1952 | Robinson |
| 14 | 607 | 141053783...537328128 | 366 | 1952 | Robinson |
| 15 | 1 279 | 541625262...984291328 | 770 | 1952 | Robinson |
| 16 | 2 203 | 108925835...453782528 | 1327 | 1952 | Robinson |
| 17 | 2 281 | 994970543...139915776 | 1373 | 1952 | Robinson |
| 18 | 3 217 | 335708321...628525056 | 1937 | 1957 | Riesel |
| 19 | 4 253 | 182017490...133377536 | 2561 | 1961 | Hurwitz |
| 20 | 4 423 | 407672717...912534528 | 2663 | 1961 | Hurwitz |
| 21 | 9 689 | 114347317...429577216 | 5834 | 1963 | Gillies |
| 22 | 9 941 | 598885496...073496576 | 5985 | 1963 | Gillies |
| 23 | 11 213 | 395961321...691086336 | 6751 | 1963 | Gillies |
| 24 | 19 937 | 931144559...271942656 | 12003 | 1971 | Tuckerman |
| 25 | 21 701 | 100656497...141605376 | 13066 | 1978 | Noll y Nickel |
| 26 | 23 209 | 811537765...941666816 | 13973 | 1979 | Noll |
| 27 | 44 497 | 365093519...031827456 | 26790 | 1979 | Nelson y Slowinski |
| 28 | 86 243 | 144145836...360406528 | 51924 | 1982 | Slowinski |
| 29 | 110 503 | 136204582...603862528 | 66530 | 1988 | Colquitt y Welsh |
| 30 | 132 049 | 131451295...774550016 | 79502 | 1983 | Slowinski |
| 31 | 216 091 | 278327459...840880128 | 130100 | 1985 | Slowinski |
| 32 | 756 839 | 151616570...565731328 | 455663 | 1992 | Slowinski y Gage |
| 33 | 859 433 | 838488226...416167936 | 517430 | 1994 | Slowinski y Gage |
| 34 | 1257 787 | 849732889...118704128 | 757263 | 1996 | Slowinski y Gage |
| 35 | 1398 269 | 331882354...723375616 | 841842 | 1996 | Armengaud, Woltman, et. al. (GIMPS) |
| 36 | 2976 221 | 194276425...174462976 | 1791864 | 1997 | Spence, Woltman, et. al. (GIMPS) |
| 37 | 3021 377 | 811686848...022457856 | 1819050 | 1998 | Clarkson, Woltman, Kurowski, et. al. (GIMPS) |
| 38 | 6972 593 | 955176030...123572736 | 4197919 | 1999 | Hajratwala, Woltman, Kurowski, et. al. (GIMPS) |
| 39 | 13 466 917 | 427764159...863021056 | 8107892 | 2001 | Cameron, Woltman, Kurowski, et. al. (GIMPS) |
| 40 | 20 996 011 | 793508909...206896128 | 12640858 | 2003 | Shafer, Woltman, Kurowski, et. al. (GIMPS) |
| 41 | 24 036 583 | 448233026...572950528 | 14471465 | 2004 | Findley, Woltman, Kurowski, et. al. (GIMPS) |
| 42 | 25 964 951 | 746209841...791088128 | 15632458 | 2005 | Nowak, Woltman, Kurowski, et. al. (GIMPS) |
| 43 | 30 402 457 | 497437765...164704256 | 18304103 | 2005 | Cooper, Boone, Woltman, Kurowski, et al. (GIMPS) |
| 44 | 32 582 657 | 775946855...577120256 | 19616714 | 2006 | Cooper, Boone, Woltman, Kurowski, et al. (GIMPS) |

5

3 Experimentación y Resultados

Primer Ejercicio: Primer ejercicio consistió en la realización de dos algoritmos, el primero fue la serie de Fibonacci de manera iterativa como la que se muestra en la parte de conceptos básicos la parte de su pseudocódigo, y la segunda parte fue la realización del programa pero ahora para la parte recursiva del programa, que consistió igual en hacer la serie de Fibonacci pero ahora con recursividad.

La realización del programa uno consistió en la reutilización del segundo programa que realizamos en la practica 1, usando la función de Fibonacci. Se hizo primero la declaración de la función, luego de eso se declaro las dos variables que iban a ser usadas que son a, b , que una va a ser la parte maestra del programa y la otra servirá de contador aunado, para no perder el registro del programa. una vez finalizado el programa simplemente se mostrara la cadena ya realizada y eso seria todo.

Un programa muy sencillo, pero por demás difícil si no se implementa de manera correcta y no se utilizan los contadores en la forma correcta.

```
long fibonacci(long *A, long n, long cont){  
  
    long a = 1, b = 0, i=0; (cont++);  
  
    A[0]=0;  
    A[1]=1;  
    (cont++);  
    (cont++);  
  
    for(i = 2; i < n; i+=2){  
        (cont++);  
        b = b + a; (cont++);  
        a = a + b; (cont++);  
  
        A[i]=b; (cont++);  
        A[i+1]=a; (cont++);  
    }  
    (cont++);  
  
    return cont; (cont++);  
}
```

Figura 2. Función Fibonacci Iterativa

Y para la segunda parte del problema se realizó la solución para el mismo problema de la sucesión de Fibonacci pero ahora, haciendo uso de la recursividad. El uso de la recursividad no es difícil y reduce el gasto de memoria, e inclusive el gasto de tiempo, esta manda a llamar la función reiteradas veces, permitiendo el uso de las mismas variables.

```
void fibonacciINFINITOv(long a, long b, long n){  
  
    long c; (cont++);  
  
    printf("[%d] ", b); (cont++);  
    if(n>1){  
        c = a + b; (cont++);  
        n--; (cont++);  
        fibonacciINFINITOv(b,c,n); (cont++);  
    }  
}
```

Figura 3. Función Fibonacci con recursividad

Finalmente se hizo el análisis de manera a priori que consiste de hacer el análisis de como funciona de manera teórica el sistema del algorítmico, para determinar el orden de las funciones.

| <code>void fibonacci(int *A, int n){</code> | cosntante de tiempo | mejor caso | peor caso |
|---|---------------------|------------|-----------|
| <code>int a = 1, b = 0, i=0;</code> | c1 | 1 | 1 |
| <code>A[0]=0;</code> | c2 | 1 | 1 |
| <code>A[1]=1;</code> | c3 | 1 | 1 |
| <code>for(i = 2; i < n; i+=2){</code> | c4 | n | n |
| <code> b = b + a;</code> | c5 | n-1 | n-1 |
| <code> a = a + b;</code> | c6 | n-1 | n-1 |
| <code> A[i]=b;</code> | c7 | n-1 | n-1 |
| <code> A[i+1]=a;</code> | c8 | n-1 | n-1 |
| <code> }</code> | | | |
| <code>}</code> | | | |

Figura 4. Análisis a Priori Fibonacci Iterativo

| <code>long fibonacciINFINITOv(long a, long b, long n, long cont){</code> | costo | caso | |
|--|-------|------|-----------------|
| <code>long c; (cont++);</code> | c1 | 1 | |
| <code>printf("%d ", b); (cont++);</code> | c2 | 1 | 0 si n <= 1 |
| <code>if(n>1){</code> | c3 | n-1 | f(n-1) si n > 1 |
| <code> c = a + b; (cont++);</code> | c4 | n-2 | |
| <code> n--; (cont++);</code> | c5 | n-2 | |
| <code> cont = fibonacciINFINITOv(b, c, n, cont); (cont++);</code> | c6 | n-2 | |
| <code>}</code> | | | |
| <code>return cont;</code> | c7 | 1 | |
| <code>}</code> | | | |

Figura 5. Análisis a Priori Fibonacci Recursivo.

Como se muestra en el análisis a priori el sistema tiene un orden lineal en el que en los casos sea el que sea se mantiene y comporta de la misma manera para las dos funciones tanto iterativa como la recursiva.

Ya para finalizar se realizo el análisis a posteriori del programa dando las curvas del sistema que se analiza. Como se muestra se ve que ambas curvas tanto la de recursividad y las iterativa se comporta de la misma manera exactamente.

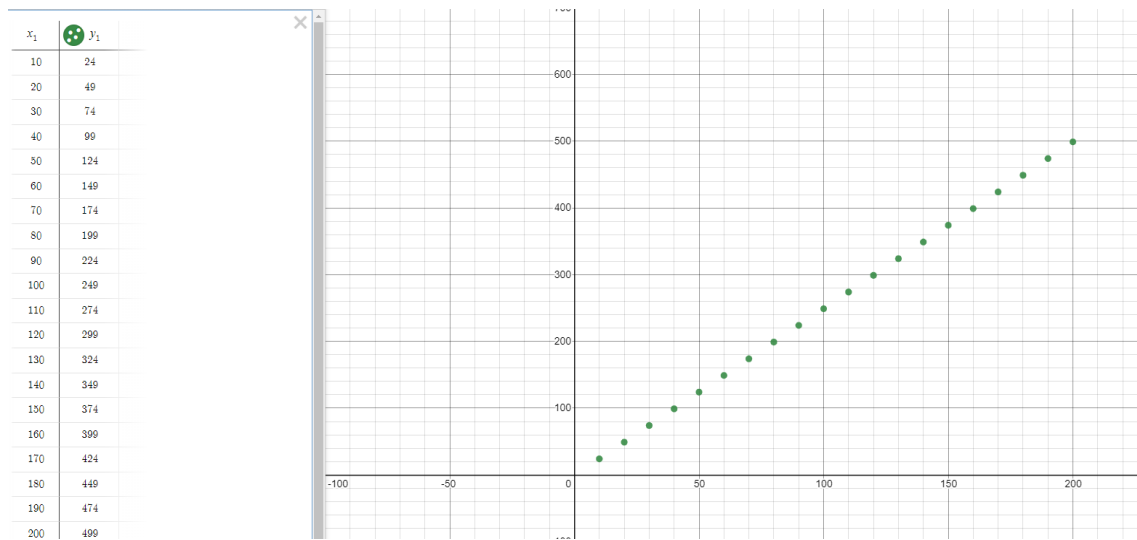


Figura 7. Curva Gráfica Fibonacci Iterativo.

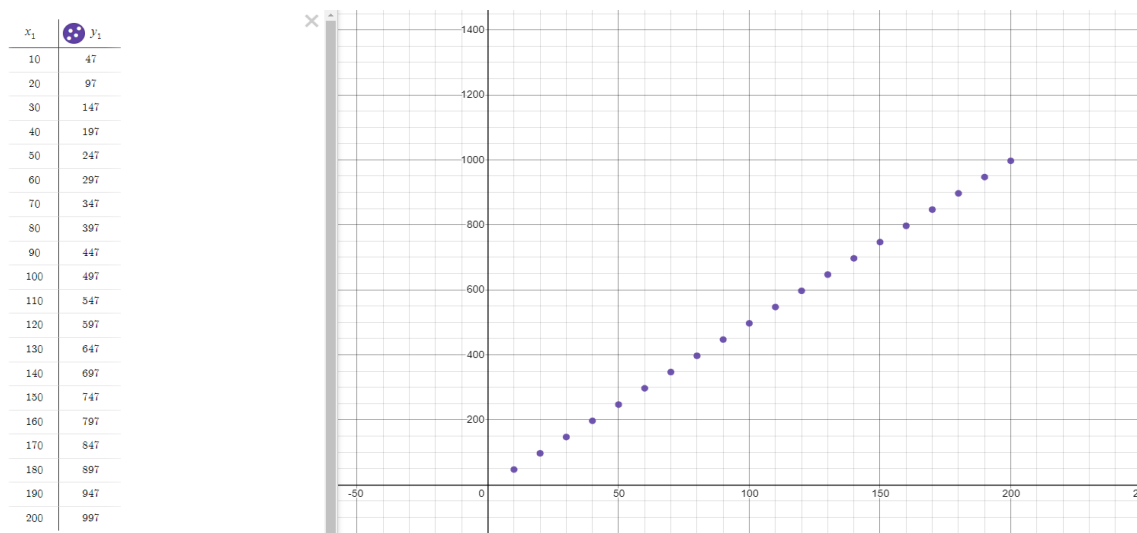


Figura 8. Curva Gráfica Fibonacci Recursivo.

Segundo Ejercicio: Para la realización de este segundo programa se hizo como primer paso el análisis a priori de de la función llamada "Perfectos" que realiza la función de decir si un numero es perfecto.

| | Constante de tiempo | Mejor Caso | Peor Caso |
|---|---------------------|------------|-----------|
| <code>int Perfectos(int n){</code> | C1 | 1 | 1 |
| <code>int factor, SumFactores;</code> | C2 | n+2 | n+2 |
| <code>for(factor=n/2 ; factor>=1 ; factor--){</code> | C3 | n+1 | n+1 |
| <code>if(n%factor == 0)</code> | C4 | n | n |
| <code>SumFactores = SumFactores + factor;</code> | | | |
| <code>}</code> | C5 | 1 | 1 |
| <code>if(SumFactores == n)</code> | C6 | 0 | 1 |
| <code>return 1;</code> | C7 | 1 | 0 |
| <code>return 0;</code> | | | |
| <code>}</code> | | | |
| <code>}</code> | | | |

Figura 9. Análisis a priori números perfectos

El la **Figura 9** se muestra el procedimiento del análisis a priori de la función Perfecto, donde sustituyendo en la ecuación para hallar orden de complejidad se tiene lo siguiente:

Para peor caso

$$T(n) = C1 + c2(n + 2) + C3(n + 1) + C4(n) + C5 + C6$$

$$T(n) = (C2 + C3 + C4)n + 2C2 + (C1 + C3 + C5 + C6)$$

$$T(n) = an + 2b + c$$

$$\therefore T(n) \in \theta(n)$$

Para mejor caso

$$T(n) = C1 + c2(n + 2) + C3(n + 1) + C4(n) + C5 + C7$$

$$T(n) = (C2 + C3 + C4)n + 2C2 + (C1 + C3 + C5 + C7)$$

$$T(n) = an + 2b + c$$

$$\therefore T(n) \in \theta(n)$$

Por lo tanto, en ambos casos tienen el mismo crecimiento asintótico, por lo que la función tiene un orden de complejidad lineal. Después se realizó el análisis a posteriori donde se comenzó implementando la función Perfectos.

```
int Perfectos(int n,int *ct){
    int factor ,SumFactores = 0;
    (*ct)++;

    (*ct)++;
    for(factor=n/2 ; factor>=1 ; factor--){
        (*ct)++;

        if(n%factor == 0){
            SumFactores = SumFactores + factor;(*ct)++;
        }
        (*ct)++;
    }
    (*ct)++;

    (*ct)++;
    if(SumFactores == n){
        (*ct)++;
        return 1;
    }

    (*ct)++;
    return 0;
}
```

Figura 10. Función Perfectos Implementada

En la **Figura 10** se muestra la implementación del algoritmo y se le agregó el contador (ct) que almacenara el número de pasos ejecutados para cada número y así traficar.

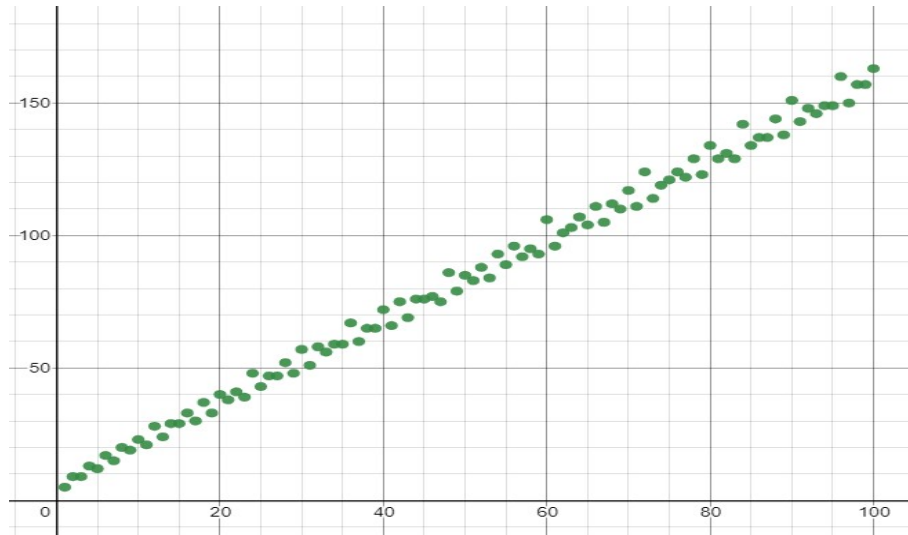


Figura 11. Gráfica de la función Perfecto

La gráfica mostrada en la **Figura 10** contiene en el eje vertical el numero de pasos y el eje horizontal el numero que se ingreso a la función, en esta se observa que efectivamente la función tiene un crecimiento lineal. Ahora por ultimo se realizo la implementación de una función llamada "MostrarPerfectos" que realizara la función de mostrar n numero perfectos, posteriormente se realizara el análisis a posteriori.

```
void MostrarPerfectos(int n,int *ct){
    int i,j; (*ct)++;

    (*ct)++;
    (*ct)++;
    for(i=1,j=n;j>=1;i++){
        (*ct)++;

        if(Perfectos(i, &(*ct))){
            j--;
            printf(" Perfecto %d \n",i);
            (*ct)++;
        }
        (*ct)++;

        (*ct)++;
    }
    (*ct)++;
}
```

Figura 12. Implementación función para mostrar perfectos

Como se puede observar en la **Figura 12** al igual que en la función pasada se coloco un contador para el numero de pasos ejecutados, pero esta hace un llamado a la función pasada llamada "Perfectos" por lo que hace que el orden de complejidad de "MostrarPerfectos" sea mayor.

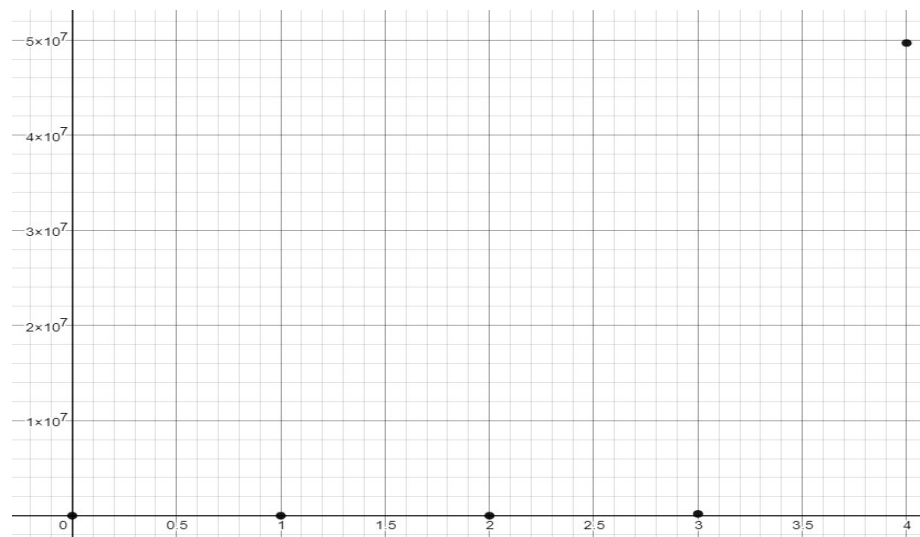


Figura 13. Gráfica de la función mostrar perfectos


| x_1 |  | y_1 |
|-------|---|----------|
| 0 | | 4 |
| 1 | | 88 |
| 2 | | 891 |
| 3 | | 191155 |
| 4 | | 49679638 |

Figura 14. Tabla de coordenadas de puntos

Tal y como se ilustra en la gráfica (**Figura 13**) solo se colocaron 4 puntos debido a que la computadora solo pudo mostrar 4 números perfectos al momento de ejecutar el programa y como se muestra en la **Figura 14** cuando

se pide mostrar 4 números el número de pasos ejecutados es de 49,679,638. Se dejó trabajando la computadora ejecutando el programa por 10 min pero no logro mostrar mas de 4 números. Debido a esto no se pudo concluir con exactitud que orden de crecimiento tiene la función.

Características PC Venegas Garcia Andres

- Procesador Intel i5-10400f a 2.9 Ghz six core
- 16 Gb de Ram
- Nvidia Geforce GTX 970

Características PC Vaquera Aguilera Ethan Emiliano

- Procesador ryzen 7-2700 a 3.2 Ghz octa core
- 16 Gb de Ram
- AMD Radeon RX 5600XT

4 Conclusiones

Conclusión General

De manera general se llegó a la conclusión que esta practica muestra uno de los casos donde un algoritmo iterativo y uno recursivo pueden tener el mismo orden de complejidad y crecimiento asintótico. Por parte del segundo ejercicio se obtuvo como conclusión que el poder de procesamiento de una computadora actual puede no ser suficiente para resolver cualquier problema implementando un algoritmo como lo es este caso, por lo que buscar un algoritmo que resuelva un problema en un número de pasos y a un tiempo menor puede llegar a ser complicado. Por ultimo, esta practica nos ayudo a observar la importancia que tiene realizar el análisis de un algoritmo para mostrarnos el tiempo que puede llegar a tardar para resolver un problema.

Conclusión Individual Vanegas García Andrés

Con los resultados dados mostrados del primer ejercicio puedo concluir que algunos algoritmos para resolver el mismo problema pueden tener el mismo orden de complejidad sin importar si el algoritmo es una versión iterativa o recursiva como lo es este caso. Por otro lado, el segundo ejercicio se me hizo un poco mas interesante ya que con la implementación del algoritmo se experimento un caso peculiar donde nuestra computadora no logro mostrar mas de 4 números debido a la gran cantidad de pasos que debe realizar, así como el tamaño de cifras de estos números perfectos y es sorprendente que solo se han descubierto 51 números hasta el momento. Además tuve unos problemas con la implementación del contador ya que al momento de hacer la gráfica los puntos no formaban una linea recta como el primer ejercicio, lo cual me generaba una confusión de si estaba realizando bien la implementación del contador o no.

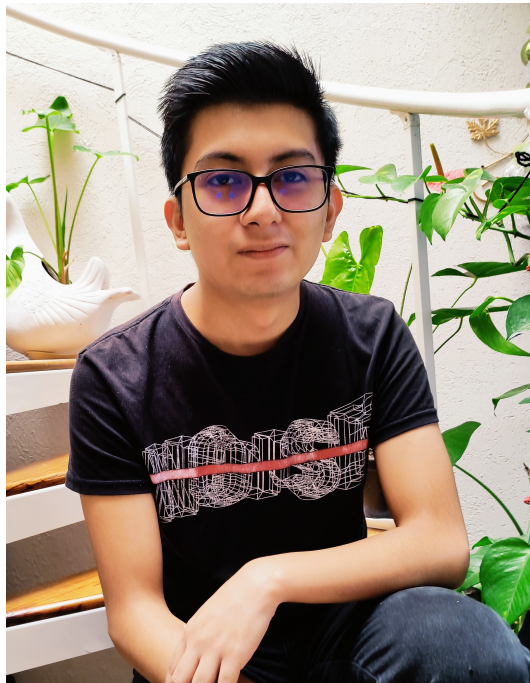


Figura 8. Vanegas García Andrés

Conclusión Individual Vaquera Aguilera Ethan Emiliano

La practica en cuestión fue interesante, ya que el caso del programa dos es sumamente interesante como se generan los números a lo largo del tiempo se ve de manera muy interesante, y es fascinante como el poder llegar a generar un algoritmo que pueda generar mas 51 números, o siquiera que pueda generar mas de 4 números (el caso de nuestras computadoras), sin tener que usar una súper computadora o una serie de paralelo secuencial. El primer programa de Fibonacci fue igualmente interesante pero mas sencillo todavía que el anterior, el complicado en cierto modo fue la serie recursiva, que a manera practica su comportamiento es muy interesante ya que apesar de usar una manera de generación muy diferente a la iterativa llega a comportarse de la misma manera.



Figura 9. Vaquera Aguilera Ethan Emiliano

5 Bibliografía

Rocio Bravo. (2021). La sucesión de Fibonacci en el Diseño. 2021, de EADE
Sitio web: <https://eade.es/blog/186-la-sucesión-de-fibonacci-en-el-diseño>

programmerclick.com. (2021). Complejidad temporal y complejidad espacial.
2021, de programmerclick.com Sitio web: <https://programmerclick.com/article/89671544092/>

Agrawal, Manindra; Kayal, Neeraj; Saxena, Nitin: "PRIMES is in P". Annals of Mathematics 160 (2004), no. 2, pp. 781–793.

