Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem.: 2022-1, 3CV11, Práctica 4, 12 noviembre 2021

Práctica 4: Divide y venceras

Vanegas García Andrés, Vaquera Aguilera Ethan Emiliano.

*avanegasg*1601@*alumno.ipn.mx*, *evaqueraa*@*alumno.ipn.mx*

Resumen: Se realizo la implementación de dos programas empleando diferentes técnicas para resolver un mismo problema, donde el primero usa un algoritmo que implementa la técnica divide y vencerás para rotar 90 grados una imagen de arte ascii y el segundo programa no implementa esta técnica. Esto se realiza con la finalidad de observar cual algoritmo logra llegar a una solución en el menor tiempo y costo posible. De esta forma se observa como el uso de la técnica divide y vencerás no siempre sígnica que un algoritmo sea mejor o peor.

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 so resolución y determinar la complejidad de un algoritmo en especifico.

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 serian las sentencias de while, for, do-while, etc.

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

Divide y vencerás: Es una técnica para diseñar algoritmos que consiste en dividir el problema general en otros subproblemas de menor tamaño y del mismo tipo. Si un subproblema sigue siendo lo demasiado grande, esta técnica se emplea repetidas veces hasta que cada subproblema sea considerablemente pequeño para ser resuelto de una forma directa.

Algoritmo de Strassen: Es un algoritmo que realiza la multiplicación de matrices cuadradas en un tiempo menor al algoritmo estándar. Este algoritmo emplea la técnica de divide y vencerás para dividir la matriz de entrada en una mas pequeña y realizar las operaciones de una forma mas eficiente.

1 Introducción

Hoy en día el uso de sistemas lógicos para la resolución de problemas en un tiempo récord es mas necesario que nunca, por ello que en el campo del diseño de algoritmos existen diferentes técnicas que se emplean tales como divide y vencerás, greedy, fuerza bruta, entre otras. En esta ocasión nos enfocaremos en la técnica de divide y vencerás, donde se analizara que tan eficiente puede ser aplicar esta técnica para un determinado problema.

Esta técnica se llama así porque sigue los siguiente pasos:

1. **Dividir:** Este paso involucra dividir el problema en subproblemas mas pequeños y cada subproblema debe representar un parte del problema general y normalmente esto supone utilizar recursividad para genera cada uno de los subproblemas

2. **Vencer:** Se debe a que cada subproblema se puede resolver de una forma mas directa y sencilla.

3. **Combinar:** Finalmente el resultado de cada problema se combina hasta formar el resultado al problema original.

Existen algoritmos muy reconocidos en el mundo de la programación que logran sobresalir a otros algoritmos por resolver un problema en un menor tiempo posible, por ejemplo, el algoritmo estándar para resolver una multiplicación de matrices tiene un orden $\Theta(n^3)$, pero el algoritmo de Strassen aplicando la técnica de divide y vencerás logro bajar el orden de complejidad a $\Theta(n^{\log 7})$, sin embargo, no significa que siempre en todo los algoritmos donde se puede aplicar esta técnica son mejore. Para mostrar un ejemplo de esto, mas adelante se muestra el análisis de dos algoritmos utilizando técnicas diferentes.

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: $\Theta(\text{g(n)}) = \{f(n) : existe\, n, c_1, c_2, > 0\, y\, n_0 > 0 \,||\, 0 \leq c_1 g(n \leq f(n) \leq c_2 g(n)\, \forall\, n \geq n_0\}$

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 da la función del peor caso, y denota los puntos experimentales de esta también.

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

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: $\Omega(g(n)) = \{f(n) : existen, c > 0\, \&\, n_0 > 0\, constantes \,||\, cg(n) \leq f(n)\, \forall\, n \geq n_0\}$

Recursividad: La recursividad es la serie de procesos definidos, en la cual se quiere la sucesión de varios factores y en vez de usar una función de la clase iterativa, se usan los mismos procesos ya definidos para crear esa sucesión de procesos, Imaginemos una muñeca de Matryoshkas (tipo de muñecas rusas), cada que abrimos una para sacar una nueva estas son mas pequeñas y además, todas y cada una de de ellas son iguales, usando ese ejemplo podemos definir un poco mejor la función recursiva, donde cada vez que mandamos a llamar a la función los valores de esta pueden ir disminuyendo o aumentando, dependiendo de lo que se quiera solucionar, hasta llegar a una condición de frontera, y de igual manera a pesar de que los valores cambien, lo que nunca se modifica son los procesos de la función recursiva que se mantiene de la misma manera.

Algorithm 1 Algoritmo Ejemplo Recursivo

```
function RECURSIVO(n)
    a ← n
    if a ≠ 0 then
        return Recursivo(a ← a − 1)
    else
        return a
    end if
end function
```

3 Experimentación y Resultados

3.1 Primera Parte

Para la primera parte de la practica se tiene que el algoritmo que soluciona el problema tiene una ecuación de forma $T(n) = 4T(n/2) + O(n^2)$, por lo que a continuación se resuelve el orden de concurrencia de la ecuación.

Aplicando Divide y Vencerás

$$T(n) = 4T(n/2) + O(n^2)$$
$$a = 4, b = 2$$
$$a \geq 1, b > 1$$
$$n^{\log_b a} = n^{\log_2 4} = n^2$$
$$Ahora$$
$$\sum_{j=1}^{K=\log_2 n} f(b^j)/a^j = \sum_{j=1}^{K=\log_2 n} = ((b^j)^2)/a^j$$
$$= \sum_{j=1}^{K=\log_2 n} 2^{2j}/4^j = \sum_{j=1}^{K=\log_2 n} 2^j/2^j = \sum_{j=1}^{K=\log_2 n} 1 = k$$
$$Juntamos$$
$$(n^2)(\log_2 n)$$
$$T(n) = O(n^2 \log_2 n)$$

Así podemos observar que el orden del algoritmo es mas menos eficiente del que se piensa que debería de tener ya que la función solo tiene un orden mas bajo, pero en palabras del profesor, no porque el algoritmo este basado en divide y vencerás, este debe de tener un orden menor al de otros algoritmos, apesara de ser eficiente este no se ve como una opción muy apta para el desarrollo de un programa mas complejo. Aquí podemos ver el programa con su correspondiente análisis a priori.

```
int rotar2(int A[][16], int n){

    fprintf(onion, "%d,%d\n", n, contg);

  n = n/2; (contg++);
  int j = 0, i = 0; (contg++);
  int aux1[n][n]; (contg++);
  int aux2[n][n];(contg++);
  int aux3[n][n];(contg++);
  int aux4[n][n];(contg++);
  int auxt[16][16];(contg++);

  if(n == 1){   //-----------------------------------------------------
    (contg++);
    for (i = 0; i < n; i++)
    {(contg++);
      for (j = 0; j < n; j++)
      {(contg++);
        aux1[i][j] = A[i][j];(contg++);
      }(contg++);
    }(contg++);

    auxt[0][0] = aux1[0][0];(contg++);
    auxt[0][1] = aux1[0][1];(contg++);
    auxt[1][0] = aux1[1][0];(contg++);
    auxt[1][1] = aux1[1][1];(contg++);

    aux1[0][0] = auxt[0][1];(contg++);
    aux1[0][1] = auxt[1][1];(contg++);
    aux1[1][0] = auxt[0][0];(contg++);
    aux1[1][1] = auxt[1][0];(contg++);

    i = 0;(contg++);
    j = 0;(contg++);

    for (int i = n; i < n+n; i++)
    {(contg++);
```

O(n)

O(n^2)

O(1)

```
38      for (int j = 0; j < n; j++)
39      {(contg++);
40        aux2[i-n][j] = A[i][j];
41      }(contg++);
42    }(contg++);
43
44    auxt[0][0] = aux2[0][0];(contg++);
45    auxt[0][1] = aux2[0][1];(contg++);
46    auxt[1][0] = aux2[1][0];(contg++);
47    auxt[1][1] = aux2[1][1];(contg++);
48
49    aux2[0][0] = auxt[0][1];(contg++);
50    aux2[0][1] = auxt[1][1];(contg++);
51    aux2[1][0] = auxt[0][0];(contg++);
52    aux2[1][1] = auxt[1][0];(contg++);
53
54    i = 0;(contg++);
55    j = 0;(contg++);
56
57    for (int i = 0; i < n; i++)
58    {(contg++);
59      for (int j = n; j < n+n; j++)
60      {(contg++);
61        aux3[i][j-n] = A[i][j];(contg++);
62      }(contg++);
63    }(contg++);
64
65    auxt[0][0] = aux3[0][0];(contg++);
66    auxt[0][1] = aux3[0][1];(contg++);
67    auxt[1][0] = aux3[1][0];(contg++);
68    auxt[1][1] = aux3[1][1];(contg++);
```

O(n^2)

O(1)

O(n^2)

O(n^2)

```
69
70    aux3[0][0] = auxt[0][1];(contg++);
71    aux3[0][1] = auxt[1][1];(contg++);
72    aux3[1][0] = auxt[0][0];(contg++);
73    aux3[1][1] = auxt[1][0];(contg++);
74
75    i = 0;(contg++);
76    j = 0;(contg++);
77
78    for (int i = n; i < n+n; i++)
79    {(contg++);
80      for (int j = n; j < n+n; j++)
81      {(contg++);
82        aux4[i-n][j-n] = A[i][j];(contg++);
83      }(contg++);
84    }(contg++);
85
86    auxt[0][0] = aux4[0][0];(contg++);
87    auxt[0][1] = aux4[0][1];(contg++);
88    auxt[1][0] = aux4[1][0];(contg++);
89    auxt[1][1] = aux4[1][1];(contg++);
90
91    aux4[0][0] = auxt[0][1];(contg++);
92    aux4[0][1] = auxt[1][1];(contg++);
93    aux4[1][0] = auxt[0][0];(contg++);
94    aux4[1][1] = auxt[1][0];(contg++);
95
96    A[0][0] = aux2[0][0];(contg++);
97    A[0][1] = aux2[0][1];(contg++);
98    A[1][0] = aux2[1][0];(contg++);
99    A[1][1] = aux2[1][1];(contg++);
100
101   A[0][2] = aux4[0][0];(contg++);
102   A[0][3] = aux4[0][1];(contg++);
103   A[1][2] = aux4[1][0];(contg++);
104   A[1][3] = aux4[1][1];(contg++);
```

O(1)

O(n^2)

O(1)

```
105
106   A[2][0] = aux1[0][0];(contg++);
107   A[2][1] = aux1[0][1];(contg++);
108   A[3][0] = aux1[1][0];(contg++);
109   A[3][1] = aux1[1][1];(contg++);
110
111   A[2][2] = aux3[0][0];(contg++);
112   A[2][3] = aux3[0][1];(contg++);
113   A[3][2] = aux3[1][0];(contg++);
114   A[3][3] = aux3[1][1]; (contg++);
115
116   /*printf("\n");
117   for (int i = 0; i < 4; i++)
118   {
119     for (int j = 0; j < 4; j++)
120     {
121       printf("%d", A[i][j]);
122     }
123     printf("\n");
124   }
125
126   printf("\nEsto es el final**********************************************\n");*/
127   (contg++);
128   return 0;
129
130 }
131 else{ //-------------------------------------------------------------------
132
133   i = j = 0;(contg++);
134
135   for (i = 0; i < n; i++) //*****************************
136   {(contg++);
137     for (j = 0; j < n; j++)
138     {(contg++);
139       aux1[i][j] = A[i][j];(contg++);
140     }(contg++);
```

O(n)

O(n^2)

```
141     }(contg++);
142
143     i = 0;(contg++);
144     j = 0;(contg++);
145
146     for (i = n; i < n+n; i++) //****************************
147     {(contg++);
148       for (j = 0; j < n; j++)
149       {(contg++);
150         aux2[i-n][j] = A[i][j];(contg++);
151       }(contg++);
152     }(contg++);
153
154     i = 0;(contg++);
155     j = 0;(contg++);
156
157     for (i = 0; i < n; i++) //****************************
158     {(contg++);
159       for (j = n; j < n+n; j++)
160       {(contg++);
161         aux3[i][j-n] = A[i][j];(contg++);
162       }(contg++);
163     }(contg++);
164
165     i = 0;(contg++);
166     j = 0;(contg++);
167
168     for (i = n; i < n+n; i++) //****************************
169     {(contg++);
170       for (j = n; j < n+n; j++)
171       {(contg++);
172         aux4[i-n][j-n] = A[i][j];(contg++);
173       }(contg++);
174     }(contg++);
175
```

O(n)

O(n^2)

O(n)

O(n^2)

O(n)

O(n^2)

O(n^2)

```
176     //*************************************************************
177
178     /*printf("Prueba-------------------------------------------------
179     for (i = 0; i < n; i++)
180     {
181       for (j = 0; j < n; j++) //-----------------------------------
182       {
183         printf("%d", aux3[i][j]);
184       }
185       printf("\n");
186     }
187     printf("\n-------------------------------------------------------
188
189     //-------------------------------------------------------------
190     rotar2(aux1, n);(contg++);
191     rotar2(aux2, n);(contg++);
192     rotar2(aux3, n);(contg++);
193     rotar2(aux4, n);(contg++);
194     //-------------------------------------------------------------
195
196     /*printf("Prueba de la UNION-------------------------------------
197     for (int i = 0; i < n; i++)
198     {
199       for (int j = 0; j < n; j++) //-------------------------------
200       {
201         printf("%d\n", aux3[i][j]);
202       }
203       printf("\n");
204     }
205     printf("\n-------------------------------------------------------
206
207     for (i = 0; i < n; i++)
208     {(contg++);
209       for (j = 0; j < n; j++) //-----------------------------
210       {(contg++);
211         A[i][j] = aux4[i][j];(contg++);
```

T(n)= 4T(n/2) + O(n^2)

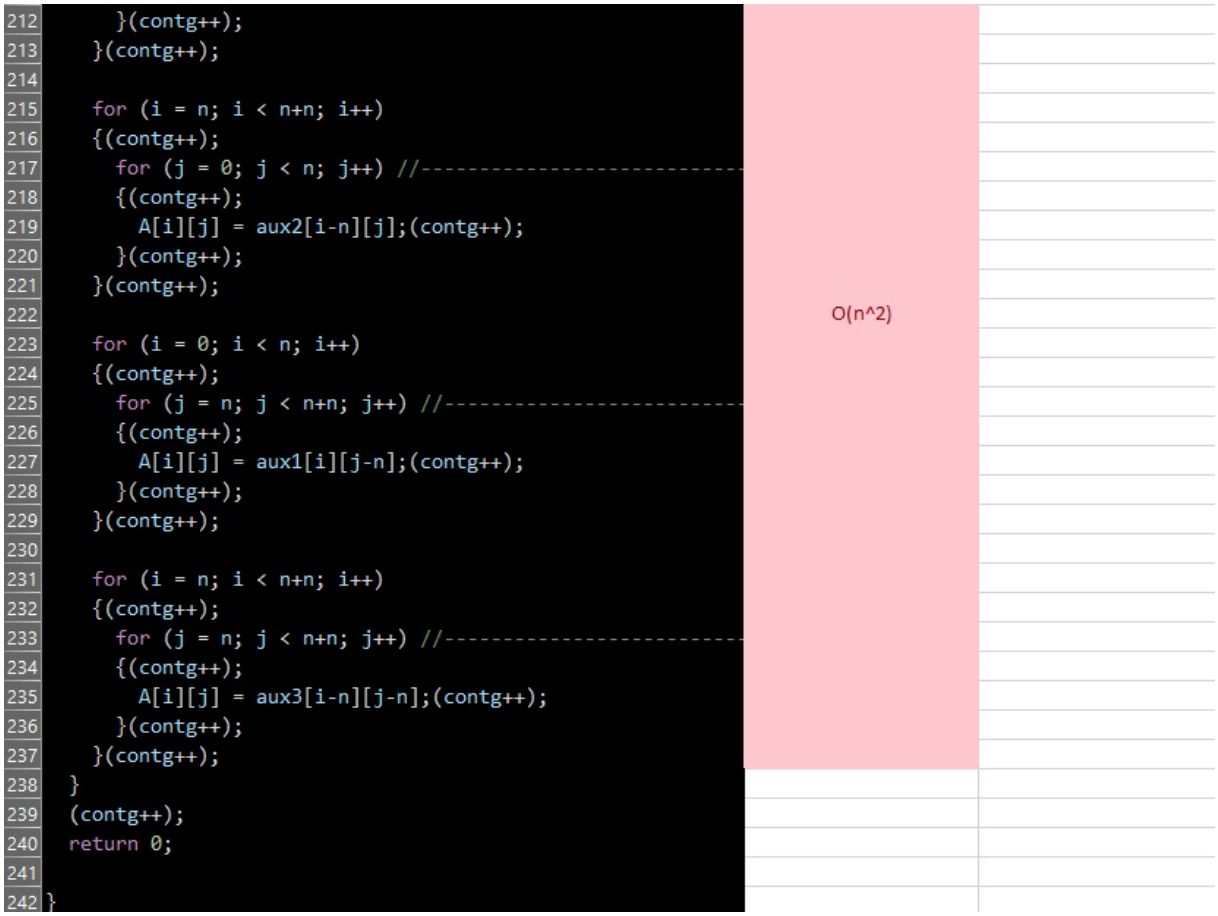
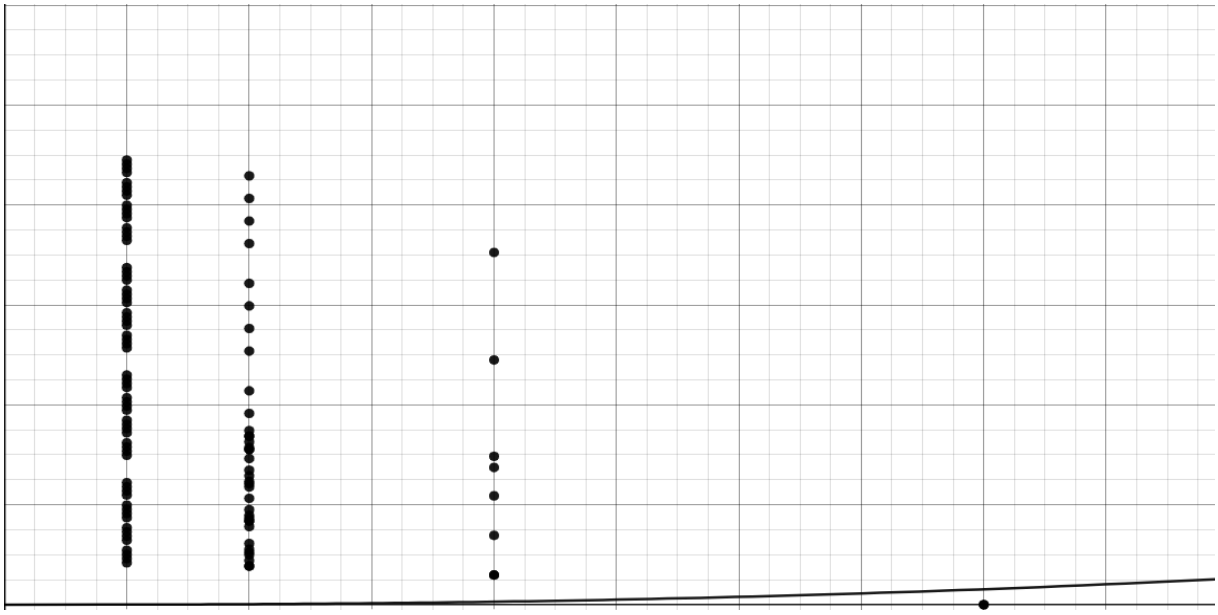


Figura 1. Código y Análisis de Algoritmo Divide y Vencerás

Una vez terminada la parte de análisis a priori, se puede empezar el análisis a posterior, siendo que empezamos ahora con la parte de ver la gráfica de que genera el algoritmo es cuestión que seria la siguiente:

Figura 2. Gráfica del comportamiento del Algoritmo



Como podemos ver en la gráfica el muestro de puntos de como el algoritmo va sucediendo las **n** muestras del programa, y ahora como estas van dividiéndose hasta llegar a ser mas concurrentes en dos donde vemos que se hacen mas operación que en los demás puntos de la muestra, y podremos ver que solo esta un punto solo en el $n = 16$.

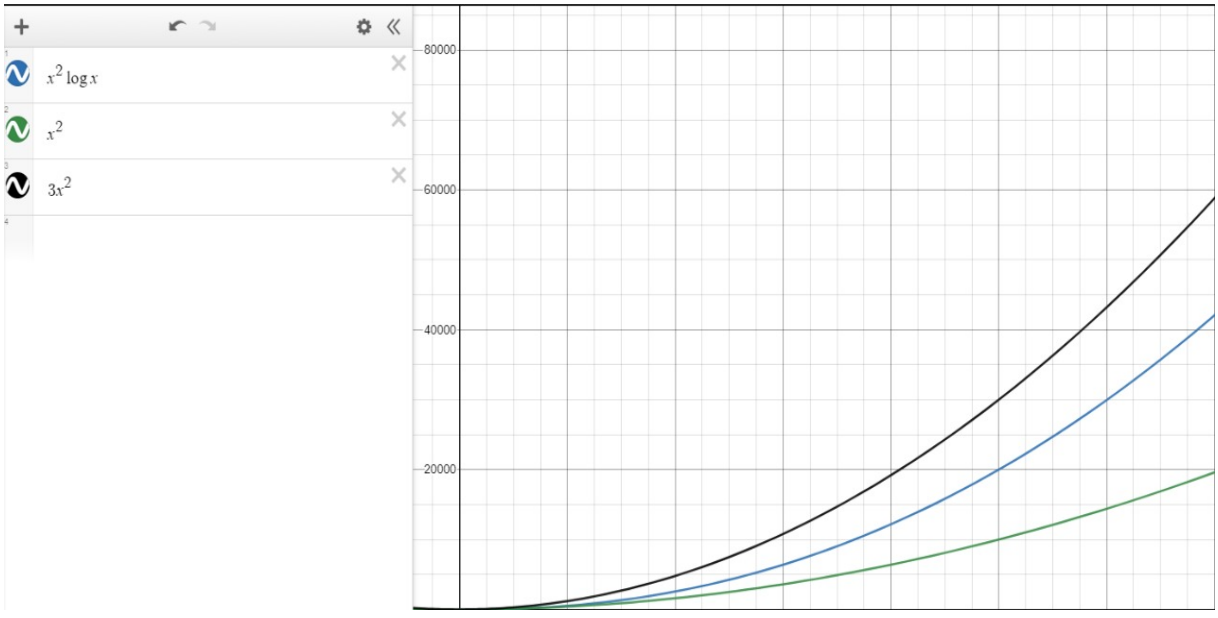


Figura 3. Gráfica de la función Divide y Vencerás

Y aquí podemos ver la función del comportamiento del Algoritmo divide y vencerás, y sus respectivas acotaciones, siendo la cuadrática por si sola la mas interesante, ya que para el segundo algoritmo podemos ver que esta seria su función característica y el orden que toma el algoritmo respectivamente.

Nota: En este algoritmo se tubo problemas de memoria al parecer por lo que algunas veces el programa funciona de manera correcta y otras veces la memoria de los arrays bidimensionales se corrompen sin razón aparente, como solución se intento el uso de una matriz mas grande para almacenar la matriz volteada pero de igual manera esta se corrompía antes de iniciarse el programa o mandarla a la función.

Por ultimo podemos ver una prueba de como funciona el programa respectivamente:

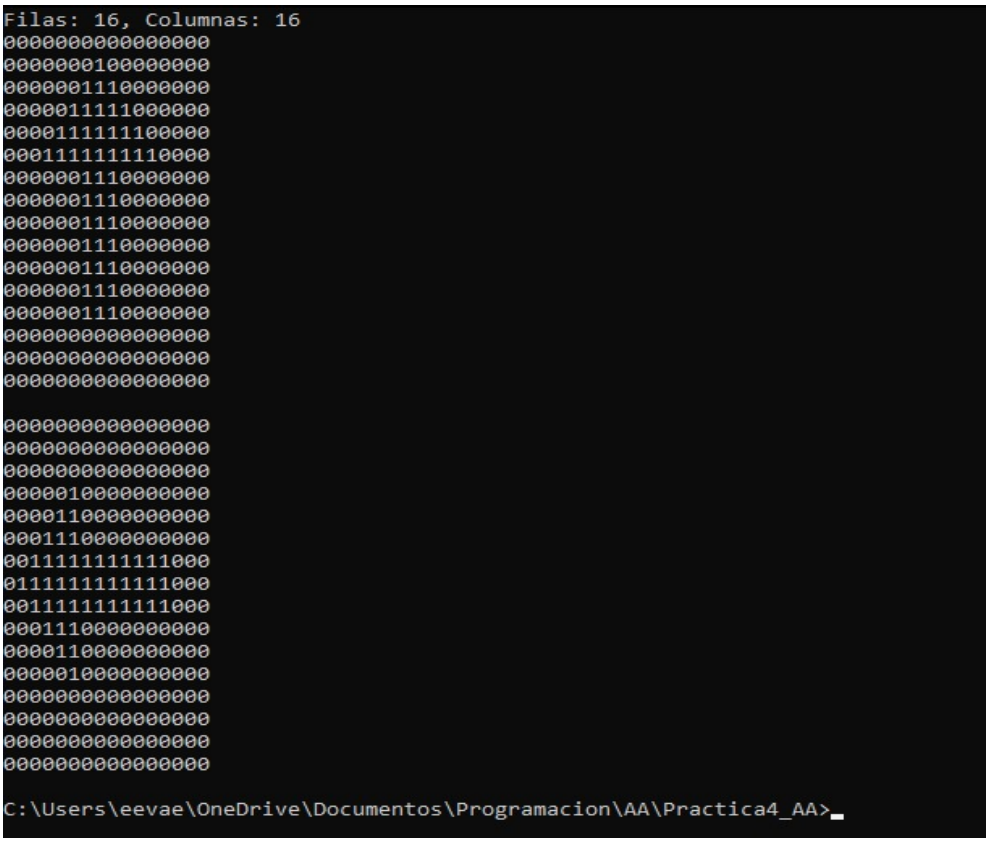


Figura 4. Funcionamiento Divide y Vencerás

3.2 Segunda Parte

Para lograr un algoritmo con el mismo objetivo pero sin implementar divide y venceras, se realizo un algoritmo que simplemente recorriera cada dato dentro de la matriz e intercambiara filas por columnas, similar a una transpuesta de una matriz.
Para realizar esto utilizamos dos ciclos for anidados, donde uno recorre las filas y otro las columnas y posteriormente se realiza el movimiento entre finas y columnas. En la siguiente figura se muestra la implementación de este algoritmo.

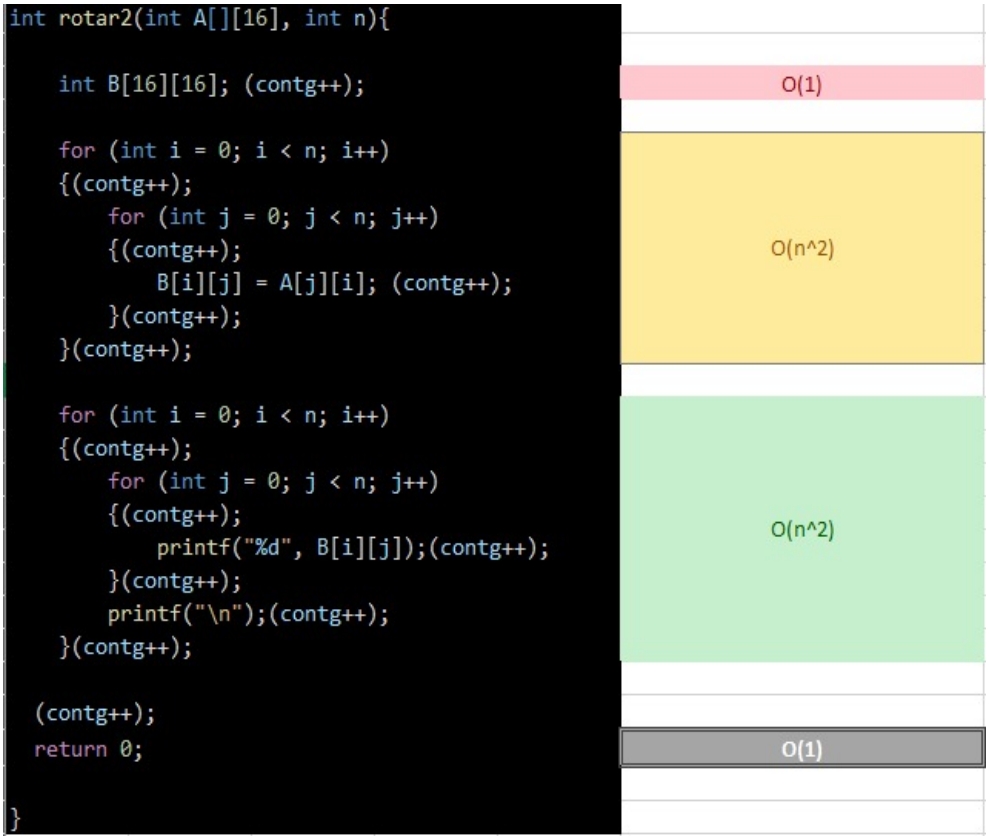


Figura 5. Función Rotar y análisis a priori

Como se observa en la **Figura 5** cada ciclo for tiene orden $O(n)$, así para cada for anidado la complejidad es de $O(n^2)$, teniendo así un algoritmo con orden de complejidad $O(n^2)$.

Posteriormente se realizó el análisis a posteriori donde implementando un contador en el codigo se obtuvo el numero de pasos ejecutados contra el tamaño n de la imagen en arte ascii.

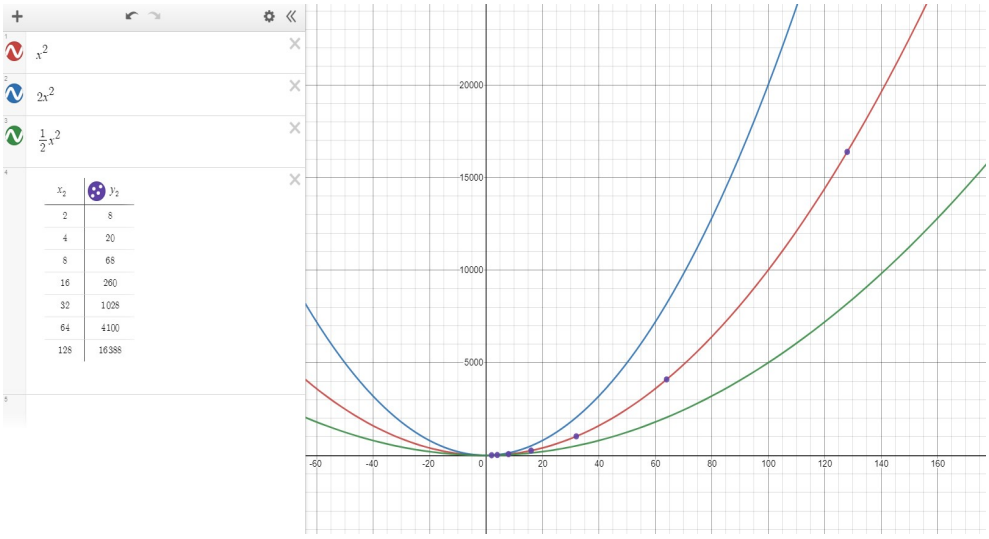


Figura 6. Gráfica análisis a posteriori

En esta gráfica podemos observar que los puntos de los pasos ejecutados tiene un comportamiento muy similar a la función x^2, y este comportamiento siempre sera igual para cualquier imagen de tamaño potencia de dos, por lo que el mejor caso y peor caso es el mismo. Para acotar esta función se propuso como cota superior $2x^2$ y como cota inferior $1/2x^2$ donde a partir del punto $x = 0$ estas cotas se cumplen.

Para finalizar, la salida de la ejecución de la implementación es exactamente la misma que en el algoritmo implementando divide y vencerás tal y como se muestra en la **Figura 4**.

Características PC Venegas Garcia Andres
-Procesador Intel i5-10400f a 2.9 Ghz six core
-16 Gb de Ram
-Nvidia Geforce GTX 970

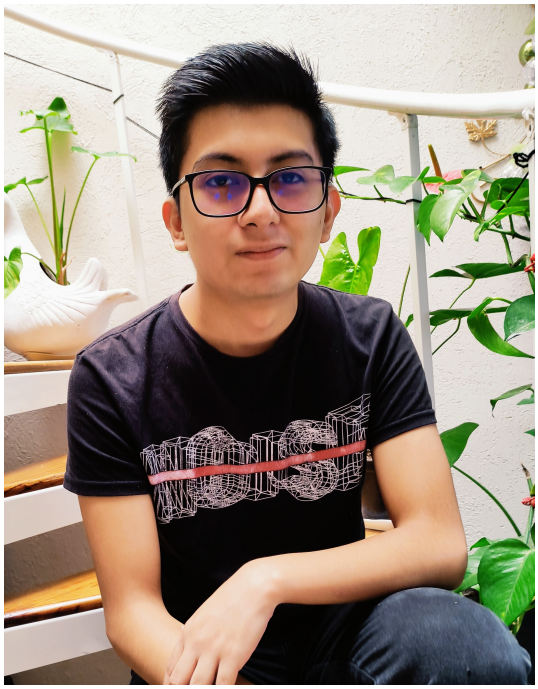
Características PC Vaquera Aguilera Ethan Emiliano
-Procesador ryzen 7-2700 a 3.2 Ghz octa core
-16 Gb de Ram
-AMD Radeon RX 5600XT

4 Conclusiones

Conclusión General: Durante el desarrollo de la practica se tuvieron diversas dificultades, tales como; implementar la tecnica divide y vencerás, entender la rotación de una matriz y unirlas. Además, se tuvo un poco de problema con el arte ascii ya que no entendíamos como se podía obtener los datos para graficar en el análisis a posteriori. Hablando del análisis, se obtuvo que no siempre la técnica divide y venderás sera la mejor para cualquier problema donde se pueda aplicar por lo que es mejor buscar otra la cual se adapte mejor a nuestro problema y obtengamos resultados óptimos y eficientes.

Conclusión Individual Vanegas García Andrés: Durante la realización de la practica se tuvo demasiados problemas con la implantación del algoritmo divide y vencerás ya que no entendíamos como podíamos llevar a cabo su uso en este problema y como se iba a realizar la rotación de la matriz. Respecto al análisis, quedo claro que el uso de una técnica para resolver un problema no siempre significa que de una solución óptima al problema, por lo que, debemos buscar una mejor forma de obtener mejores resultados en el menor tiempo.

Figura 15. Vanegas García Andrés



Conclusión Individual Vaquera Aguilera Ethan Emiliano: Para esta practica podre concluir que el funcionamiento de los programas si es muy dependiente de la manera en la que este se quiera implementar, pero no por ello puede que estos sean mejores, como podemos ver en el divide y vencerás, que a pesar de que supuestamente reduce un problema a su manera mas sencilla, este presento un comportamiento muy por mayor de la manera concurrente normal, que fue mas óptimo y fácil de implementar.

Figura 16. Vaquera Aguilera Ethan Emiliano

5 Anexo

5.1 Tarea 1:

Probar mediante substitución hacia atrás que $T(n) \in n\log(n)$
Solución: Se tiene que $T(n) = 1$ **si** $T(n) = 2T(n/2) + 1$ **si** $n > 1$
Haciendo el cambio de variable $n - 2^k(k - n\log n)$
De esta forma se tiene que $T(2^k) = 2$ **si** $k = 1$ **y** $T(2^k) = 2T(2^k - 1 + 2^k)$ **si** $n = 2^k$**, para** $k > 1$

$$T(2^k) = 2T(2^2 - 1 + 2^k)$$
$$T(2^k) = 2[2T(2^k - 1 + 2^k) + 2^k] + 2^k$$
$$T(2^k) = 4T(2^k - 2) + 2^k - 1 + 2^k$$
$$.$$
$$.$$
$$.$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^k k - i - 1k - 1 + i$$
$$T(2^k) = 2^{k-1} T(2^{k-k+1}) + (k-1)2^k$$
$$T(2^k) = 2^{k-1} T(2^1) + k2^k - 2^k$$
$$T(2^k) = (2^{k-1})(2) + k2^k - 2^k$$
$$T(2^k) = k2^k$$

Regresando el cambio de variable se tiene que :

$$T(n) = n\log(n)$$
$$\therefore T(n) \in n\log(n)$$

5.2 Tarea 2:

Utilizando decremento por uno, pruebe que $T(n) \in O(n^2)$
Solución: $T(n) = T(n-1) + c(n+1)$**.**
Así se tiene que:

$$T(n) = T(0) + \sum i = 1^n f(i)$$

Sea $f(i) = c(n+1)$**, se tiene que;**

$$T(n) = c + cn(n+1) + cn$$
$$\therefore T(n) \in O(n^2)$$

5.3 Anexo 3:

Utilizando el teorema maestro, probar que $T(n) \in \omega(n\log(n))$
Solución:

$$T(n) = 2T(n/2) + \theta(n)$$

Aplicando el Teorema tenemos que a=2, b=2 y f(n)=n

$$n^{log_b a} = n^{log_2 2} = n$$

Ya que

$$f(n) = n \in \Omega(n^{log_b a}) = \Omega(n^1 = n)$$

Aplicando el caso 2 se tiene que

$$f(n) = \Omega(n^{log_b a \log_b n})$$
$$\therefore T(n) = \Omega(n \log_2(n))$$

5.4 Anexo 4:

Multiplicación usual de números tiene orden de complejidad $O(n^2)$.
Solución:
El algoritmo usual para la multiplicacion contiene dos for donde se recorren del menos al mas significativo, así:

Algorithm 2 Algoritmo de multiplicación

```
for i=n to 0 do
    for j=n to 0 do
        Multiplicaciones y sumas
    end for
end for
```

Como los dos for tiene orden de complejidad $O(n)$ osea lineal,por el teorema estrella se tiene que el algoritmo tiene orden $O(n^2)$

$$T(n) \therefore O(n^2)$$

5.5 Anexo 5:

Probar que $T(n) \in \theta(n^2)$, donde $T(n) = 4T(n/2) + \theta(n)$
Solución:
Utilizando el teorema maestro con $a = 4, b = 2$ y $f(n) = cn$

$$n^{log_b a} = n^{log_2 4} = n^2$$

De esa forma se tiene que

$$f(n) = cn \in \theta(n^{log_b a - \epsilon}) = \theta(n^{log_2 4 - \epsilon}) = \theta(n^{2-\epsilon})$$

Aplicando el teorema maestro caso 1, se obtiene

$$f(n) = \theta(n^{\log_b a})$$
$$\therefore T(n) = \theta(n^2)$$

5.6 Anexo 6:

Probar que $T(n) \in \theta(n^{\log 3})$**, donde** $T(n) = 3T(n/3) + \theta(n)$
Solución:
Utilizando el teorema maestro se tiene que $a = 3, b = 2$ **y** $f(n) = cn$

$$n^{\log_b a} = n^{\log_2 3}$$

Se tiene que

$$f(n) = cn \in \theta(n^{\log_b a}) = \theta(n^{\log_2 3})$$

Usando el caso 2 del teorema maestro se obtiene

$$f(n) = cn \in \theta(n^{\log_b a})$$
$$\therefore T(n) = \theta(n^{\log_2 3})$$

6 Bibliografía

programmerclick.com. (2021). Complejidad temporal y complejidad espacial. 2021, de programmerclick.com Sitio web: https://programmerclick.com/article
Agrawal, Manindra; Kayal, Neeraj; Saxena, Nitin: "PRIMES is in P". Annals of Mathematics 160 (2004), no. 2, pp. 781–793.
J.M.+Gimeno+y+J.L.+González. (2021). Recursividad. 2021, de web Sitio web: http://ocw.udl.cat/enginyeria-i-arquitectura/programacio-2/continguts-1/2-recursividad.pdf
Carrasco, A. P., Iturbide, J. Á. V., Martínez, F. A. (2011). La representación de algoritmos diseñados bajo la técnica "divide y vencerás". Indagatio Didactica, 3(3), 44-68.