Tarea 4. Técnicas de Diseño de Algoritmos – Análisis de Complejidades

ANDRÉS BARRAGÁN SALAS, A01026567, ITESM

El siguiente documento muestra los algoritmos utilizado para resolver los problemas 4 y 7 presentados en el "Quiz 3. Técnicas de diseño de algoritmos". Se detalla brevemente el funcionamiento de dichos algoritmos y se muestra su programación. Posteriormente se desglosa cada uno de los códigos de programación para su análisis en cuanto a complejidad temporal y posteriormente obtener sus respectivas cotas asintóticas.

Los archivos de los códigos de programación mostrados en este documento, así como ejemplos del uso de los mismos y su ejecución se encuentran documentados y guardados en el siguiente link a un repositorio de *GitHub*:

https://github.com/tec-csf/tc2017-t3-primavera-2020-Andres-Barragan-Salas

1. LOS SUBCONJUNTOS DE SUMA DADA

1.1. Problema y algoritmo por utilizar

La pregunta 4 nos presenta el siguiente problema: *Sea W un conjunto de enteros no negativos y M un número entero positivo. El problema consiste en diseñar un algoritmo para encontrar todos los posibles subconjuntos de W cuya suma sea exactamente M.* Basándonos en la información anterior sabemos que el resultado que debe presentado es un conjunto de *n-tuplas* con las cuales se exprese cada uno de los posibles resultados. Además, al buscar todos los subconjuntos que cumplan cierta característica dentro de un conjunto W más grande estamos hablando de un recorrido en profundidad de todas las posibles combinaciones que se generan a partir de los elementos de W. Por las razones anteriores está claro que el algoritmo que resolverá el problema es de *Vuelta atrás* o *Backtracking*. Con un algoritmo de dicha naturaleza se recorrerán todos los subconjuntos del conjunto presentado, determinando su suma al ir avanzando e imprimiendo el subconjunto actual si es que se encuentra una solución. Para optimizar el algoritmo y evitar recorridos innecesarios el subconjunto W deberá estar ordenado ascendentemente, de manera que si en una actual ramificación el número en cuestión sobrepasa la suma M buscada este regrese para continuar por otro camino.

1.2 Código de programación y análisis de complejidad

A continuación, se presenta el código utilizado para implementar el algoritmo anteriormente detallado junto con todas las funciones que este utiliza para su funcionamiento. Para cada una de las líneas del código se analiza la cantidad de operaciones elementales y su forma como polinomio para el cálculo de su complejidad temporal y posteriormente su cota asintótica. La función principal en este caso se llama "*subconjuntos*" y para su llamada recursiva, en cualquiera de los casos, la variable que se ve modificada es el índice *i* actual del elemento del arreglo (conjunto) que se está analizando, esta variable siempre se incrementa en uno. De esta manera, a pesar de que realmente no se está reduciendo el número de elementos en el conjunto, siempre se recorre un elemento menos en cada llamada recursiva. Por esta razón, para el análisis de la complejidad de este código se establece que en cada llamada recursiva la cantidad de elementos se reduce en uno, siendo $T(n-1)$ y la condición de parada cuando $n = 0$ o $T(0)$ lo cual se cumple cuando $i = n$ o el índice es igual al número de elementos en el arreglo y por tanto su análisis a llegado a una conclusión.

El análisis de la complejidad temporal y cota asintótica se realizó de la siguiente manera:

Función "printStack": Función auxiliar para imprimir el contenido del subconjunto analizado en determinado momento, el cual se encuentra dentro de una pila. Es utilizada en la función principal pera evitar la redundancia de código.

```
void printStack(stack <T> s) {
        while (!s.empty()) {                    (1)
                cout << "  " << s.top();         (2)
                s.pop();                        (3)
        }
         cout<<endl;                            (4)
}
```

Línea	Código	OE	Polinomio
1	while (!s.empty()) {	1	1(n) + 1
2	cout << " " << s.top();	2	2
3	s.pop();	1	1
4	cout<<endl;	1	1
T	1 + 1(n)[2 + 1] + 1 = 1 + n(3) + 1 = 2 + 3n		

Función “subconjuntos”: Función principal que implementa el algoritmo de vuelta atrás con llamadas recursivas.

```
stack<T> s;                                                     (0)
void subconjuntos(T *w, int size, T m, T sol, int i){
        s.push(w[i]);                                           (1)

        if (sol + w[i] == m) {                                  (2)
                cout<<"\tS: ";                                  (3)
                printStack(s);                                  (4)
                s.pop();                                        (5)
                return;                                         (6)
        } else if (i == size || sol + w[i] > m) {               (7)
                s.pop();                                        (8)
                return;                                         (9)
        } else {                                                (10)
                subconjuntos(w, size, m, sol+w[i], i+1);        (11)
                s.pop();                                        (12)
        }

        if (i != size){                                         (13)
                subconjuntos(w, size, m, sol, i+1);             (14)
        }
}
```

Línea	Código	OE	Polinomio
0	stack<T> s;	1	1
1	s.push(w[i]);	2	2
Mejor caso, se encuentra una solución y regresa, no utilizado para O(n)			
2	if (sol + w[i] == m) {	3	3
3	cout<<"\tS: ";	1	1
4	printStack(s);	1	1(2 + 3n)
5	s.pop();	1	1
6	return;	1	1
Peor caso/condición de parada, nunca encuentra una solución (cuando i == size → n=0)			
7	} else if (i == size || sol + w[i] > m) {	5	5
8	s.pop();	1	1
9	return;	1	1
Llamadas recursivas (Por el índice [i] se reduce la cantidad de elementos analizados por 1)			
10	} else {	0	0
11	subconjuntos(w, size, m, sol+w[i], i+1);	1	T(n-1)
12	s.pop();	1	1
13	if (i != size){	1	1
14	subconjuntos(w, size, m, sol, i+1);	1	T(n-1)

Se obtiene la complejidad temporal para el caso base y el resto de los casos:
T(n):

- Para n=0: 1 + 2 + 3 + (0) + 5 + (1 + 1) = 13
- Para n>0: 1 + 2 + 3 + (0) + 5 + (0) + 0 + (T(n-1) + 1) + 1 + (T(n-1))
  = 12 + (T(n-1) + 1) + (T(n-1))
  = 2T(n-1) + 13

Se realiza una sucesión de complejidades a partir del caso base para encontrar un patrón.
T(0) = 13
T(1) = 2T(0) + 13 = 2(13) + 13 = 39
T(2) = 2T(1) + 13 = 2(39) + 13 = 91
T(3) = 2T(2) + 13 = 2(91) + 13 = 195
…

$\therefore \quad T(n) = (13)2^n$

Realizando un límite al infinito del polinomio anterior se obtiene que su cota asintótica es la siguiente:

$\therefore \quad O(n) = 2^n$ ■

1.3 Resultados

A partir del análisis anterior se obtuvieron los siguientes valores para el polinomio correspondiente a la complejidad temporal y su respectiva cota asintótica.

$$T(n) = 13 \cdot 2^n$$

$$O(n) = 2^n$$

2. ALGORITMO DE STRASSEN

2.1. Problema y algoritmo por utilizar

La pregunta 7 nos presenta el siguiente problema: *Si se desea implementar el algoritmo de Strassen para multiplicar dos matrices*. Para encontrar un algoritmo apto para implementar aquel de Strassen es importante primero entender el analizar el algoritmo de Strassen mismo. El algoritmo de Strassen sirve para la multiplicación de matrices cuadradas de orden *n*x*n* cuando *n* es una potencia de *2*, al tener una de estas matrices, el algoritmo se encarga de dividirla en cuatro matrices de orden (*n/2*)x(*n/2*) y realizar 7 multiplicaciones a partir de las 4 submatrices (aumentando cantidad de sumas requeridas) sustituyendo las usuales 8 multiplicaciones realizadas y por tanto reduciendo la complejidad normal de n^3. Para realizar las multiplicaciones si hace uso de las siguientes formulas tomando en cuenta dos matrices *a* y *b*:

$$m_1 = (a_{12} - a_{22})(b_{21} + b_{22})$$
$$m_2 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$m_3 = (a_{11} - a_{21})(b_{11} + b_{12})$$
$$m_4 = (a_{11} + a_{12})b_{22}$$
$$m_5 = a_{11}(b_{12} - b_{22})$$
$$m_6 = a_{22}(b_{21} - b_{11})$$
$$m_7 = (a_{21} + a_{22})b_{11}$$

Y a partir de los valores anteriores se realizan las siguientes sumas para agrupar en las posiciones de una matriz resultante *c*:

$$c_{11} = m_1 + m_2 - m_4 + m_6$$
$$c_{12} = m_4 + m_5$$
$$c_{21} = m_6 + m_7$$
$$c_{22} = m_2 - m_3 + m_5 - m_7$$

Todos los valores anteriormente expresados son submatrices de las matrices A, B, y C, y al estar reduciendo una matriz con un orden igual a una potencia de *2* en 4 submatrices eventualmente se llegará a matrices de orden *1*x*1*, en donde la operación a realizar será una simple multiplicación.

Al entender el funcionamiento del algoritmo de Strassen y saber que el problema se divide en 4 partes iguales con cada recursión del mismo es fácil distinguir que se trata de un algoritmo de *divide y vencerás*. Con un algoritmo de dicha naturaleza se utilizarán llamadas recursivas para analizar en cada recursión una cuarta parte de las matrices originales y al llegar al caso base, multiplicar e ir combinando los resultados hasta regresar al tamaño de las matrices originales.

2.2 Código de programación y análisis de complejidad

A continuación, se presenta el código utilizado para implementar el algoritmo anteriormente detallado junto con todas las funciones que este utiliza para su funcionamiento. Para cada una de las líneas del código se analiza la cantidad de operaciones elementales y su forma como polinomio para el cálculo de su complejidad temporal y posteriormente su cota asintótica. La función principal en este caso se llama "*strassen*" y para su llamada recursiva las variables que se modifican son las posiciones dentro las matrices para un análisis individual de sus secciones (en las posiciones *0, N/2* y *N*) utilizando las propiedades de apuntadores para cambiar el índice inicial de *i* y un número entero para modificar aquel de *j*. De esta manera, se reduce cada vez el orden de las matrices de *N*x*N* a (*N/2*)x(*N/2*) y eventualmente esta reducción nos llevará al caso base donde *N = 1*, y siendo esta una matriz de 1 elemento. Entonces, la operación efectuada será una simple multiplicación y a partir de ella los resultados obtenidos serán combinados en cada paso de las llamadas recursivas hasta regresar a las matrices originales.

El análisis de la complejidad temporal y cota asintótica se realizó de la siguiente manera:

Función "matrixOperation": Función auxiliar para realizar la suma entre 2 submatrices dadas de las matrices A y B. Puede realizar una suma o una resta según se necesite. Es utilizada en la función principal, pero evita la redundancia de código.

```
T** matrixOperation(T **A, int a0, T **B, int b0, int n, string operation) {
        T **C = new T*[n];                                                      (1)
        for (int i = 0; i < n; ++i)                                             (2)
                C[i] = new int[n];                                              (3)

        for (int i=0; i<n; ++i) {                                               (4)
                for (int j=0, aj=a0, bj=b0; j<n; ++j, ++aj, ++bj) {             (5)
                if (operation == "sum") {                                       (6)
                        C[i][j] = A[i][aj] + B[i][bj];                          (7)
                } else if (operation == "substract"){                           (8)
                        C[i][j] = A[i][aj] - B[i][bj];                          (9)
                } else {                                                        (10)
                        cout<<"Missing operation"<<endl;                        (11)
                }
            }
        }

        return C;                                                               (12)
}
```

Línea	Código	OE	Polinomio
1	T **C = new T*[n];	1	1
2	for (int i = 0; i < n; ++i)	3	1 + 1(n+1) + 1(n)
3	C[i] = new int[n];	2	2
4	for (int i=0; i<n; ++i) {	3	1 + 1(n+1) + 1(n)
5	for (int j=0, aj=a0, bj=b0; j<n; ++j, ++aj, ++bj) {	7	3 + 1(n+1) + 3(n)
6	if (operation == "sum") {	1	1
7	C[i][j] = A[i][aj] + B[i][bj];	8	8
8	} else if (operation == "substract"){	1	1
9	C[i][j] = A[i][aj] - B[i][bj];	8	8
10	} else {	0	0
11	cout<<"Missing operation"<<endl;	1	1
12	return C;	1	1
T	1 + 1(n+1) + 1(n) + (n)[2] + 1 + 1(n+1) + 1(n) + (n)[3 + 1(n+1) + 3(n) + (n)[1 + 8]] = 2 + n + 1 + n + 2n + n + 1 + n + (n)[3 + n + 1 + 3n + (n)[9]] = 3 + 6n + (n)[4 + 13n] = 3 + 6n + 4n + $13n^2$ = $13n^2$ + 10n + 3		

Función "strassen": Función principal que implementa el algoritmo de Strassen con llamadas recursivas

```
T** strassen(T **A, int a0, T **B, int b0, int N){
        int n = N/2;                                                    (1)
        T **C = new T*[N];                                              (2)
        for (int i = 0; i < N; ++i)                                     (3)
                C[i] = new int[N];                                      (4)
        T ***p = new T**[7];                                            (5)
        T ***submatrix = new T**[4];                                    (6)
        for (int i = 0; i < 7; ++i) {                                   (7)
                p[i] = new T*[n];                                       (8)
                submatrix[i] = new T*[n];                               (9)
                for (int j=0; j < n; ++j){                              (10)
                        p[i][j] = new int[n];                           (11)
                        submatrix[i][j] = new int[n];                   (12)
                }
        }

        if (N==1) {                                                     (13)
                C[0][0] = A[0][a0]*B[0][b0];                            (14)
                return C;                                               (15)
        } else {                                                        (16)
                p[0] = strassen(A, a0, matrixOperation(B, b0+n, B+n, b0+n, n, "substract"), 0, n);
                                                                        (17)
                p[1] = strassen(matrixOperation(A, a0, A, a0+n, n, "sum"), 0, B+n, b0+n, n);
                                                                        (18)
                p[2] = strassen(matrixOperation(A+n, a0, A+n, a0+n, n, "sum"), 0, B, b0, n);
                                                                        (19)
                p[3] = strassen(A+n, a0+n, matrixOperation(B+n, b0, B, b0, n, "substract"), 0, n);
                                                                        (20)
                p[4] = strassen(matrixOperation(A, a0, A+n, a0+n, n, "sum"), 0, matrixOperation(B, b0,
                B+n, b0+n, n, "sum"), 0, n);                            (21)
                p[5] = strassen(matrixOperation(A, a0+n, A+n, a0+n, n, "substract"), 0,
                matrixOperation(B+n, b0, B+n, b0+n, n, "sum"), 0, n);   (22)
                p[6] = strassen(matrixOperation(A, a0, A+n, a0, n, "substract"), 0, matrixOperation(B, b0,
                B, b0+n, n, "sum"), 0, n);                              (23)

                submatrix[0] = matrixOperation(matrixOperation(matrixOperation(p[4], 0, p[3], 0, n,
                "sum"), 0, p[1], 0, n, "substract"), 0, p[5], 0, n, "sum");   (24)
                submatrix[1] = matrixOperation(p[0], 0, p[1], 0, n, "sum");   (25)
                submatrix[2] = matrixOperation(p[2], 0, p[3], 0, n, "sum");   (26)
                submatrix[3] = matrixOperation(matrixOperation(matrixOperation(p[0], 0, p[4], 0, n,
                "sum"), 0, p[2], 0, n, "substract"), 0, p[6], 0, n, "substract");   (27)
        }

        int j0, j1, j2, j3;                                             (28)
        for (int i=0; i<N; ++i) {                                       (29)
                j0=0, j1=0, j2=0, j3=0;                                 (30)
                for (int j=0; j<N; ++j) {                               (31)
```

```
                    if (i<n && j<n) {                                   (32)
                        C[i][j] = submatrix[0][i][j0];                  (33)
                        j0++;                                           (34)
                    } else if (i<n && j>=n) {                           (35)
                        C[i][j] = submatrix[1][i][j1];                  (36)
                        j1++;                                           (37)
                    } else if (i>=n && j<n) {                           (38)
                        C[i][j] = submatrix[2][i-n][j2];                (39)
                        j2++;                                           (40)
                    } else if (i>=n && j>=n) {                          (41)
                        C[i][j] = submatrix[3][i-n][j3];                (42)
                        j3++;                                           (43)
                    }
                }
            }

            return C;                                                   (44)
}
```

Línea	Código	OE	Polinomio
1	int n = N/2;	2	2
2	T **C = new T*[N];	1	1
3	for (int i = 0; i < N; ++i)	3	1 + 1(n+1) + 1(n)
4	C[i] = new int[N];	2	2
5	T ***p = new T**[7];	1	1
6	T ***submatrix = new T**[4];	1	1
7	for (int i = 0; i < 7; ++i) {	3	1 + 1(8) + 1(7)
8	p[i] = new T*[n];	2	2
9	submatrix[i] = new T*[n];	2	2
10	for (int j=0; j < n; ++j){	3	1 + 1(n+1) + 1(n)
11	p[i][j] = new int[n];	3	3
12	submatrix[i][j] = new int[n];	3	3
Caso base, las matrices ingresadas solo contienen 1 elemento (n==1)			
13	if (N==1) {	1	1
14	C[0][0] = A[0][a0]*B[0][b0];	8	8
15	return C;	1	1
Llamadas recursivas con matrices de n/2 elementos (mientras n>1)			
16	} else {	0	0
17	p[0] = strassen(A, a0, matrixOperation(B, b0+n, B+n, b0+n, n, "substract"), 0, n);	4	$2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
18	p[1] = strassen(matrixOperation(A, a0, A, a0+n, n, "sum"), 0, B+n, b0+n, n);	4	$2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
19	p[2] = strassen(matrixOperation(A+n, a0, A+n, a0+n, n, "sum"), 0, B, b0, n);	4	$2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
20	p[3] = strassen(A+n, a0+n, matrixOperation(B+n, b0, B, b0, n, "substract"), 0, n);	4	$2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$

21	p[4] = strassen(matrixOperation(A, a0, A+n, a0+n, n, "sum"), 0, matrixOperation(B, b0, B+n, b0+n, n, "sum"), 0, n);	5	$2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
22	p[5] = strassen(matrixOperation(A, a0+n, A+n, a0+n, n, "substract"), 0, matrixOperation(B+n, b0, B+n, b0+n, n, "sum"), 0, n);	5	$2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
23	p[6] = strassen(matrixOperation(A, a0, A+n, a0, n, "substract"), 0, matrixOperation(B, b0, B, b0+n, n, "sum"), 0, n);	5	$2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2)$
24	submatrix[0] = matrixOperation(matrixOperation(matrixOperation(p[4], 0, p[3], 0, n, "sum"), 0, p[1], 0, n, "substract"), 0, p[5], 0, n, "sum");	5	$2 + 3(13(n/2)^2 + 10(n/2) + 3)$
25	submatrix[1] = matrixOperation(p[0], 0, p[1], 0, n, "sum");	3	$2 + 3(13(n/2)^2 + 10(n/2) + 3)$
26	submatrix[2] = matrixOperation(p[2], 0, p[3], 0, n, "sum");	3	$2 + 3(13(n/2)^2 + 10(n/2) + 3)$
27	submatrix[3] = matrixOperation(matrixOperation(matrixOperation(p[0], 0, p[4], 0, n, "sum"), 0, p[2], 0, n, "substract"), 0, p[6], 0, n, "substract");	5	$2 + 3(13(n/2)^2 + 10(n/2) + 3)$
28	int j0, j1, j2, j3;	0	0
29	for (int i=0; i<N; ++i) {	3	1 + 1(n+1) + 1(n)
30	j0=0, j1=0, j2=0, j3=0;	4	4
31	for (int j=0; j<N; ++j) {	3	1 + 1(n+1) + 1(n)
Todas las posibilidades existirán y se ejecutaran, se tomará el caso con complejidad más alta.			
32	if (i<n && j<n) {	3	3
33	C[i][j] = submatrix[0][i][j0];	6	6
34	j0++;	1	1
35	} else if (i<n && j>=n) {	3	3
36	C[i][j] = submatrix[1][i][j1];	6	6
37	j1++;	1	1
38	} else if (i>=n && j<n) {	3	3
39	C[i][j] = submatrix[2][i-n][j2];	7	7
40	j2++;	1	1
41	} else if (i>=n && j>=n) {	3	3
42	C[i][j] = submatrix[3][i-n][j3];	7	7
43	j3++;	1	1
44	return C;	1	1

Se obtiene la complejidad temporal para el caso base y el resto de los casos:

T(n):

- Para n=1: 2 + 1 + 1 + 1(n+1) + 1(n) + (n)[2] + 1 + 1 + 1 + 1(8) + 1(7) + (7)[2 + 2 + 1 + 1(n+1) + 1(n) + (n)[3 + 3]] + 1 + [8 + 1]
  = 4 + n + 1 + n + 2n +18 + (7)[5 + n + 1 + n + 6n] + 1 + 9
  = 4 + n + 1 + n + 2n + 18 + 35 + 7n + 7 + 7n + 42n + 1 + 9
  = 60n + 75

= 60(1) + 75
= 135

T(1) = 135

- Para n>1: 2 + 1 + 1 + 1(n+1) + 1(n) + (n)[2] + 1 + 1 + 1 + 1(8) + 1(7) + (7)[2 + 2 + 1 + 1(n+1) + 1(n) + (n)[3 + 3]] + 1 + 0 + [2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 1(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 2(13(n/2)^2 + 10(n/2) + 3) + 1T(n/2) + 2 + 3(13(n/2)^2 + 10(n/2) + 3) + 2 + 1(13(n/2)^2 + 10(n/2) + 3) + 2 + 1(13(n/2)^2 + 10(n/2) + 3) + 2 + 3(13(n/2)^2 + 10(n/2) + 3)] + 0 + 1 + 1(n+1) + 1(n) + (n)[4 + 1 + 1(n+1) + 1(n) + (n)[3 + 7 + 1]] + 1
  = 4 + n + 1 + n + 2n +18 + (7)[5 + n + 1 + n + 6n] + 1 + 22 + 18((13/4)n^2 + 5n + 3) + 7T(n/2) + 1 + n + 1 + n + (n)[5 + n + 1 + n + (n)[11]] + 1
  = 4 + n + 1 + n + 2n + 18 + 35 + 7n + 7 + 7n + 42n + 23 + (117/2)n^2 + 90n + 54 + 7T(n/2) + 1 + n + 1 + n + (n)[13n + 6] + 1
  = 4 + n + 1 + n + 2n + 18 + 35 + 7n + 7 + 7n + 42n + 23 + (117/2)n^2 + 90n + 54 + 7T(n/2) + 1 + n + 1 + n + 13n^2 + 6n + 1
  = 7T(n/2) + (143/2)n^2 + 158n + 145

  T(n) = 7T(n/2) + (143/2)n^2 + 158n + 145

Se expande la recurrencia para encontrar patrones:

T(n) = 7T(n/2) + $\frac{143}{2}$n^2 + 158n + 145

T(n) = 7(7T(n/4) + $\frac{143}{2}$ (n/2)^2 + 158(n/2) + 145) + $\frac{143}{2}$n^2 + 158n + 145
= 49T(n/4) + $\frac{1001}{8}$n^2 + 553n + 1015 + $\frac{143}{2}$n^2 + 158n + 145
= 49T(n/4) + $\frac{143}{2}$n^2 + $\frac{1001}{8}$n^2 + 158n + 553n + 145 + 1015

T(n) = 49(7T(n/8) + $\frac{143}{2}$(n/4)^2 + 158(n/4) + 145) + $\frac{143}{2}$n^2 + $\frac{1001}{8}$n^2 + 158n + 553n + 145 + 1015
= 343T(n/8) + $\frac{7007}{32}n^2$ + $\frac{3871}{2}n$ + 7105 + $\frac{143}{2}$n^2 + $\frac{1001}{8}$n^2 + 158n + 553n + 145 + 1015
= 343T(n/8) + $\frac{143}{2}$n^2 + $\frac{1001}{8}$n^2 + $\frac{7007}{32}n^2$ + 158n + 553n + $\frac{3871}{2}n$ + 145 + 1015 + 7105

…

Si el proceso de expansión se repite *k* veces:

T(n) = 7^kT($\frac{n}{2^k}$) + ($\frac{143}{2}\sum_{i=1}^{k}\frac{7^{i-1}}{2^{(i-1)(2)}}$)$n^2$ + (158 $\sum_{i=1}^{k}\frac{7^{i-1}}{2^{i-1}}$)$n$ + (145 $\sum_{i=1}^{k}7^{i-1}$)

Cuando se cumpla que $2^k = n$ se utilizará el caso base, entonces:

$$2^k = n \quad \rightarrow \quad \log(2^k) = \log(n) \quad \rightarrow \quad k\log(2) = \log(n) \quad \rightarrow \quad k = \frac{\log(n)}{\log(2)}$$

$$\therefore k = \log_2(n)$$

$T(n) = 7^{\log_2(n)}T(1) + (\frac{143}{2}\sum_{i=1}^{\log_2 n}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158\sum_{i=1}^{\log_2 n}\frac{7^{i-1}}{2^{i-1}})n + (145\sum_{i=1}^{\log_2 n}7^{i-1})$

$T(n) = (135)7^{\log_2(n)} + (\frac{143}{2}\sum_{i=1}^{\log_2 n}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158\sum_{i=1}^{\log_2 n}\frac{7^{i-1}}{2^{i-1}})n + (145\sum_{i=1}^{\log_2 n}7^{i-1})$

Obtenemos los valores de la sumatoria con exponentes mayores para su consideración en la cota asintótica:

$T(n) = (135)7^{\log_2(n)} + (\frac{143}{2})(\frac{7^{\log_2 n-1}}{2^{(\log_2 n-1)(2)}})(n^2) + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158)(\frac{7^{\log_2 n-1}}{2^{(\log_2 n-1)}})(n) +$
$(158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (145)(7^{\log_2 n-1}) + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

$T(n) = (135)7^{\log_2(n)} + (\frac{143}{2})(\frac{\frac{7^{\log_2 n}}{7}}{\left(\frac{2^{\log_2 n}}{2}\right)^2})(n^2) + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158)(\frac{\frac{7^{\log_2 n}}{7}}{\frac{2^{\log_2 n}}{2}})(n) +$
$(158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (145)(\frac{7^{\log_2 n}}{7}) + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

Trabajando con los exponentes logarítmicos resultantes:

$7^{\log_2(n)} \rightarrow 2^{\log_2(7)\log_2(n)} \rightarrow 2^{\log_2(n)\log_2(7)} \rightarrow n^{\log_2(7)}$

Y similarmente:

$2^{\log_2(n)} \rightarrow n^{\log_2(2)} \rightarrow n$

De esta manera obtenemos que:

$T(n) = (135)n^{\log_2(7)} + (\frac{143}{2})(\frac{4}{7})(\frac{n^{\log_2(7)}}{n^2})(n^2) + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158)(\frac{2}{7})(\frac{n^{\log_2(7)}}{n})(n) +$
$(158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (\frac{145}{7})(n^{\log_2(7)}) + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

$T(n) = (135)n^{\log_2(7)} + (\frac{286}{7})n^{\log_2(7)} + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (\frac{316}{7})n^{\log_2(7)} + (158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n +$
$(\frac{145}{7})n^{\log_2(7)} + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

$T(n) = (\frac{1692}{7})n^{\log_2(7)} + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

Una vez que se han tomado todos los valores posibles con el exponente más grande no vale la pena analizar el resto de la sumatoria pues no es relevante para calcular la cota asintótica, obteniendo que:

$\therefore \quad T(n) \approx (\frac{1692}{7})n^{2.807} + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$

Realizando un límite al infinito del polinomio anterior se obtiene que su cota asintótica es la siguiente:

$\therefore \quad O(n) = n^{2.807}$ ■

2.3 Resultados

A partir del análisis anterior se obtuvieron los siguientes valores para el polinomio correspondiente a la complejidad temporal y su respectiva cota asintótica.

$$T(n) \approx (\frac{1692}{7})n^{2.807} + (\frac{143}{2}\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{(i-1)(2)}})n^2 + (158\sum_{i=1}^{\log_2 n-1}\frac{7^{i-1}}{2^{i-1}})n + (145\sum_{i=1}^{\log_2 n-1}7^{i-1})$$

$$O(n) = n^{2.807}$$

3. CONCLUSIÓN

Después de la realización de los códigos de ambos programas uno puede darse cuenta de que resuelven los problemas en una manera fácil de entender mediante el uso de le recursión. Algoritmos como el de Strassen que ayudan a reducir la complejidad de los algoritmos usuales, aunque no significativamente, ayudan cuando se están tratando de grandes cantidades de elementos. La programación de los mismos no es complicada, sin embargo, al estar analizando su complejidad posteriormente uno puede darse cuenta de ciertas modificaciones que se podrían realizar al código para reducir la cantidad de operaciones elementales utilizadas, sin embargo, la complejidad *BigO* usualmente siempre tenderá hacia la misma cota asintótica si es que se sigue el mismo planteamiento de un algoritmo, sin importar la implementación utilizada.

4. REFERENCIAS

[1] Guerequeta, R., Vallecido, A. (1998) "Técnicas de Diseño de Algoritmos" [PDF] Servicio de Publicaciones de la Universidad de Malaga. Obtenido de: http://www.lcc.uma.es/~av/Libro/

[2] Cubells, V. (2020). "Complejidad y Notación Asintótica" [Power Point] (Presentación de Clase).

[3] Cubells, V. (2020). "Análisis de Algoritmos Recursivos" [Power Point] (Presentación de Clase).