Práctica 3: Divide y vencerás

Hernández Castellanos César Uriel, Aguilar Garcia Mauricio

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*uuriel*12009*u*@*gmail.com*, *mauricio.aguilar.garcia*,90@*gmail.com*

Resumen: Se implementa el algoritmo mergesort el cual utiliza la técnica "divide y vencerás que es el de descomponer en pequeños problemas y resolverlos individualmente".

Palabras Clave: Algoritmo, Complejidad, MergeSort, Divide, Vencerás.

1. Introducción

En la cultura popular, divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas.

En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (quicksort, mergesort, entre muchos otros), multiplicar números grandes (Karatsuba), análisis sintácticos (análisis sintáctico top-down) y la transformada discreta de Fourier.

Por otra parte, analizar y diseñar algoritmos de DyV son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y no hay un método sistemático de generalización.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada (o su equivalente en computación numérica, el algoritmo de bisección para búsqueda de raíces). Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de “divide y vencerás”; en particular, si es usando una serie de recursiones que lo convierten en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de “divide

y vencerás". El nombre decrementa y vencerás ha sido propuesta para la subclase simple de problemas.

La corrección de un algoritmo de "divide y vencerás", está habitualmente probada una inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.[1]

2. Conceptos Básicos

2.1. Cota ajustada asintótica

En análisis de algoritmos una cota ajustada asintótica es una función que sirve de cota tanto superior como inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\theta(g(x))$ para referirse a las funciones acotadas por la función $g(x)$.[2]

2.2. Cota inferior asintótica

En análisis de algoritmos una cota inferior asintótica es una función que sirve de cota inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\Omega(g(x))$ para referirse a las funciones acotadas inferiormente por la función $g(x)$.[2]

2.3. Cota superior asintótica

En análisis de algoritmos una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación de Landau: $O(g(x))$, Orden de $g(x)$, coloquialmente llamada Notación O Grande, para referirse a las funciones acotadas superiormente por la función $g(x)$.[2]

2.4. Algoritmos

2.4.1. MergeSort

Fue desarrollado en 1945 por John Von Neumann.

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso: Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla. Mezclar las dos sublistas en una sola lista ordenada. El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

Una lista pequeña necesitará menos pasos para ordenarse que una lista grande. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas. A continuación se describe el algoritmo en pseudocódigo (se advierte de que no se incluyen casos especiales para vectores vacíos, etc.; una implementación en un lenguaje de programación real debería tener en cuenta estos detalles):

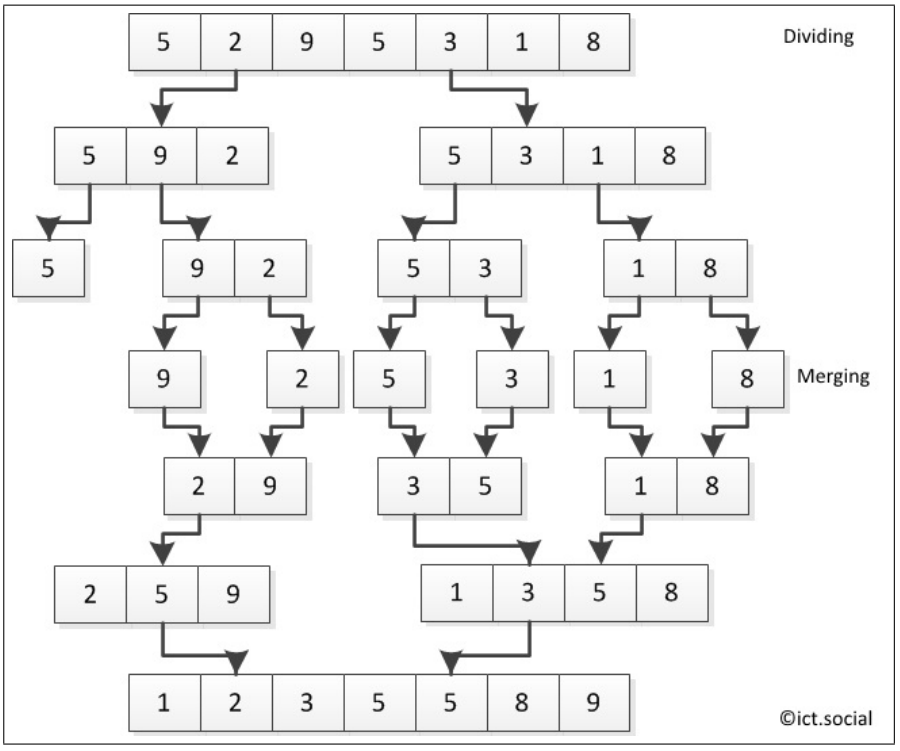

Figura 1: Algoritmo de MergeSort

3. Experimentación y Resultados

3.1. MergeSort

(i). Mediante gráficas, muestre que el algoritmo Merge tiene complejidad lineal: Se tiene el algoritmo implementado del Merge:

Como salida del código tenemos a la n del tamaño de la entrada y la t la cual es el contador del algoritmo.

Si graficamos estos valores obtenemos la siguiente gráfica la cual nos demuestra que tiene complejidad lineal.

(ii). Demuestre analíticamente que el algoritmo Merge tiene complejidad lineal. Encontrando las sumatorias del algoritmo de Merge:

```
Algoritmo Merge(A,p,q,r)
  Input: Un arreglo A[p,1,2,...,r] con p <= q <=r
  Output:
      n1 = q-p+1                    //C1 = 1
      n2 = r-q

      Sean L[0,1,...,n1-1] y Q[0,1,...,n2-1] 2 arreglos
      for i=0 hasta i < n1          //C2 = n1 + 1
          L[i] = A[p+i]             //C3 = sum(0,n1-1,1)
```

```
void MergeSort(int A[],int p,int r){
    int q;
    cont++;
    if(p<r){
        q=(p+r)/2;
        MergeSort(A,p,q);
        MergeSort(A,q+1,r);
        Merge(A,p,q,r);
    }
}

//Algoritmo Merge implementado

void Merge(int A[],int p, int q, int r){
    int n1 = q-p+1;
    int n2 = r-q;
    int L[n1],R[n2],i,j;
    for(i=0; i<n1; i++){
        L[i] = A[p+i];
    }
    for(j=0; j<n2; j++)
        R[j] = A[q+1+j];
    i=0;
    j=0;
    for(int k=p; k<=r; k++ ){
        cont++;
        if(L[i]<=R[j]){
            A[k]=L[i];
            i++;
        }else{
            A[k]=R[j];
            j++;
        }
    }
}
```

Figura 2: Código de algoritmo Merge

```
2: 2
4: 4
8: 8
16: 16
32: 32
64: 64
128: 128
256: 256
512: 512
1024: 1024
```

Figura 3: Salida del programa Merge

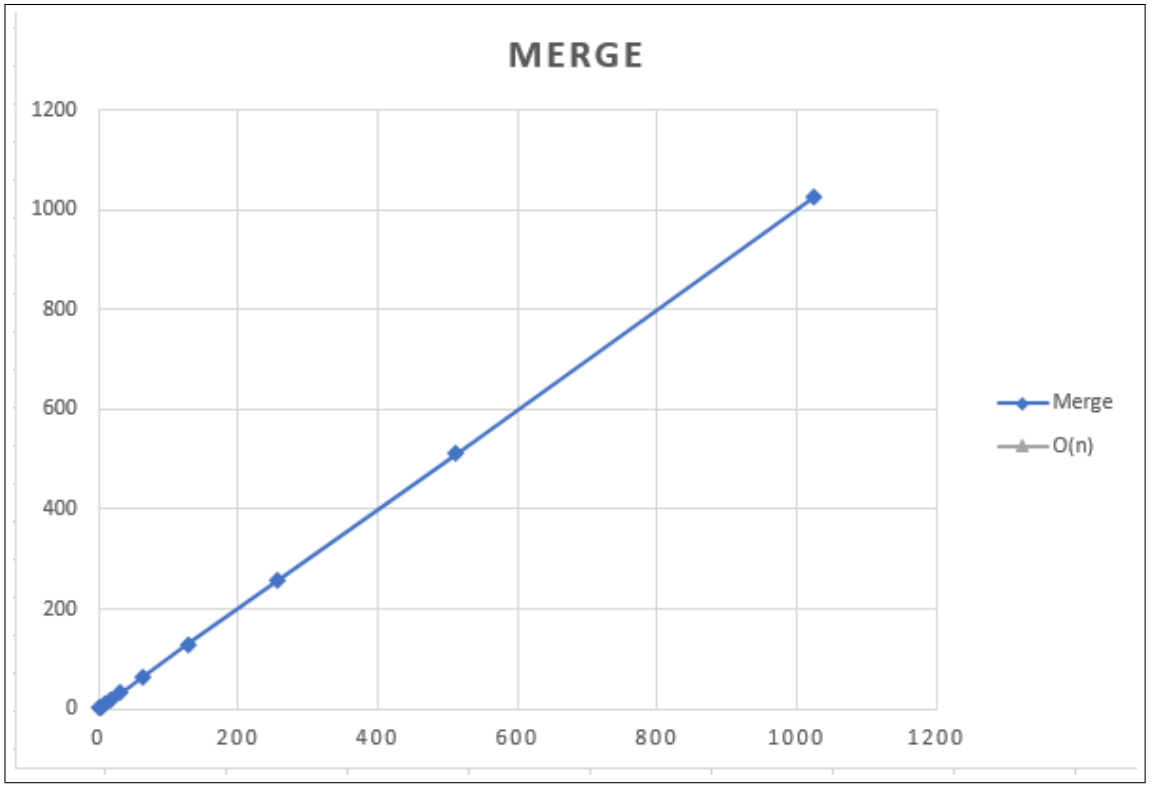

Figura 4: Salida del programa Merge

```
for i=0 hasta i < n2          //C4 = n2 + 1
    Q[i] = A[q+1+i]           //C5 = sum(0,n2-1,1)
i=0,j=0

for k=p to k<= r              //C6 = r-p+1
    if(i<n1)                  //C7 = sum(p,r,1)
        if(j<n2)              //C8 = sum(p,r,1)
            if(L[i]<=Q[j])    //C9 = sum(p,r,1)
                A[k] = L[i]
                i++;
            else
                A[k] = Q[j]
                j++;
        else
            A[k] = L[i]
            i++;
    else
        A[k] = Q[j]
        j++;
```

Donde $sum(a, b, c)$ representa a la sumatoria desde a hasta b de c.
Utilizamos los cn para calcular la complejidad del algoritmo de Merge.

$$T(n) = C1 + C2(n1+1) + C3\left[\sum_{i=0}^{n1-1}(1)\right] + C4(n2+1) + C5\left[\sum_{i=0}^{n2-1}(1)\right] + C6(r-p+2) + (C7+C8)\left[\sum_{p}^{r}1\right] + (C9)\left[\sum_{p}^{r-1}1\right]$$

$$T(n) = C1 + C2(n1+1) + C3[n1] + C4(n2+1) + C5[n2] + C6(n+1) + (C7+C8)[n] + (C9)[n-1]$$

$$T(n) = (C6+C7+C8+C9)n\ (C2+C3)[n1] + (C4+C5)n2 + C1 + C2 + C4 + C6 - C9$$

$$T(n) = 4n + 2[n1] + 2[n2] + 3$$

$$n1 = q-p+1 \quad n2 = r-q \longrightarrow (n1+n2) = r-q+1 = n$$

$$T(n) = 4n + 2[n1+n2] + 3 = \mathbf{6n+3}$$

$$\boldsymbol{T(n)\ \epsilon\ \theta(n)}$$

(iii)Mediante gráficas, muestre que el algoritmo MergeSort tiene complejidad $\theta(nlogn)$.
Se tiene el algoritmo implementado del MergeSort:

Como salida del código tenemos a la n del tamaño de la entrada y la t la cual es el contador del algoritmo.

Si graficamos estos valores obtenemos la siguiente gráfica la cual nos demuestra que tiene complejidad $\theta(nlogn)$, puesto que aunque la gráfica de la salida del programa se encuentre por debajo de la de $nlogn$ se puede notar que si sigue la misma función.
(iv)Demuestre analíticamente que el algoritmo MergerSort tiene complejidad $\theta(nlogn)$.

```
Algoritmo MergeSort(A,p,r)
  Input: Un arreglo A[p,1,2,...,r] con p <= r
  Output: Un arreglo A[p,1,2,...,r] ordenado de manera creciente
```

```
2: 5
4: 15
8: 39
16: 95
32: 223
64: 511
128: 1151
256: 2559
512: 5631
1024: 12287
```

Figura 5: Salida del programa MergeSort

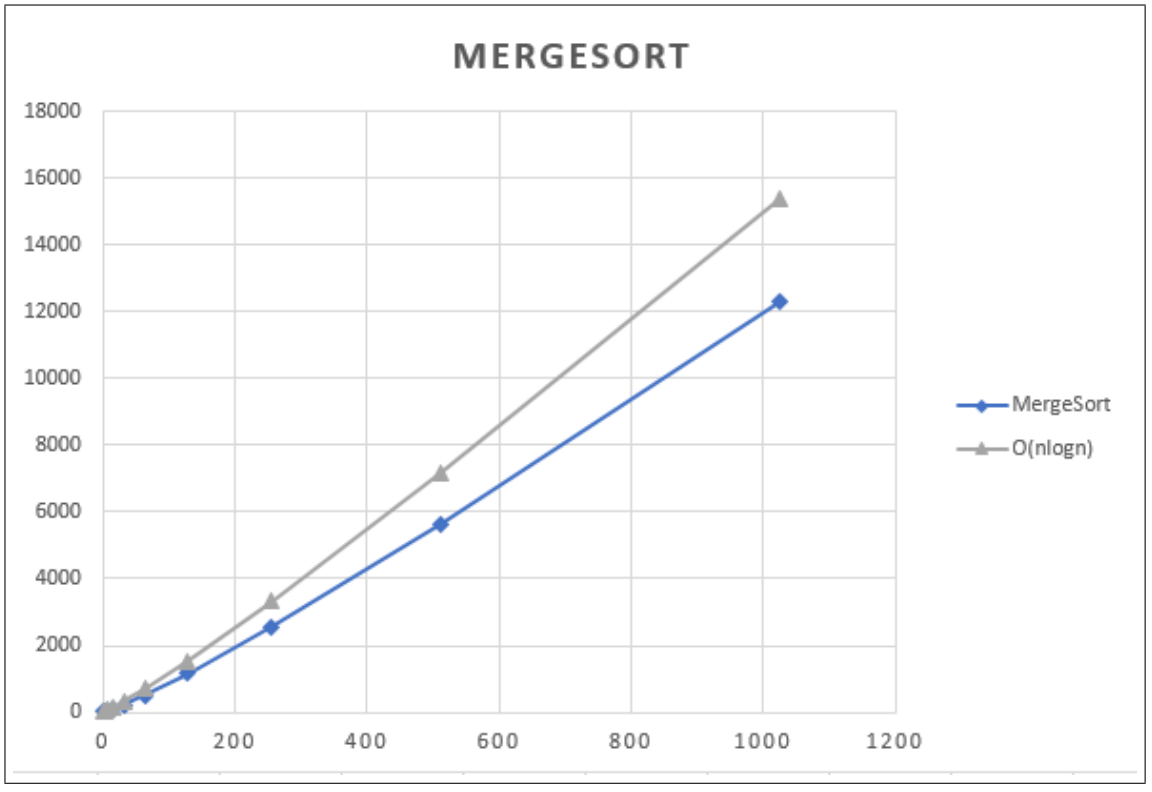

Figura 6: Gráfica Merge

```
if(p<r)
      q = (p+r)/2          // 1
 MergeSort(A,p,q)          //  2T(n/2)
 MergeSort(A,q+1,r)        //  2T(n/2)
 Merge(A,p,q,r)            //  M(n) = complejidad del Merge
```

Utilizamos los cn para calcular la complejidad del algoritmo de MergeSort.

$$T(n) = \begin{cases} \theta(1); & si\ n = 1 \\ 2\left[T\left(\frac{n}{2}\right)\right] + M(n) + 1; & si\ n > 1 \end{cases}$$

$$T(n) = \begin{cases} \theta(1); & si\ n = 1 \\ 2\left[T\left(\frac{n}{2}\right)\right] + 6n + 4; & si\ n > 1 \end{cases}$$

$k = log_2(n)$

$\boldsymbol{i}$	$\boldsymbol{T(n) = T(2^k)}$
1	$2[2[T(2^{k-2})] + 6(2^{k-1}) + 4] + 6(2^k) + 4$
	$2^2[T(2^{k-2})] + 12(2^k) + 3(4)$
2	$2^2[2[T(2^{k-3})] + 6(2^{k-2}) + 4] + 12(2^k) + 3(4)$
	$2^3[T(2^{k-3})] + 18(2^k) + 7(4)$
i	$2^{i+1}[T(2^{k-(i+1)})] + 6(i+1)(2^k) + (2^{i+1} - 1)4$

$$k - (i+1) = 0 \qquad \therefore i = k - 1$$

$$T(2^k) = 2^k[T(2^0)] + 6(k)(2^k) + (2^k - 1)4$$

$$T(1) = 0, 2^k = n\ y\ \ k = log_2(n)$$

$$T(n) = n[1] + 6(log_2(n))(n) + (n-1)4$$

$$T(n) = n[5 + 6log_2(n)] - 4$$

$$\therefore\ \boldsymbol{\theta\left(nlog_2(n)\right)}$$

4. Conclusiones

Aguilar Garcia Mauricio

La estrategia de divide y venceras nos ayuda a pensar que los problemas los podemos resolver por partes las cuales son mas sencillas ya que es mejor pensar llegar a un lado y de ahi resolverlo de .^{eq}uis"manera que pensar desde el inicio como reolverlo ya que si pierdes un poco la logica en el caso que lo tengas completo se puede llegar a que inicias denuevo todo el problema pero en divide y venceras simplifica tanto la complejidad como la logica para programarlo.

Hernández Castellanos César Uriel

Divide y vencerás nos provee una estrategia de solución de problemas, ya que es posible dividir el problema de forma tal que nos ayuda a mejorar la solución a diferentes problemas.

Divide y vencerás nos proporciona en ocasiones la mejora del nivel de complejidad de diferentes algoritmos.

Conclusión general

En esta práctica pudimos verificar que hay algoritmos que pueden o no tener la misma complejidad dependiendo de diversos factores, como puede ser el orden de un arreglo, el tamaño de un arreglo, si un número es muy grande, etc. Por ejemplo, el MergeSort es un algoritmo que invariablemente de como esté ordenado el arreglo al entrar, siempre va a tener la misma complejidad, en cambio en la Funcion 2, dependiendo en donde este el elemento máximo y el elemento mínimo el algoritmo puede ser n o n^2.

5. Anexo

Calcular el orden de complejidad de los siguientes algoritmos en el mejor caso y en el peor de los casos (No es necesario hacer el análisis linea por linea, en este caso, pueden aplicar propiedades de los algoritmos vistos en clase)

5.1. Función uno

Algorithm 1 Función uno

```
1: procedure FuncionUno(n par)
2:
3:     i ← 0
4:     while i<n do
5:         for i ← 1, 10 do
6:             Acción(i)
7:             j++
8:         end for
9:         i = i + 2
10:    end while
11: end procedure
12:
```

5.2. Función dos

Algorithm 2 Función uno

```
1: procedure FuncionDos(A[0, ..., n − 1], x entero)
2:
3:     for i ← 0, n do
4:         if A[i]<x then
5:             A[i] ← min(A[0, ..., n − 1])
6:         else if bitOne || bitTwo then
7:             A[i] ← max(A[0, ..., n − 1])
8:         else
9:             exit
10:        end if
11:    end for
12: end procedure
13:
```

5.3. Función uno implementada

```
1
2   def main():
3       FuncionUno(20)
4
5   if __name__ == '__main__':
6       main()
7
8   def FuncionUno(n):
9       i = 0
10      accion = 0
11      while i < n:
12          for j in range(1,11):
13              accion = i
14          i += 2
15
```

Figura 7: Función uno implementada en el lenguaje de programación Python

5.4. Función dos implementada

```
1   def main():
2       arreglo = [6,7,8,10,324,12,3,4,3,23,432,232,121]
3       x = 0
4       Funcion(arreglo,x)
5       print arreglo
6       while(1):
7           x=10
8
9   if __name__ == '__main__':
10      main()
11
12  def Funcion(arreglo, x):
13      for i in range(0,len(arreglo)):
14          if(arreglo[i] < x):
15              arreglo[i] = min(arreglo)
16          elif(arreglo[i] > x):
17              arreglo[i] = max(arreglo)
18          else:
19              break
20
```

Figura 8: Función dos implementada en el lenguaje de programación Python

5.5. Cálculo de la complejidad de la función uno

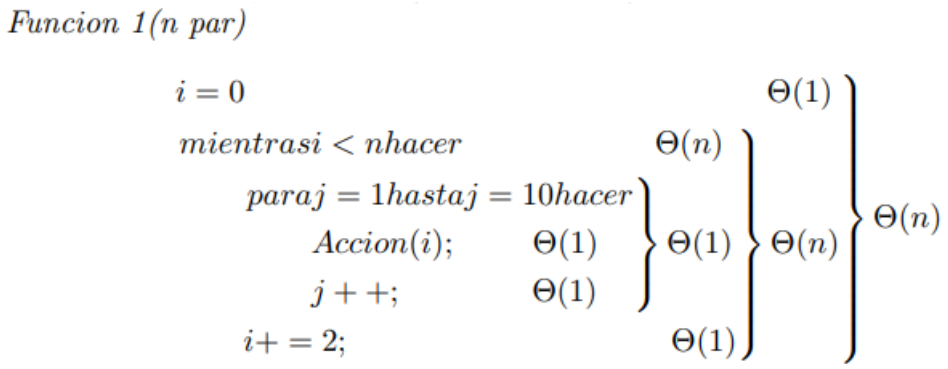

Figura 9: Uso de los teoremas para el cálculo de la complejidad del algoritmo

El for interno es constante independientemente del valor de n, se ejecutará 10 veces siempre por lo que $\Theta(1)$. Y también se aprecia que el ciclo while se ejecutará n/2 veces, por lo que es $\Theta(n)$, por lo que es posible concluir que en el mejor caso y peor caso la complejidad del algoritmo será lineal.

5.6. Cálculo de la complejidad de la función dos

Haciendo uso de los teoremas vistos en clase es posible observar que el mejor caso es cuando el número máximo y mínimo son el primer elemento, por lo que únicamente se ejecutará en una ocasión el ciclo.

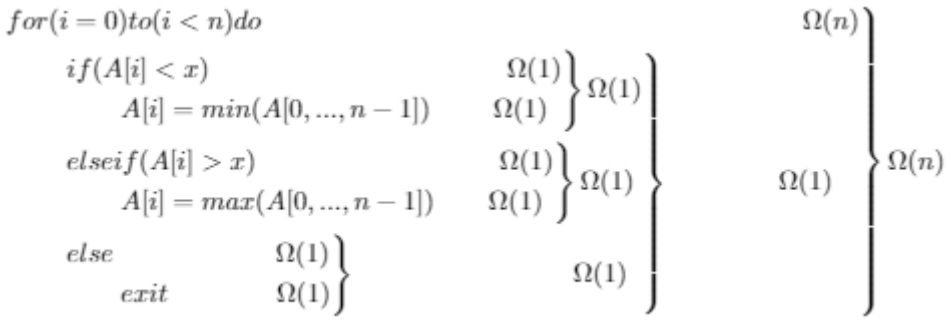

Figura 10: Uso de los teoremas para el cálculo de la complejidad del algoritmo

De manera similar calculamos el peor de los casos que se presenta cuando el número máximo y minimo son el último elemento por lo que tiene que recorrer todo el arreglo
Esto nos quiere decir que la función dos tiene complejidad lineal y cuadrática en su mejor y peor caso respectivamente.

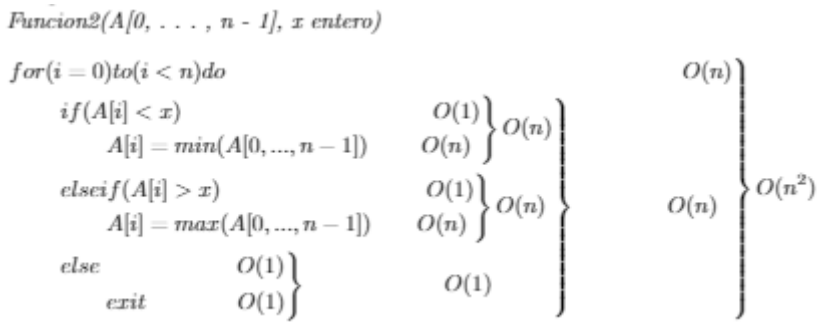

Figura 11: Uso de los teoremas para el cálculo de la complejidad del algoritmo

6. Bibliografía

Referencias

[1] 'Algoritmo divide y vencerás' Disponible en https://es.wikipedia.org/wiki/Algoritmo_divide_y_vencerás// consultado 12 de Septiembre del 2017

[2] Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein