Instituto Politécnico Nacional
Escuela Superior de Cómputo

Analisis de algoritmos, Sem: 2021-1, 3CV1,Práctica 3, 11/11/2020

Práctica 4: Divide y Vencéras

Payán Téllez René
rpayant1500@alumno.ipn.mx

Resumen: En esta practica se analizaran 2 algoritmos que utilizan la tecnica de divide y venceras para resolver un problema, uno de ellos el quick sort y otro el sub arreglo maximo.
Palabras clave: QuickSort,SubArreglo maximo,divide y venceras,C/C++

1 Introduccion

El saber implementar tecnicas de divide y venceras es fundamental para el desarrollo de algoritmos computacionales, ya que esta tecnica de resolver problemas abarca una enorme cantidad de algoritmos que resuelven problemas extremadamente complejos, o vitales para el funcionamiento de un sistema. En esta practica analizaremos a profundiad 2 de ellos QuickSort y MaxSubArray, tambien veremos como algunos de estos son afectados por las condiciones del problema y los compararemos con soluciones mas sencillas pero menos optimas.

2 Conceptos Basicos

2.1 Algoritmo

La palabra algoritmo proviene del sobrenombre de un matemático árabe del siglo IX, Al-Khwarizmi, que fue reconocido por enunciar paso a paso las reglas para las operaciones matemáticas básicas con decimales (suma, resta, multiplicación y división). Vemos definición de algoritmo como un grupo de órdenes consecutivas que presentan una solución a un problema o tarea. Algunos ejemplos de algoritmos los podemos encontrar en las matemáticas (como el algoritmo para resolver una multiplicación) y en los manuales de usuario de un aparato (como una lavadora o una impresora). Sin embargo, hoy en día se relaciona la palabra algoritmo con el mundo de la informática, más concretamente en la programación; los conocidos como algoritmos informáticos.[1]

2.2 Divide y venceras

El término Divide y Vencerás en su acepción más amplia es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos. En nuestro contexto, Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de del mismo tipo, pero de menor tamaño. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos. La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos: 1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño

nk y donde $0 \leq nk \leq n$. A esta tarea se le conoce como división. 2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base. 3. Por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.[2]

QuickSort

Algoritmo 1: Partition A[p,...,r]

Data: Entrada: A[p,...,r]
Result: Retorna la posicion del pivote i, un elemento que cumple con la propiedad de que todos los elementos desde p hasta i-1 son menores que A[i] y que de i+1 hasta r, todos los elementos son mayores que A[i]

```
x=A[r];
i=p-1;
for j ← p to j ≤ r − 1 do
    if A[j] ≤ x then
        i++;
        exchange(A[i],A[j]);
exchange(A[i+1],A[r]);
return i+1;
```

Algoritmo 2: QuickSort A[p,...,n]

Data: Entrada: A[p,...,n]
Result: Retorna un areglo ordenado de forma ascendente

```
if p < n then
    q=Partition(A,p,n);
    QuickSort(A,p,q-1);
    QuickSort(A,q+1,n);
```

Problema del maximo subarreglo

Algoritmo 3: MaxCrossingSubArray(A[0,...,n-1],bajo,mitad,alto)

```
Data: Entrada: A[0,...,n-1], bajo, mitad, alto
Result: Retorna la suma del mayor sub arreglo de la izquierda, la derecha y juntos
suma_izq=INT_MIN;
suma=0;
max_izq=0;
for i ← mitad to bajo do
    suma+=A[i];
    if suma > suma_izq then
        suma_izq=suma;
        max_izq=i;
suma_der=INT_MIN;
suma=0;
max_der=0;
for j ← mitad to alto do
    suma+=A[j];
    if suma > suma_der then
        suma_der=suma;
        max_der=j;
return (max_izq,max_der,suma_der+suma_izq);
```

Algoritmo 4: MaxSubArrayDC(A[0,...,n-1],bajo,alto)

```
Data: Entrada: A[0,...,n-1], bajo, alto
Result: retorna la suma y los indices del mayor sub arreglo, que se puede obtener dentro
        del arreglo A
if alto == bajo then
    return(bajo,alto,A[bajo]);
else
    mitad=(bajo+alto)/2;
    (bajo_izq,alto_izq,suma_izq)=MaxSubArrayDC(A,bajo,mitad);
    (bajo_der,alto_der,suma_der)=MaxSubArrayDC(A,mitad+1,alto);
    (cruz_izq,cruz_der,suma_cruz)=MaxCrossingSubArray(A,bajo,mitad,alto);
    if suma_izq > suma_der and suma_izq > suma_cruz then
        return (bajo_izq,alto_izq,suma_izq);
    else if suma_der > suma_izq and suma_der > suma_cruz then
        return (bajo_der,alto_der,suma_der);
    else
        return (cruz_izq,cruz_der,suma_cruz);
return (max_izq,max_der,suma_der+suma_izq);
```

Algoritmo 5: FuerzaBruta(A[0,...,n-1])

```
Data: Entrada: A[0,...,n-1]
Result: Retorna la suma y los indices del mayor sub arreglo, que se puede obtener dentro
        del arreglo A
sumaMaxima=−∞;
indiceIzquierdo = 0;
indiceDerecho = 0;
for i ← 0 to n do
    sumaLocal=0;
    for j ← i to n do
        sumaLocal+=A[j];
        if sumaLocal>sumaMaxima then
            sumaMaxima = sumaLocal;
            indiceIzquierdo=i;
            indiceDerecho=j;
return(sumaMaxima,indiceIzquierdo,indiceDerecho);
```

Algoritmo de Karatsuba

El algoritmo de Karatsuba es un algoritmo de multiplicación rápida . Fue descubierto por Anatoly Karatsuba en 1960 y publicado en 1962. Reduce la multiplicación de dos números de n dígitos a un máximo de multiplicaciones de un solo dígito en general (y exactamente cuando n es una potencia de 2). Por tanto, es más rápido que el algoritmo tradicional , que requiere productos de un solo dígito. Por ejemplo, el algoritmo de Karatsuba requiere 3 10 = 59,049 multiplicaciones de un solo dígito para multiplicar dos números de 1024 dígitos (n = 1024 = 2 10), mientras que el algoritmo tradicional requiere (2 10) 2 = 1,048,576 (una aceleración de 17,75 veces).$n^{\log_2 3} \approx n^{1.58}$. El algoritmo de Karatsuba fue el primer algoritmo de multiplicación asintóticamente más rápido que el algoritmo cuadrático de "escuela primaria". El algoritmo de Toom-Cook (1963) es una generalización más rápida del método de Karatsuba, y el algoritmo de Schönhage-Strassen (1971) es incluso más rápido, para n suficientemente grande.[3]

Algoritmo 6: Karatsuba(num_1,num_2)

```
Data: Entrada: num1, num2
Result: retorna la multiplicacion de num1 y num2
if num1 <10 or num2 <10 then
    return num1*num2;
m = min(longitud(num1),longitud(num2));
m2 = floor(m/2);
alto1, bajo1 = cortar(num1,m2);
alto2, bajo2 = cortar(num2,m2);
z0 = Karatsuba(bajo1, bajo2);
z1 = Karatsuba((bajo1 + alto1),(bajo2 + alto2));
z2 = Karatsuba(alto1, alto2);
return (z2 ∗ 10^(2m2)+(z1-z2-z0)*10^m2+z0)
```

3 Experimentacion y Resultados

3.1 Implementar el algoritmo QuickSort.

Se ejecuto el programa para generar arreglos aleatorios de tamaño n (0→100000) y ordenarlos mediante QuickSort.

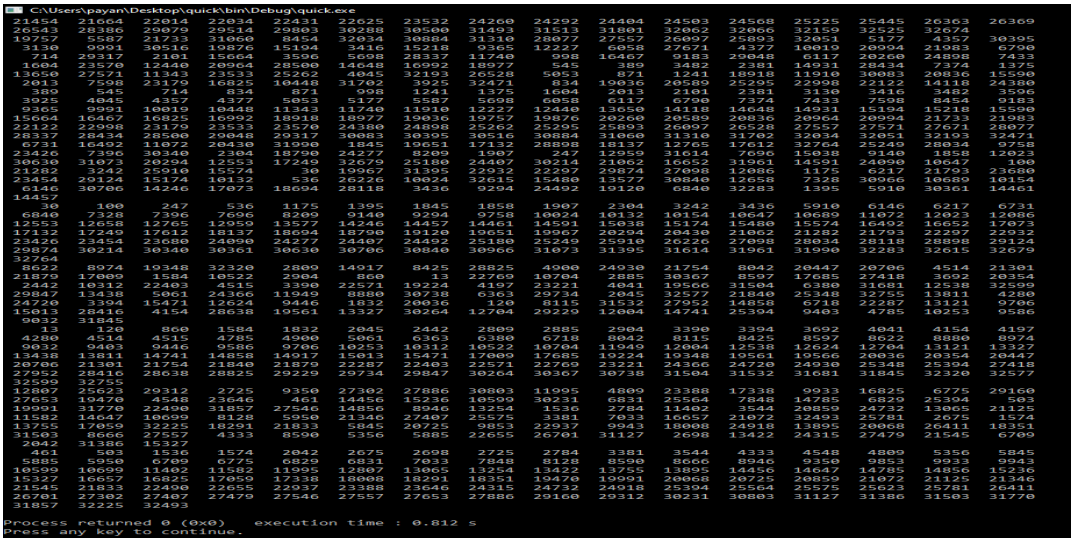


Figure 1: Ejecucion del programa, Imprime el arreglo antes de ser ordenado y despues de ser ordenado, guarda en un archivo la cantidad de operaciones y el "n" para la funcion Partition (Se ejecuta en un arreglo aleatorio, aislado del ocupado por QuickSort), y los mismos datos pero para la funcion QuickSort.

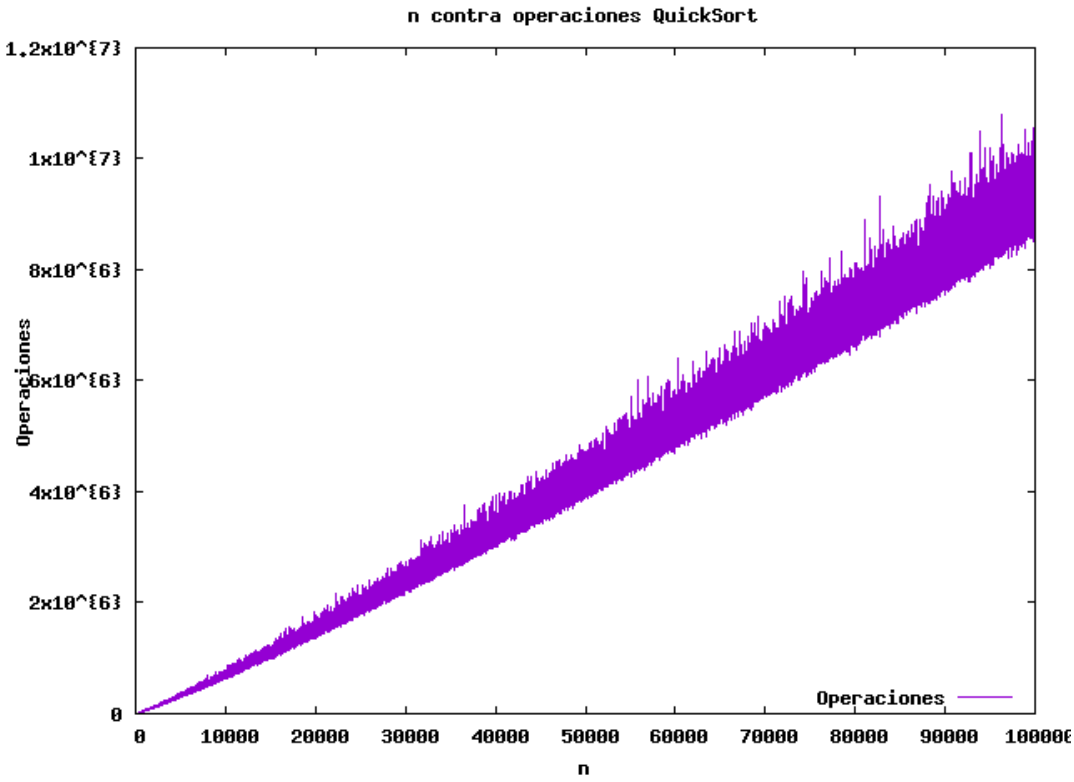


Figure 2: N contra Operaciones de la funcion QuickSort en arreglos generados de forma aleatoria

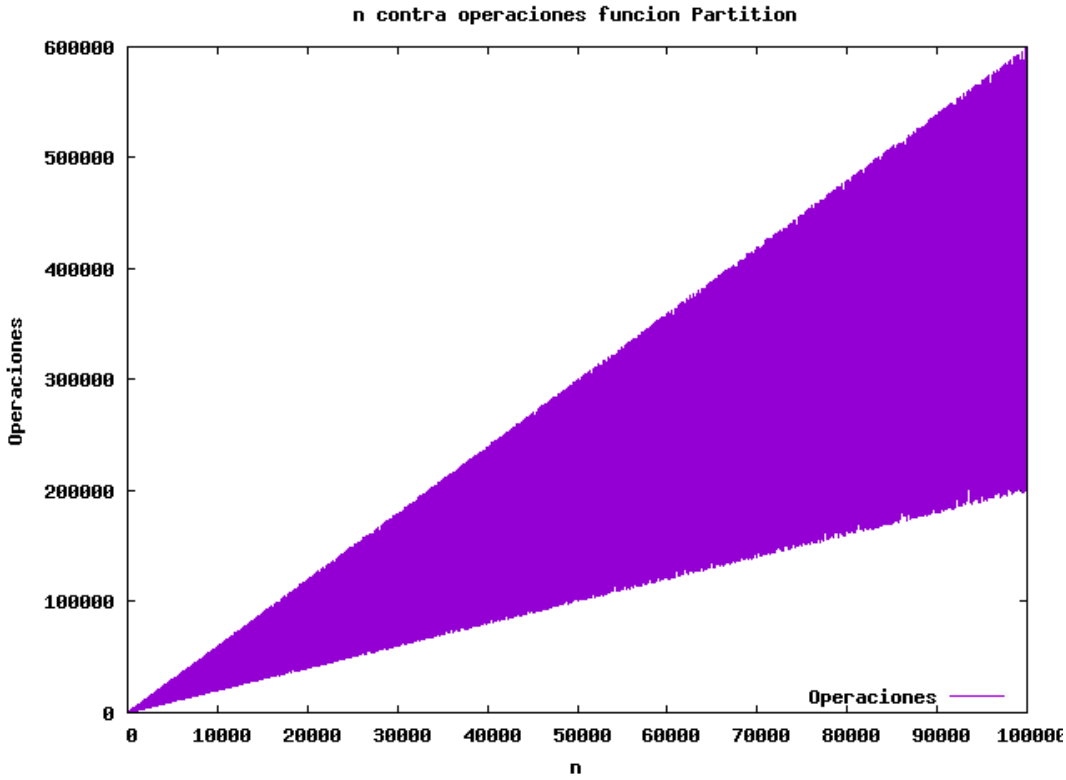


Figure 3: N contra Operaciones de la funcion Partition en arreglos generados de forma aleatoria (No es el mismo arreglo que QuickSort)

Ahora se procede a ejecutar la funcion QuickSort en un arreglo que contiene los numeros entre el 1 y n, donde n va desde 2 hasta 10000 ordenados de 4 formas distintas. Siendo estas:

1. Esta ordenado de forma ascendente. Ejemplo: 1,2,3,4
2. Esta ordenado de forma descendente. Ejemplo: 4,3,2,1
3. Esta ordenado de forma aleatoria. Ejemplo: 2,4,1,3
4. Esta ordenado de forma que el pivote corte el arreglo a la mitad, es decir la primera mitad del arreglo de forma ascendente y la siguiente de forma descendente. Ejemplo: 1,2,4,3

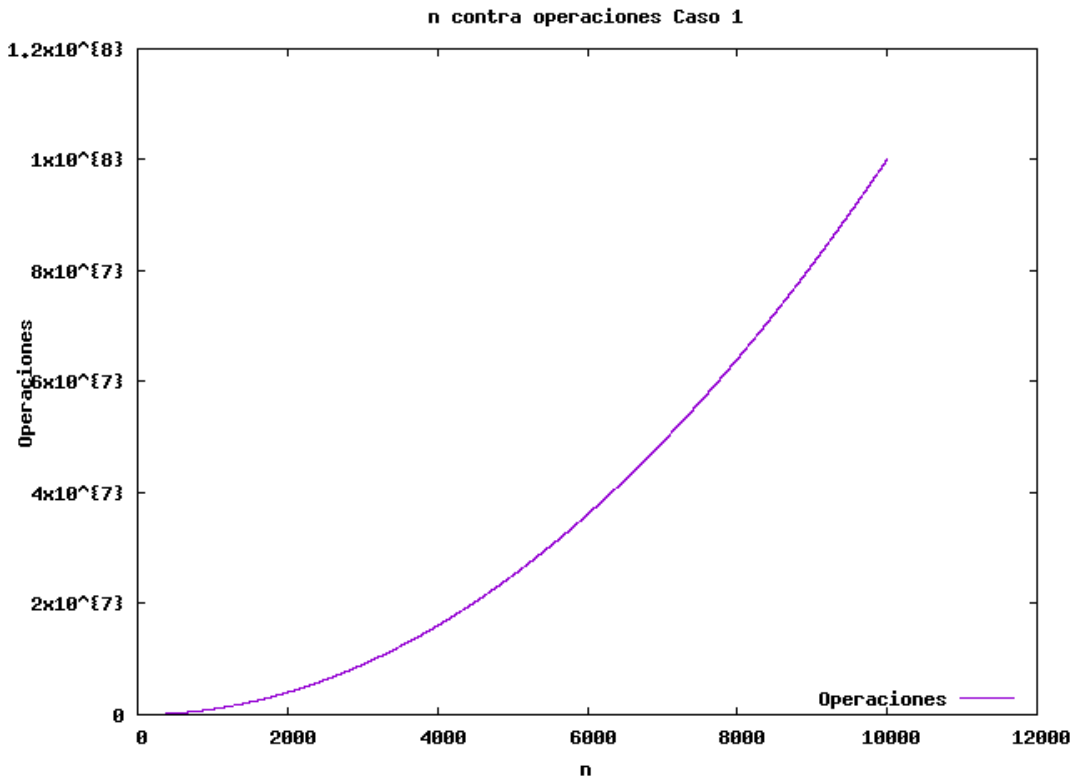


Figure 4: N contra Operaciones del caso 1 de QuickSort

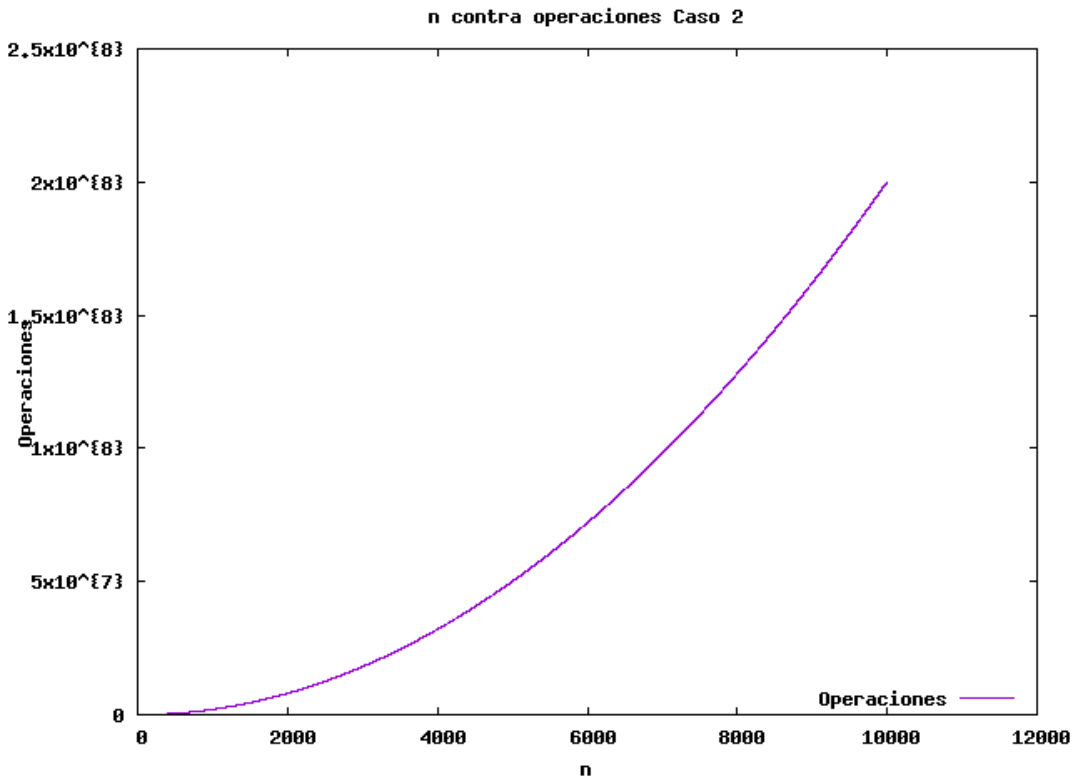


Figure 5: N contra Operaciones del caso 2 de QuickSort

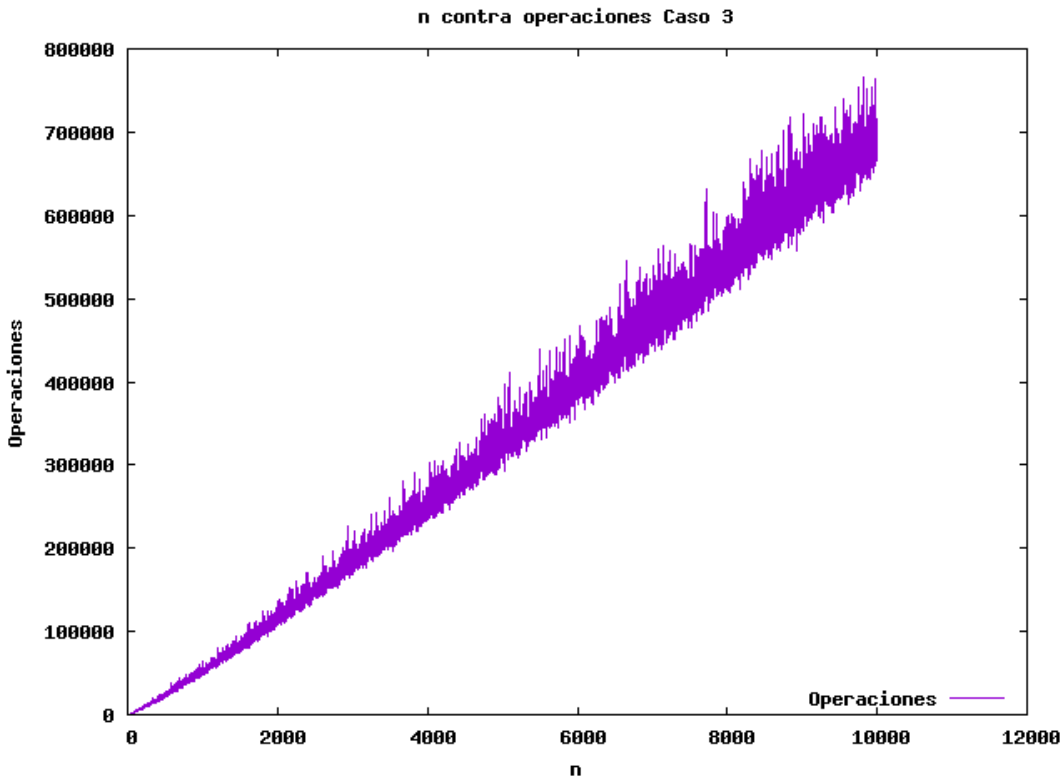


Figure 6: N contra Operaciones del caso 3 de QuickSort

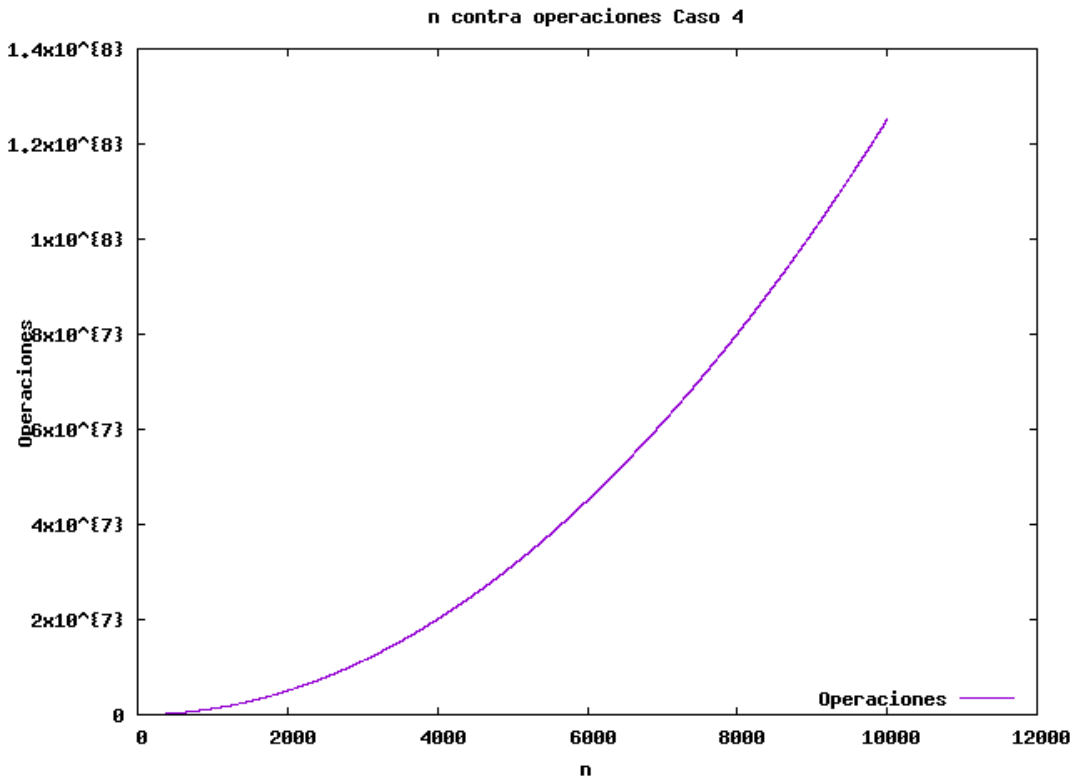


Figure 7: N contra Operaciones del caso 4 de QuickSort

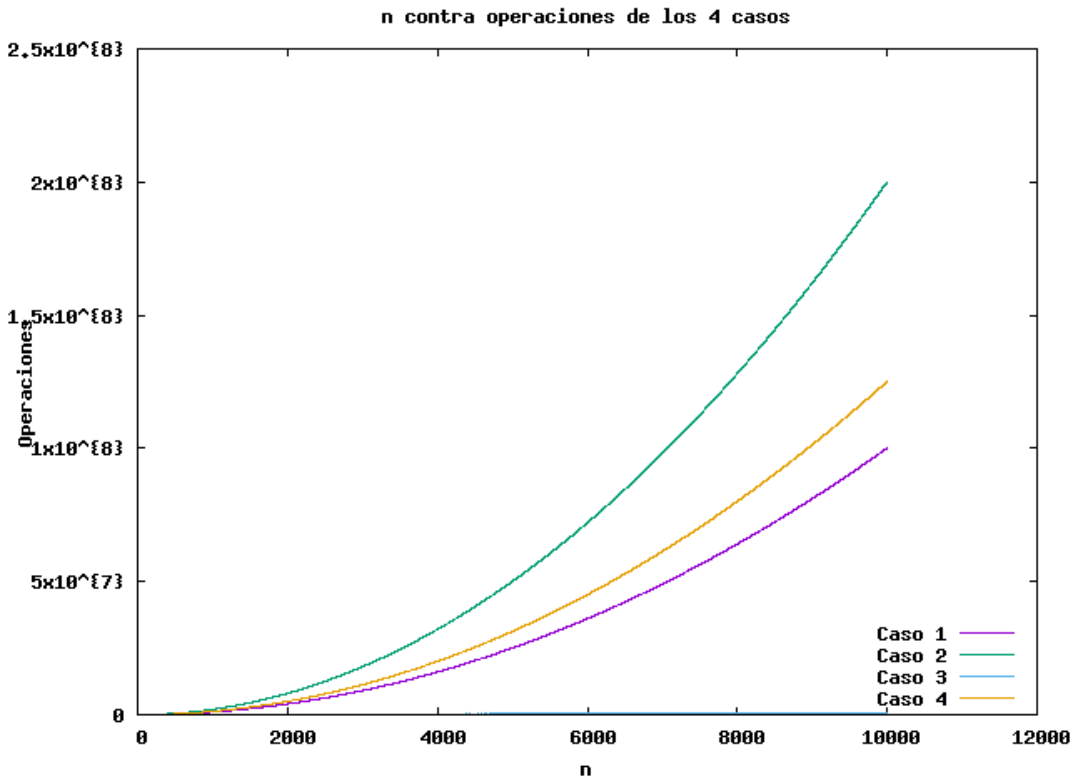


Figure 8: N contra Operaciones de todos los casos combinados de QuickSort

Ahora procedemos a calcular formalmente la complejidad de la funcion Partition Empezamos calculando la complejidad temporal de cada linea:

Codigo	Costo	Veces ejecutado
x = A[hasta]	Ø(1)	1
i = desde-1	Ø(1)	1
for(j=desde;j<hasta;j++)	Ø(n)	n+1
if(A[j]¡i)	Ø(1)	n
i++	Ø(1)	desconocido
intercambiar(A[i],A[j])	Ø(3)	desconocido
intercambiar(A[i+1],A[hasta])	Ø(3)	1
return i+1	Ø(1)	1

Esto nos indica que la complejidad de la funcion partition se ve determinada por el for que recorre desde 0 hasta n y como no es posible calcular la cantidad de veces que entrara dentro del if (ya que esto dependera del pivote), y apoyandonos por la Figure 3, podemos concluir que la funcion partition tiene complejidad Ø(n).
Ahora procedemos a calcular formalmente la complejidad de la funcion QuickSort (suponiendo que el pivote corta en medio del arreglo) Empezamos calculando la complejidad temporal de cada linea:

Codigo	Costo	Veces ejecutado
if(desde ¡ hasta)	$\emptyset(1)$	1
q = Partition(A,desde,hasta)	$\emptyset(n)$	1
QuickSort(A,desde,q-1)	$T(\frac{n}{2})$	1
QuickSort(A,q+1,hasta)	$T(\frac{n}{2})$	1

Bajo este escenario que coincide con el caso 4 de los analisis respecto a QuickSort (Figure 7) obtenemos la siguiente ecuacion de recurrencia:
$T(n) = 2T(\frac{n}{2}) + \emptyset(n)$
Que al ser desarrollada llegaremos a la conclusion de que: $T(n)$ tiene complejidad $\Theta(nlog_2(n))$.
Ahora bien, para los otros 3 escenarios la cosa cambia bastante, si miramos la figure 4 podemos ver que el comportamiento de la funcion se torna cuadratico, muy similar al escenario 2 (Figure 5), ya que generamos casos donde el arreglo es dividido en 0 y n elementos, lo cual vuelve la complejidad del ordenamiento cuadratica.
Pero tomando el caso de números aleatorios (Figure 6 y Figure 2) podemos ver que en la mayoria de los casos, el algoritmo tendra complejidad $\emptyset(nlog_2(n))$, aunque dependera de la implementación, en este caso se tomo el pivote como el ultimo valor del arreglo, lo cual es puede ser poco eficiente, por ello se recomienda tomar valores como la media o mediana.

3.2 Implementar el algoritmo del Maximo SubArreglo

Para este algoritmo se decidio generar un arreglo aleatorio de n elementos (n va de 1 hasta 10000), y medir la complejidad del algoritmo de fuerza bruta y del algoritmo de divide y venceras.

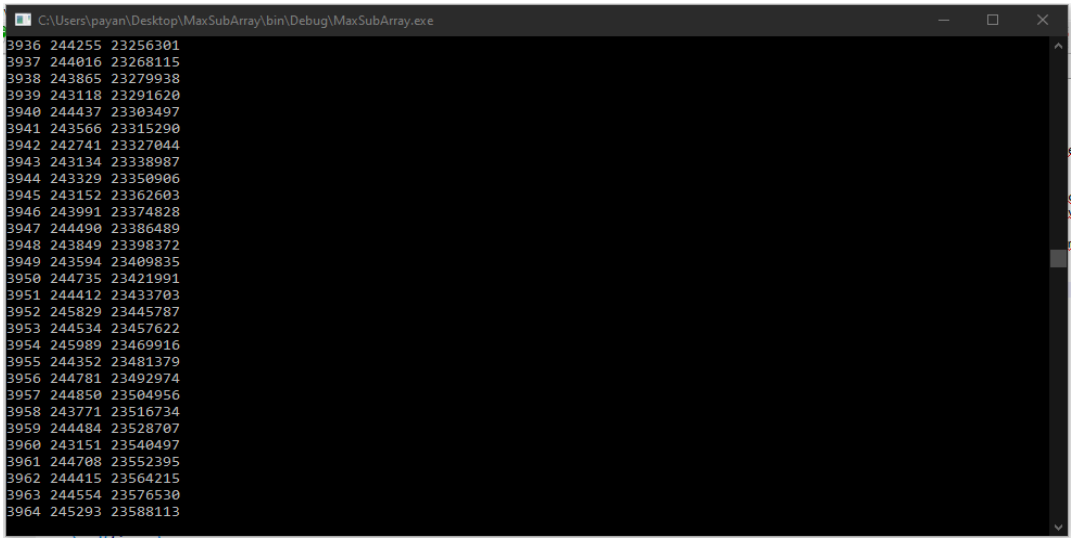


Figure 9: Ejecucion del programa en la seccion de prueba de números

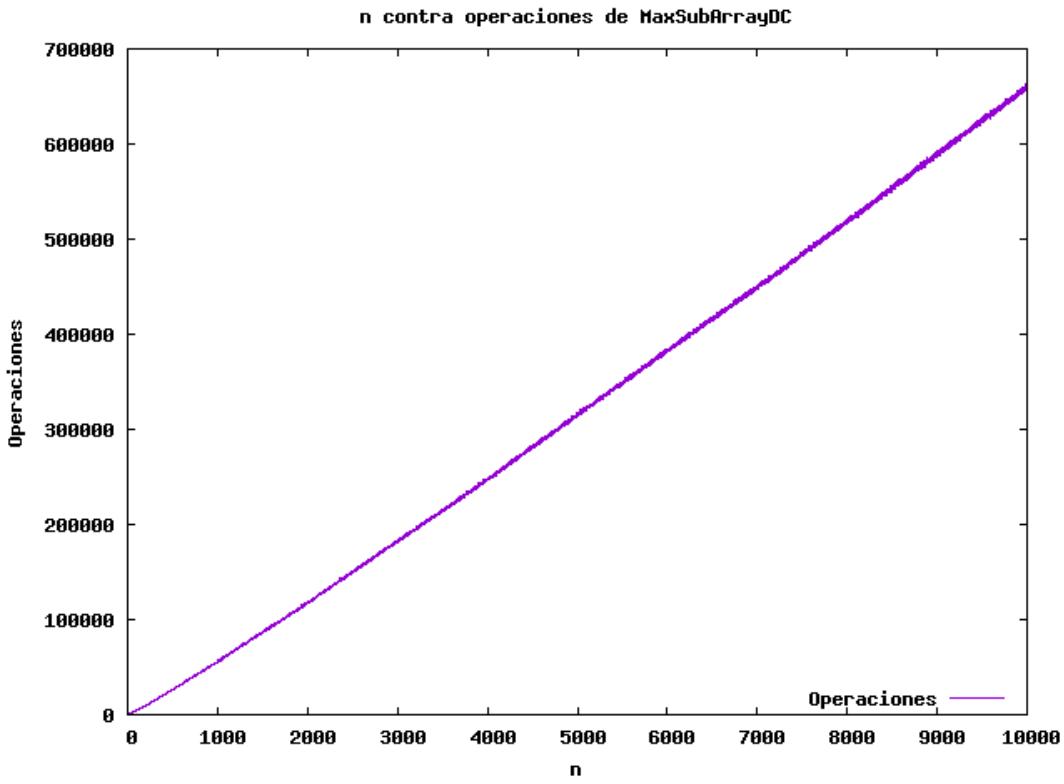


Figure 10: N contra Operaciones de MaxSubArrayDC en arreglos aleatorios de tamaño n

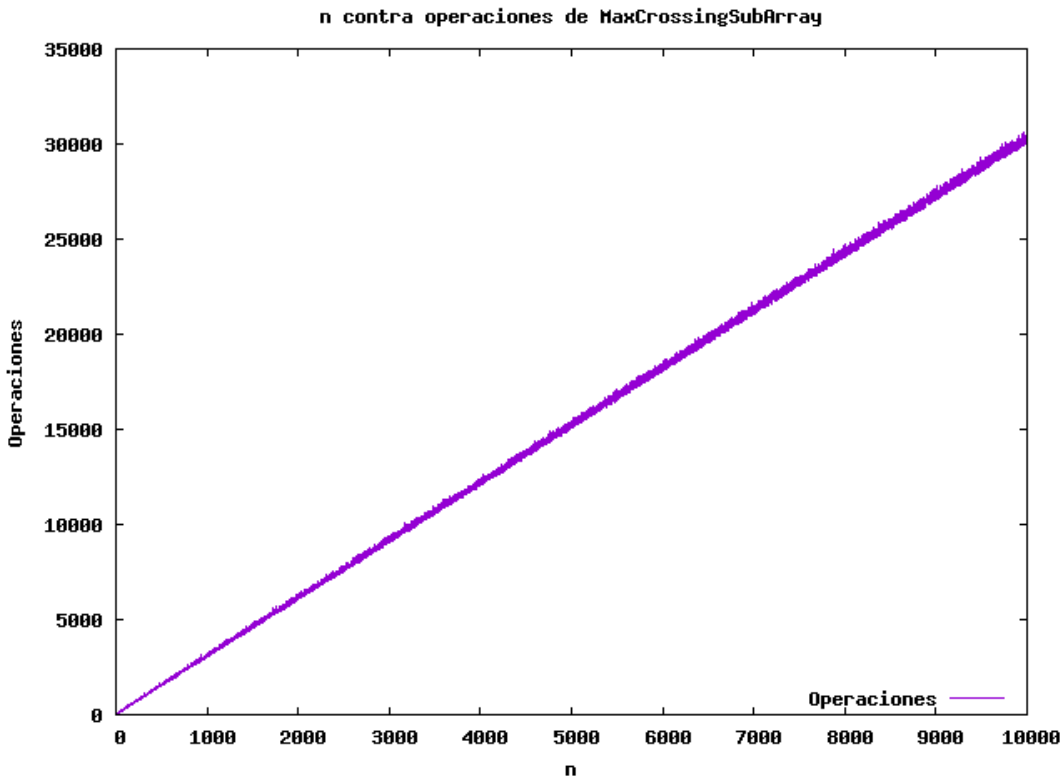


Figure 11: N contra Operaciones de MaxCrossingSubArray en arreglos aleatorios de tamaño n

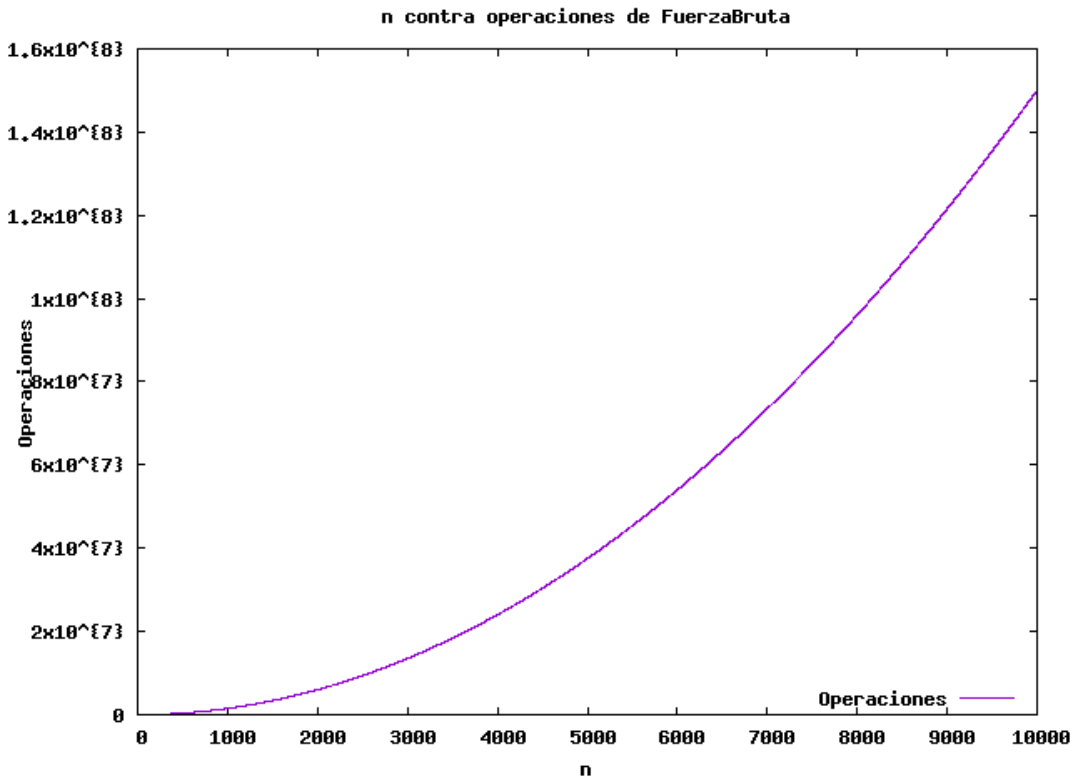


Figure 12: N contra Operaciones de FuerzaBruta en arreglos aleatorios de tamaño n

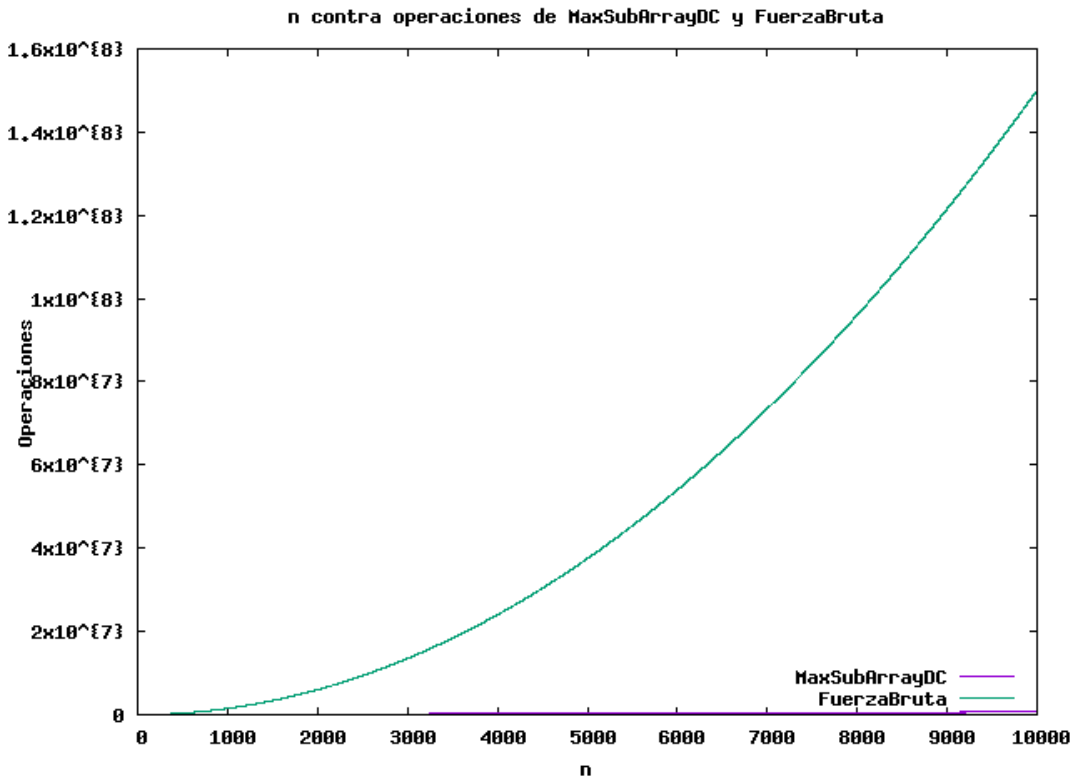


Figure 13: N contra Operaciones de MaxSubArrayDC y FuerzaBruta combinados en arreglos aleatorios de tamaño n

Ahora procederemos a calcular formalmente la complejidad de la funcion MaxCrossingSubArray:

Codigo	Costo	Veces ejecutado
suma, suma_izq, suma_der	Ø(1)	1
max_izq = max_der = INT_MIN;	Ø(1)	1
for(i=mitad;i≥bajo;i–)	Ø(1)	$\frac{n}{2}+1$
suma+=arreglo[i]	Ø(1)	$\frac{n}{2}$
if(suma>suma_izq)	Ø(1)	$\frac{n}{2}$
max_izq=i	Ø(1)	indefinido
suma_izq=suma	Ø(1)	indefinido
for(i=mitad+1;i<alto;i++)	Ø(1)	$\frac{n}{2}$+1
suma+=arreglo[i]	Ø(1)	$\frac{n}{2}$
if(suma>suma_der)	Ø(1)	$\frac{n}{2}$
max_der=i	Ø(1)	indefinido
suma_der=suma	Ø(1)	indefinido
return(suma_izq+suma_der,max_izq,max_der)	Ø(1)	1

Al analizar la complejidad linea a linea y apoyado por la grafica "Figure 11" , podemos decir que la complejidad de la funcion es $O(n)$. Una vez que tenemos este dato, procedemos a analizar la complejidad de la funcion MaxSubArrayDC:

Codigo	Costo	Veces ejecutado
suma, suma_izq, suma_der	$\emptyset(1)$	1
max_izq = max_der = INT_MIN;	$\emptyset(1)$	1
for(i=mitad;i≥bajo;i–)	$\emptyset(1)$	$\frac{n}{2}+1$
suma+=arreglo[i]	$\emptyset(1)$	$\frac{n}{2}$
if(suma>suma_izq)	$\emptyset(1)$	$\frac{n}{2}$
max_izq=i	$\emptyset(1)$	indefinido
suma_izq=suma	$\emptyset(1)$	indefinido
for(i=mitad+1;i<alto;i++)	$\emptyset(1)$	$\frac{n}{2}+1$
suma+=arreglo[i]	$\emptyset(1)$	$\frac{n}{2}$
if(suma>suma_der)	$\emptyset(1)$	$\frac{n}{2}$
max_der=i	$\emptyset(1)$	indefinido
suma_der=suma	$\emptyset(1)$	indefinido
return(suma_izq+suma_der,max_izq,max_der)	$\emptyset(1)$	1

Terminado el analisis, obtendremos que la complejidad esta determinada por la siguiente ecuacion de recurrencia: $T(n) = 2T(\frac{n}{2})+\emptyset(n)$, una vez resuelta, determinaremos que la complejidad final de la funcion MaxSubArrayDC es $\emptyset(nlog_2(n))$ esto lo podemos respaldar con las Figure 10 y Figure 13 , en las cuales se aprecia que la complejidad de la funcion esta acotada por $\emptyset(nlog_2(n))$. Finalmente como contraste se analizara la funcion FuerzaBruta:

Codigo	Costo	Veces ejecutado
sumaLocal, lIndex, rIndex = 0	$\emptyset(1)$	1
suma = INT_MAX	$\emptyset(1)$	1
for(i=0;i<n;i++)	$\emptyset(1)$	$n+1$
sumaLocal=0	$\emptyset(1)$	n
for(j=i;j<n;j++)	$\emptyset(1)$	n
if(sumaLocal>suma)	$\emptyset(1)$	$\frac{(n)(n-1)}{2}+1$
suma = sumaLocal	$\emptyset(1)$	indefinido
lIndex = i	$\emptyset(1)$	indefinido
rIndex = j	$\emptyset(1)$	indefinido
return(suma,lIndex,rIndex)	$\emptyset(1)$	1

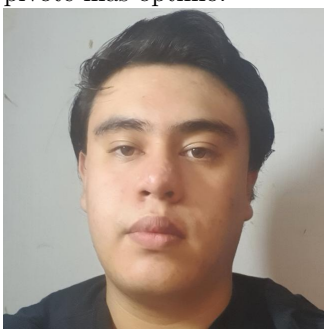
Al terminar el analisis podemos ver que la complejidad de la funcion es $\emptyset(n^2)$ lo cual coincide bastante con lo que se aprecia en las graficas (Figure 12 y Figure 13).

4 Conclusiones

4.1 Payán Téllez René

Despues de analizar los algoritmos del maximo sub arreglo y del QuickSort, puedo concluir que la tecnica de divide y venceras es extremadamente potente y sirve para resolver problemas que de otra forma tomarian mucha mas complejidad computacional, por ejemplo la funcion FuerzaBruta y el BubbleSort son prueba de ello ya que estos algoritmos que implementan soluciones mas directas, tienen complejidades enormes. Por otro lado no todo es perfecto, ya que en el caso particular del QuickSort, pude apreciar que la complejidad podia crecer hasta ser extremadamente ineficiente en casos donde el arreglo este ordenado de tal forma que el pivote no corte nada del mismo, aunque analizando a profundidad estos casos pueden ser diezmados y en ciertas implementaciones que me encontre en la web, se toman varias medidas para hacerlo, como aleatorizar el arreglo o buscar un

pivote mas optimo.



5 Anexo

5.1 Investigar el algoritmo de Karatsuba que permite obtener la multiplicación de enteros muy grandes

Resuelto, se encuentra en los conceptos basicos

5.2 Resolver los siguientes problemas

6 Bibliografia

[1]https://openwebinars.net/blog/que-es-un-algoritmo-informatico/
[2]http://www.lcc.uma.es/~av/Libro/CAP3.pdf
[3]https://es.qaz.wiki/wiki/Karatsuba_algorithm