



Escuela Superior de Cómputo.
Instituto Politécnico Nacional, México.



Práctica 3: Divide y Vencerás. Algoritmo MergeSort.

Blancas Pérez Bryan Israel
orionmunecaycanica@gmail.com

Resumen: Aplicar la técnica "Divide y Vencerás", específicamente con el algoritmo de ordenamiento MergeSort, y encontrar el orden de complejidad de éste.

Palabras Clave: Divide y Vencerás, Merge, MergeSort.

1. Introducción

En ocasiones, existen problemas complicados de resolver si se abordan como uno solo, comúnmente a estos problemas se les aplica un enfoque de "Divide y Vencerás". En esta práctica se muestra el empleo de esta técnica para ordenar arreglos de números, utilizando el algoritmo MergeSort.

2. Conceptos Básicos

Divide y Vencerás.

El enfoque "Divide y Vencerás", está basado en la resolución recursiva de un problema, separándolo en sub-problemas hasta que éstos lleguen a ser suficientemente sencillos para resolverlos directamente, para luego combinar las soluciones de esos sub-problemas para resolver el problema principal [1]. Se puede explicar el método en tres pasos.

1. **Divide** el problema en sub-problemas de la misma naturaleza.
2. **Vence** a los sub-problemas sencillos resolviéndolos.
3. **Combina** las soluciones de los sub-problemas para resolver el problema principal.

MergeSort.

MergeSort es un algoritmo de ordenamiento basado en la técnica "Divide y Vencerás". Fue desarrollado por John Von Neumann 1945. [2]
Este algoritmo divide una la lista desordenada hasta que quedan sublistas de tamaño uno, las cuales, por concepto, ya están ordenadas. Una vez hecho esto, utiliza el algoritmo Merge para comparar y ordenar las sublistas, para luego mezclarlas en una sola lista, así sucesivamente hasta ordenar todo el arreglo original.

3. Experimentación y Resultados

Ejercicio 1.

Implementar el algoritmo MergeSort.

Pseudocódigo del algoritmo Merge:

```
1 Merge(A, p, q, r)
2   n1=q-p+1
3   n2=r-q
4
5   Sean L[0, ..., n1-1] y R[0, ..., n2-1]
6
7
```

```

8   for i=0 to i < n1 do
9       L[i]=A[p+i]
10  for j=0 to j < n2 do
11      R[j]=A[q+1+j]
12
13  i=j=0
14
15  for k = p to k <= r do
16      if L[i] <= R[j]
17          A[k] = L[i]
18          i++
19      else
20          A[k] = R[j]
21          j++

```

El algoritmo Merge, recibe como entrada un arreglo $A[p, \dots, q, \dots, r]$ bajo la premisa de que los números de p a q están ordenados de manera creciente, al igual que los números de $q + 1$ a r . Como salida, el algoritmo devuelve el arreglo $A[p, \dots, q, \dots, r]$ ordenado de manera creciente.

Pseudocódigo del algoritmo MergeSort:

```

1 MergeSort(A[p, ... r], p, r)
2   if p < r then
3       q = (p+r)/2
4       MergeSort(A[p, ... , r], p, q)
5       MergeSort(A[p, ... , r], q+1, r)
6       Merge(A, p, q, r)

```

El algoritmo MergeSort, recibe como entrada un arreglo $A[p, \dots, r]$, arreglo que va dividiendo por medio de llamadas recursivas. Se realiza esa operación hasta que $p = r$, es decir, cuando se halla dividido el arreglo A en subarreglos de tamaño uno. Una vez hecho esto, se hace uso de la función Merge, la cual ordenará esos subarreglos en orden creciente. MergeSort devuelve al arreglo A ordenado de crecientemente.

Ejecución de la función Merge

```
bryan@bryan-ubuntu:~$ python3 merge.py
5,6,7,8,9,1,2,3,4,
Left: 5,6,7,8,9,
Right: 1,2,3,4,
1,2,3,4,5,6,7,8,9,
bryan@bryan-ubuntu:~$
```

Figura 1: Ejecución del algoritmo Merge.

En la figura 1, se aprecia la ejecución del algoritmo Merge. Como se puede ver, recibe un arreglo el cual tiene las mitades ordenadas crecientemente. Se muestra parte del procedimiento, imprimiendo cómo el algoritmo ha separado el arreglo original en dos subarreglos. Como salida se muestra todo el arreglo ordenado de manera creciente.

Gráfica del orden de complejidad del algoritmo Merge.

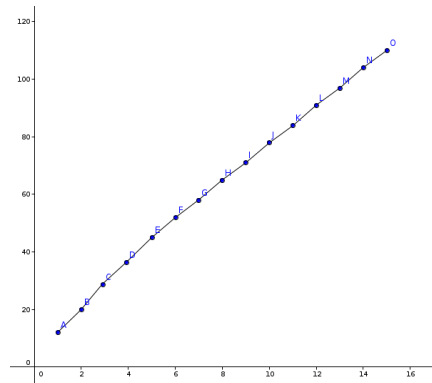


Figura 2: Gráfica del orden de complejidad del algoritmo Merge.

En la figura 2, se muestra la gráfica $T(n)$ vs n . Como es apreciable, el orden de complejidad del algoritmo Merge, para ordenar un arreglo tiene orden de complejidad lineal. Observación: la gráfica muestra "saltos" debido a las condiciones implementadas en el algoritmo para su correcto funcionamiento, las cuales modifican $T(n)$ (cont).

Demostración analítica del orden de complejidad del algoritmo Merge.

```

1 Merge(A, p, q, r)
2   n1=q-p+1
3   n2=r-q
4   Sean L[0, ..., n1-1] y R[0, ..., n2-1]
5
6   for i=0 to i < n1 do
7     L[i]=A[p+i]
8   for j=0 to j < n2 do
9     R[j]=A[q+1+j]
10
11   i=j=0
12
13   for k = p to k <= r do
14     if L[i] <= R[j]
15       A[k] = L[i]
16       i++
17     else
18       A[k] = R[j]
19       j++

```

Análisis del Algoritmo	
Línea de Código	Orden de complejidad
2	$\theta(1)$
3	$\theta(1)$
4	$\theta(1)$
6	$\theta(n1)$
8	$\theta(n2)$
11	$\theta(1)$
13	$\theta(r - p + 1) = \theta(n)$

Cuadro 1: Análisis del algoritmo.

Aclaración: como las líneas de código dentro de los *for's* son orden $\theta(1)$, por el teorema de la multiplicación $t_1(n)t_2(n) \in \theta(f_1(n)f_2(n))$, consideramos únicamente el orden de complejidad de la línea del *for*.

Entonces.

$$T(n) = \theta(1) + \theta(1) + \theta(1) + \theta(n1) + \theta(n2) + \theta(1) + \theta(n).$$

Pero

$$\theta(n1) + \theta(n2) = \theta(n). \text{ Puesto que } n \text{ fue particionado en } n1 \text{ y } n2.$$

Entonces.

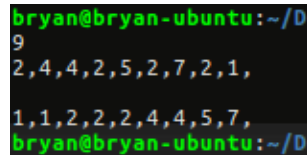
$$T(n) = \theta(1) + \theta(1) + \theta(1) + \theta(n) + \theta(1) + \theta(n).$$

Y por el teorema de la suma $t_1(n) + t_2(n) \in \theta(\max(f_1(n), f_2(n)))$. Entonces.

$$T(n) = \theta(n).$$

Por lo tanto , $T(n) \in \theta(n)$

Ejecución de la función MergeSort

A terminal window with a black background and green text. The prompt is 'bryan@bryan-ubuntu:~/De'. The first line shows the number '9'. The second line shows the array '2,4,4,2,5,2,7,2,1,'. The third line shows the sorted array '1,1,2,2,2,4,4,5,7,'. The prompt 'bryan@bryan-ubuntu:~/De' is visible at the bottom.

```
bryan@bryan-ubuntu:~/De
9
2,4,4,2,5,2,7,2,1,
1,1,2,2,2,4,4,5,7,
bryan@bryan-ubuntu:~/De
```

Figura 3: Ejecución del algoritmo MergeSort.

En la figura 3, se aprecia la ejecución del algoritmo MergeSort. Como se puede ver, recibe como entrada un número n , el cual representa el largo del arreglo a ordenar. Una vez introducido el número, se crea el arreglo tamaño n y se llena con números aleatorios. Como salida, MergeSort devuelve el arreglo ordenado de manera creciente.

Gráfica del orden de complejidad del algoritmo MergeSort.

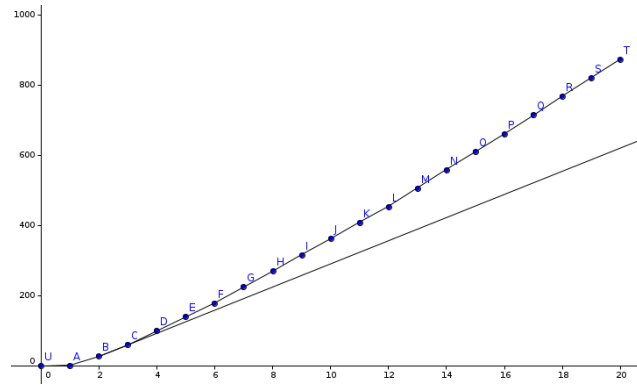


Figura 4: Gráfica del orden de complejidad del algoritmo MergeSort.

En la figura 4, se muestra la gráfica $T(n)$ vs n . En la gráfica, la curva se aprecia muy poco si se deja sola, así que de manera auxiliar para probar que existe esa curvatura, trace una recta que pasa por los puntos B y C. Como se puede ver, el orden de complejidad del algoritmo MergeSort es $n \log n$.

Demostración analítica del orden de complejidad del algoritmo MergeSort.

```

1 MergeSort(A[p, ... r], p, r)
2   if p < r then
3     q = (p+r)/2
4     MergeSort(A[p, ... , r], p, q)
5     MergeSort(A[p, ... , r], q+1, r)
6     Merge(A, p, q, r)

```

Análisis del Algoritmo	
Línea de Código	Orden de complejidad
3	$\theta(1)$
4	$T(n/2)$
5	$T(n/2)$
6	Cn (previamente demostrado)

Cuadro 2: Análisis del algoritmo.

Entonces.

$$T(n) = \begin{cases} \theta(1) & \text{si } n = 1 \\ 2T(n/2) + Cn & \text{si } n > 1 \end{cases}$$

pero $\theta(1) = C$, entonces

$$T(n) = \begin{cases} C & \text{si } n = 1 \\ 2T(n/2) + Cn & \text{si } n > 1 \end{cases}$$

Así tenemos:

Sea $n|k = \log_2(n)$,

$$T(2^k) = \begin{cases} C & \text{si } 2^k = 1 \\ 2T(2^{k-1}) + C2^k & \text{si } 2^k > 1 \end{cases}$$

Usando el método de sustitución hacia atrás.

$$T(2^k) = 2^2T(2^{k-2}) + 2C2^k$$

$$T(2^k) = 2^3T(2^{k-3}) + 3C2^k$$

$$T(2^k) = 2^i T(2^{k-i}) + iC2^k$$

cuando $i=k$

$$T(2^k) = 2^k T(2^{k-k}) + kC2^k$$

$$T(2^k) = 2^k T(1) + kC2^k$$

$$T(2^k) = C2^k + kC2^k$$

$$T(2^k) = (k+1)C2^k$$

pero $k = \log_2(n)$, entonces $T(2^k) = (\log_2(n) + 1)Cn$

$$T(2^k) = Cn\log_2(n) + Cn$$

Por lo tanto, $T(n) \in \theta(n\log_2(n))$

4. Conclusiones

Esta práctica me pareció muy buena para entender el concepto de "Divide y Vencerás", y aparte de comprenderlo, lo pudimos aplicar al hacer el algoritmo MergeSort. Tuve un par de complicaciones al hacer el algoritmo Merge, ya que al principio no tenía bien claro que, para que funcione el algoritmo, es necesario que el algoritmo este dividido en dos arreglos **ya** ordenados previamente. Como yo no estaba tomando en cuenta eso, el algoritmo no funcionaba.

La otra complicación que tuve, fue al darme cuenta de que, al momento de comparar los subarreglos, existen casos en los que los índices se acaban, pero lo solucioné poniendo un par de condiciones para ordenar el resto del arreglo. Los resultados fueron los esperados, y a mi parecer esta fue una práctica muy productiva, tanto teórica como prácticamente.

5. Anexo

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

A.

```
1 Funcion1(n par)
2   i=0
3   mientras i < n hacer
4     para j = 1 hasta j = 10 hacer
5       accion(i)
6       j++
7     i+=2
```

Suponga $\text{Accion}(i) \in \theta(1)$.

Análisis del Algoritmo	
Línea de Código	Orden de complejidad
2	$\theta(1)$
3	$\theta(n/2)$
4	$\theta(1)$
5	$\theta(1)$
6	$\theta(1)$
7	$\theta(1)$

Cuadro 3: Análisis del algoritmo.

Entonces el código dentro del *while* tiene orden de complejidad constante, por lo tanto el bloque *while*, aplicando el teorema de la multiplicación, queda con orden $\theta(n)$.

El bloque *while* depende de n , sin hacer distinción entre el peor y mejor caso.

Por lo tanto $\text{Funcion1} \in \theta(n)$.

B.

```
1 Funcion2(A[0,...,n-1, x entero])
2   for i = 0 to i < n do
3     if A[i] < x then
4       A[i]=min(A[0,...,n-1])
5     else if A[i] > x then
6       A[i]=max(A[0,...,n-1])
7     else
8       exit
```

Análisis del Algoritmo	
Línea de Código	Orden de complejidad
2	$\theta(n)$
3	$\theta(1)$
4	$\theta(n)$
5	$\theta(1)$
6	$\theta(n)$
7	$\theta(1)$
8	$\theta(1)$

Cuadro 4: Análisis del algoritmo.

El min y max de un arreglo tiene orden de complejidad n debido a que es necesario recorrer todo el arreglo para encontrar el número que cumpla a condición.

Entonces.

Si $A[i] = x$, $i = 0, 1, 2, \dots, n - 1$.

Entonces $Funcion2 \in \Omega(n)$, debido a que las líneas de código 4 y 6 jamás se ejecutan por que la condición en la que están se los impide.

Y si $A[i] \neq x$, $i = 0, 1, 2, \dots, n - 1$.

Entonces $Funcion2 \in O(n^2)$, debido a que las líneas de código 4 y 6 se pueden ejecutar, y por el teorema de la multiplicación, los se multiplican.

6. Bibliografía

- [1] https://es.wikipedia.org/wiki/Algoritmo_divide_y_vencerás
- [2] https://en.wikipedia.org/wiki/Merge_sort