Análisis de Algoritmos, Sem: 2018-2, 3CV1, Práctica 3, 11 de Septiembre de 2018

Práctica 3: Divide y Vencerás: Algoritmo MergeSort

Reyes Valenzuela Alejandro, Gutiérrez Povedano Pablo.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*areyesv*11@*gmail.com*, *gpovedanop*@*gmail.com*

Resumen: En este reporte se utilizará el Divide y Vencerás para determinar la complejidad del algoritmo MergeSort, tanto de la parte de la separación de arreglos como el ordenamiento como tal.

Palabras Clave: Divide, Vence, Combina, Bloques.

1 Introducción

Al momento de analizar algoritmos, es necesario hacerlo de tal manera que podamos hacerlo de manera rápida, independientemente del tamao de la función que tendremos. Para ello los métodos observados anteriormente podrían ser algo engorrosos, ya que analizamos línea por línea.

Para ello podemos utilizar el Divide y Vencerás (cuyo funcionamiento se explicará en la siguiente sección), el cual nos ahorra bastantes pasos, siendo escencial para algoritmos grandes. Para ello tomaremos de ejemplo el algoritmo MergeSort, un algoritmo que ocupa recursividad y que depende de otra subfunción para funcionar.

2 Conceptos Básicos

El paradigma de Divide y Vencerás separa un problema en subproblemas que se parecen al problema original, de manera recursiva resuelve los subproblemas y, por último, combina las soluciones de los subproblemas para resolver el problema original.

Como divide y vencers resuelve subproblemas de manera recursiva, cada subproblema debe ser más pequeo que el problema original, y debe haber un caso base para los subproblemas.

Para ello recurrimos a tres pasos fundamentales:

- **Divide:** Dividir el problema en un número de subproblemas que son instancias más pequeas del mismo problema.
- **Vence:** Resolver los subproblemas de manera recursiva. Si son los suficientemente pequeños, resolver los subproblemas como casos base.
- **Combina:** Juntar las soluciones de los subproblemas en la solución para el problema original.

Utilizaremos el paradigma para calcular el órden de complejidad del alogritmo MergeSort. Como sabemos, MergeSort se auxilia de otra función denominada Merge que se encarga de separar arreglos.

Algoritmo Merge

$Merge(A[p, ..., q, ..., r], p, q, r)$; donde p $\leq$ q $<$ r
$n_1 = q$ - $p + 1$
$n_2 = r$ - q
Sean $L[n_1]$ y $R[n_2]$ dos arreglos
$for(i = 0; i < n_1; i++)$
$L[i] = A[p + i]$
$for(j = 0; j < n_2; j++)$
$R[j] = A[q + i + j]$
$i = 0$; $j = 0$
$for(k = 0; k < r - p + 1; k++)$
$if(L[i] \leq R[j])$
$A[k] = L[i]$
$i++$
$else$
$A[k] = R[j]$
$j++$

Algoritmo MergeSort

$MergeSort(A[p, ..., r], p, r)$
In: *Arreglo* $A[p, ..., r]$
Out: *Arreglo ordenado*
$if(p < r)$
    $q = \frac{p+r}{2}$
    $MergeSort(A, p, q)$
    $MergeSort(A, q + 1, p)$
    $Merge(A, p, q, r)$

3 Experimentación y Resultados

Calculando el órden de complejidad de Merge: (n=r-p+1)

$n_1 = q$ - $p + 1$	C_1	1	$\Theta(1)$
$n_2 = r$ - q	C_2	1	$\Theta(1)$
Sean $L[n_1]$ y $R[n_2]$ dos arreglos	C_3	1	$\Theta(1)$
$for(i = 0; i < n_1; i++)$	C_4	$n_1 + 1$	$\Theta(n_1 + n_2)$
$L[i] = A[p + i]$	C_5	n_1	$\Theta(n_1 + n_2)$
$for(j = 0; j < n_2; j++)$	C_6	$n_2 + 1$	$\Theta(n_1 + n_2)$
$R[j] = A[q + i + j]$	C_7	n_2	$\Theta(n_1 + n_2)$
$i = 0;\ j = 0$	C_8	1	$\Theta(1)$
$for(k = 0; k < r - p + 1; k++)$	C_9	$r - p + 2$	$\Theta(n)$
$if(L[i] \leq R[j])$	C_{10}	$r - p + 1$	$\Theta(n)$
$A[k] = L[i]$	C_{11}	n_1	$\Theta(n)$
$i++$			
$else$			
$A[k] = R[j]$	C_{12}	n_2	$\Theta(n)$
$j++$			

Usaremos el paradigma de divide y vencerás para resolver esto. Para ello separaremos el código y lo analizaremos por bloques.

Las primeras tres líneas y la línea nueve se ejecutan un número constante de veces, por eso su complejidad es $\Theta(1)$.

En el caso de las líneas 4, 5, 6 y 7, el número de veces depende de los valores de n_1 y n_2, además los for que hacen uso de estas variables no están anidados y tienen incrementos unitarios, por eso podemos decir que $\Theta(n_1 + n_2) = \Theta(n)$.

Lo mismo ocurre de la línea 9 a la 12, no existen más iteraciones dentro del for, por lo que en ese bloque la complejidad también es $\Theta(n)$.

Juntaremos las soluciones de los bloques para poder determinar la compeljidad del código completo, en este caso, los bloques de mayor complejidad son $\Theta(n)$, mientras que en los de menor complejidad es $\Theta(1)$, al juntar ambas soluciones tenemos que Merge ϵ $\Theta(n)$. Ahora calcularemos la complejidad de Mergesort, donde analizaremos una función con recursividad con el paradigma utilizado anteriormente:

$$
\begin{array}{ll}
MergeSort(A[q,...,r],p,r) & \\
if(p<r) & \\
\quad q=\frac{p+r}{2} & \Theta(1) \\
\quad MergeSort(A,p,q) & \mathrm{T}(\frac{n}{2}) \\
\quad MergeSort(A,q+1,p) & \mathrm{T}(\frac{n}{2}) \\
\quad Merge(A,p,q,r) & \Theta(n)
\end{array}
$$

En el bloque donde asignamos el valor de q tras la comparación tiene se ejecuta una cantidad constante de veces, por lo que su complejidad es $\Theta(1)$. En el caso donde llamamos a la función Merge, demostramos anteriormente que su complejidad es $\Theta(n)$. Ahora sólo nos enfocaremos en la recursividad de la función.

$$
T(n)=\begin{cases} \Theta(1) & si \quad n=1 \\ \\ 2T(\frac{n}{2})+\Theta(n)+\Theta(1) & si \quad n>1 \end{cases}
$$

Como $\Theta(1)$ es constante y $\Theta(n)$ se ejecutará un número constante de veces tenemos que:

$$
T(n)=\begin{cases} c & si \quad n=1 \\ \\ 2T(\frac{n}{2})+c(n) & si \quad n>1 \end{cases}
$$

Ahora resolveremos la recurrencia, como tenemos un cociente, debemos asegurarnos que nuestro resultado sea entero, para ello se propone el cambio de variable $n=2^k$, donde $k=\log_2(n)$, por lo que ahora resolveremos:

$$
T(n)=2T(\tfrac{n}{2})+C(n)=2T(2^{k-1})+C(2^k)
$$

Haciendo iteraciones:

$$
T(2^k)=2(2T(2^{k-2})+C(2^{k-1}))+C(2^k)
$$

Llegando al término i:

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

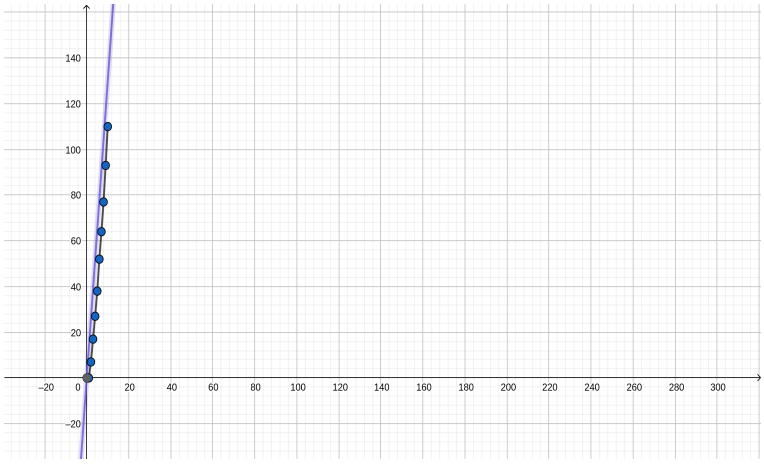
Llegando a la condición de frontera:

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

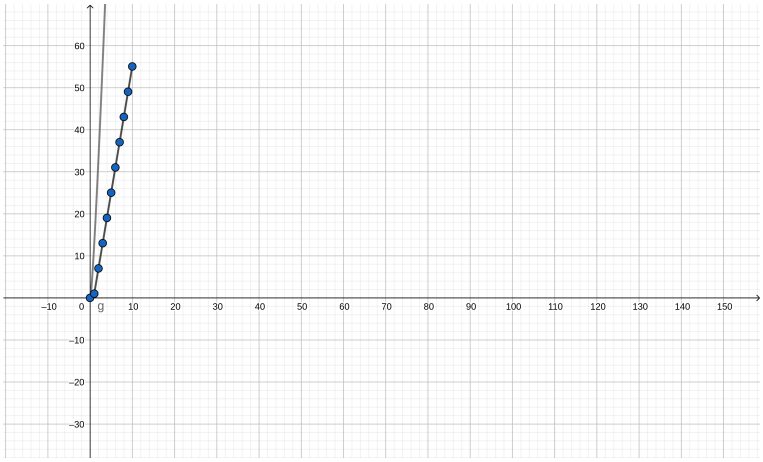
Finalmente:

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

Por lo que MergeSort ϵ $\Theta(n \log_{10}(n))$. A continuación reafirmaremos esto de manera gráfica: Algoritmo Merge (comparando los tiempos de longitud 1 a 10 con n=13n)



Algoritmo MergeSort (comparando los tiempos de longitud 1 a 10 con n=14nlog(n)



Podemos ver que las funciones propuestas acotan a los puntos de las funciones Merge y Mergesort, por lo que comprobamos que MergeSort $\epsilon\ \Theta(n \log_{10}(n))$ y que Merge $\epsilon\ \Theta(n)$.

Prueba de escritorio

```
[alejandro@alejandro-pc ~]$ ./mergesortchido
Arreglo inicial
9 5 3 6 2 4 2 5 3 1 4 3
Arrelgo ordenado
1 2 2 3 3 3 4 4 5 5 6 9
Contador Merge: 110
Contador MergeSort: 56
[alejandro@alejandro-pc ~]$
```

4 Conclusiones

Conclusión General: Dividir y vencerás nos sirve para esta clase de algortimos, ya que aqu la recursividad se usa más de una vez, además de que una función depende de la otra, por lo que para ahorrar tiempo en el análisis, podemos revisar los bloques para finalmente juntar la solución.

Conclusión Gutiérrez Povedano: Utilizando el paradigma divide y vencers se puede resolver un problema y generar un algoritmo cuya complejidad sea menor como en el caso del algoritmo merge sort el cual tiene un orden de complejidad Θ(n*log(n)) para realizar el ordenamiento de forma ascendente de un arreglo de enteros el cual es menor al orden de complejidad de otros algoritmos que resuelven el mismo problema por ejemplo el algoritmo burbuja cuya complejidad en el peor caso es $O(n^2)$.

Conclusión Reyes Valenzuela: Para este problema tuvimos que estar bastante atentos a la colocación de los índices, ya que estos nos indicaban por dónde partir y con base a esto, cómo se comportaría el algoritmo, por otra parte, pudimos ver que a pesar de devolver valores semejantes al principio, los contadores crecían de manera distinta, esto se debe a que cuando Merge es llamado, los contadores de MergeSort ya habían cambiando para ése entonces.

5 Anexo

Calcular el orden de complejidad de los siguientes algoritmos en el mejor (Ω) y en el peor de los casos (O):

$Funcion1(n\ par)$
    $i = 0$
    $mientras\ i < n\ hacer$
        $para\ j = 1\ hasta\ j = 10\ hacer$
        $Accion(i)$
        $j++;$
    $i+ = 2;$

Suponga que Accion $\epsilon\ \theta(1)$.

$Funcion2(A[0, ..., n-1]\ ,\ x\ entero)$
    $for\ i = 0\ to\ i < n\ do$
        $if\ (A[i] < x)$
            $A[i] = min(A[0, ..., n-1])$
        $else\ if\ (A[i] > x)$

$A[i] = max(A[0, ..., n-1])$
else
exit

Función 1: Para ambos casos analizaremos las funciones por bloques:

$Funcion1(n\ par)$	
$i = 0$	$\Theta(1)$
$Funcion1(n\ par)$	
$i = 0$	$\Theta(1)$
$mientras\ i < n\ hacer$	$\Theta(n)$
$para\ j = 1\ hasta\ j = 10\ hacer$	$\Theta(1)$
$Accion(i)$	–
$j++;$	$\Theta(1)$
$i+ = 2;$	–

En este caso, nuestro peor y mejor caso son el mismo, ua que el for interno siempre se ejecutará diez veces, y esas diez veces se harán con base al número n, por lo que podemos decir que el algoritmo tiene complejidad lineal.

Función 2: En este caso sí existe el mejor y el peor caso, siendo el mejor cuando el máximo o el mínimo son el primer número del arreglo, por lo que tenemos que nuestro sólo se ejecutará una vez (como n=0):

$Funcion2(A[0, ..., n-1]\ ,\ x\ entero)$	
$for\ i = 0\ to\ i < n\ do$	$\Omega(n)$
$if\ (A[i] < x)$	$\Omega(1)$
$A[i] = min(A[0, ..., n-1])$	–
$else\ if\ (A[i] > x)$	$\Omega(1)$
$A[i] = max(A[0, ..., n-1])$	–
$else$	$\Theta(1)$
exit	–

En el mejor caso es lineal. Ahora revisaremos el peor caso:

$Funcion2(A[0, ..., n-1]\ ,\ x\ entero)$	
$for\ i = 0\ to\ i < n\ do$	$O(n)$
$if\ (A[i] < x)$	$O(1)$
$A[i] = min(A[0, ..., n-1])$	$O(n)$
$else\ if\ (A[i] > x)$	$O(1)$
$A[i] = max(A[0, ..., n-1])$	$O(n)$
$else$	$O(1)$
exit	$O(1)$

En peor caso, el for se ejecutará n veces, y entrará en las comparaciones otras

n veces, por lo que al aplicar las propiedades vistas en clase obtenemos que la complejidad de este algoritmo en su peor caso es cuadrático.

6 Bibliografía

Khanacademy.org (2018). [online] Available at: https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms. [Accessed 11 Sep. 2018].