Divide y vencerás

Algoritmo para contar inversiones (Counting inversions)

Norberto Garcia Gaspar
Ángel Alberto Hamilton Lopez
Óscar David Martín Cabrera

1. Introducción

El problema del conteo de inversiones, normalmente efectuado sobre arrays, nos indica lo desordenado que esta dicho array, o lo que es lo mismo el número de movimientos que se tiene que llevar acabo para ordenar de menor a mayor (o de mayor a menor) los elementos de un array de tamaño n.

Siguiendo con el método de programación Divide y Vencerás, vamos dividiendo este array en arrays más pequeños para ordenarlos, una vez ordenados, procedemos a combinarlos y ver si hay que ordenarlo o no. El número total de inversiones por tanto serán desde los arrays más pequeños, sumados con los de las distintas combinaciones hasta obtener el array ordenado de tamaño n.

En este estudio vamos a ver una variante del Merge Sort en el cual aparte de ordenar el array resolveremos el problema mostrado anteriormente, también veremos la forma de hacerlo mediante fuerza bruta para poder apreciar las diferencias y mejoría del rendimiento entre las dos técnicas.

2. Pseudocódigos

Vamos a ver los pseudocódigos de los dos métodos para contar las inversiones. Empezamos con el de fuerza bruta en el que veremos solo obtendremos el número de inversiones, no ordenaremos también el array que estamos analizando.

```
countInversion(array){
    inversiones = 0

    para i = 1 hasta n
        para j = 1 hasta n
            si(array[i] > array[j]){
                inversiones++
            }

    return inversiones
}
```

Como observamos en el pseudocódigo anterior recorremos el array de izquierda a derecha buscando aquellos elementos que estén en una posición menor que la que le corresponde y vamos aumentando el contador de inversiones hasta que recorremos todo el array por cada posición del elemento.

Ahora vamos a ver como seria aplicando la técnica de Divide y Vencerás junto al algoritmo Merge Sort de ordenación, con la modificación de ir contando las inversiones que hacemos en el array para resolver nuestro problema.

```
countStep(array, izq, der){
    si(izq = der){ //caso base de un solo elemento.
        arrayOrdenado.add(izq)
        return arrayOrdenado
    }
    mitad = (izq + der)/2
    arrayIzq = countStep(array, izq, mitad)
    arrayDer = countStep(array, mitad+1, der)

    //Seudocodigo del merge
    i = izq
    j = mitad + 1
    mientras((i < mitad) && (j < der)){
        si(der(j) < izq(i)){
            arrayOrdenado.add(der(j))
            inversion++
            j++
        }
        sino{
            arrayOrdenado.add(izq(i))
            i++
        }
    }
    //Sa añden los que sobran si sobran por algun lado
    mientras(j  < der){
        arrayOrdenado.add(der(j))
        j++
    }
    mientras(i  < mitad){
        arrayOrdenado.add(izq(i))
        i++;
    }
    return arrayOrdenado
}
```

Vamos a destacar varias cosas para entender el análisis que viene a continuación. La primera de ella es nuestro caso base, que será cuando tengamos un array de un solo elemento. Tendremos que calcular la mitad del array, esta será la división que llevamos a cabo por la estrategia de divide y vencerás. Volvemos a llamar a la función para ir dividiendo una y otra vez nuestro array hasta obtener el caso base. Con el caso base lo que tendremos que hacer ahora es juntar dichos array de forma ordenada, comparando sus elementos e insertando el elemento correspondiente a nuestro array. Finalmente nos aseguramos de que todos los elementos están dentro de nuestro array, si no serán o mayores o menores así que los insertamos por la derecha o izquierda según corresponda. Destacar que lo que se devuelve con

este algoritmo es el array ordenado, por lo que nuestra variable inversión es un atributo de la clase y como tal tenemos que llamar a su método “get” para conocer su valor.

3. Análisis de complejidad y tiempo de ejecución

Vamos a analizar primero la complejidad del algoritmo por fuerza bruta. Tenemos 2 bucles que van desde i = 1 hasta n. Este algoritmo no tiene un caso mejor o un caso peor, puesto que la condición de parada del bucle es n y por tanto va a recorrer todo el array, este ordenado o no. La única diferencia en la ejecución es que devolverá 0 en la variable de inversiones cuando el array este ordenado, pero aun en este caso se recorre al completo el array.

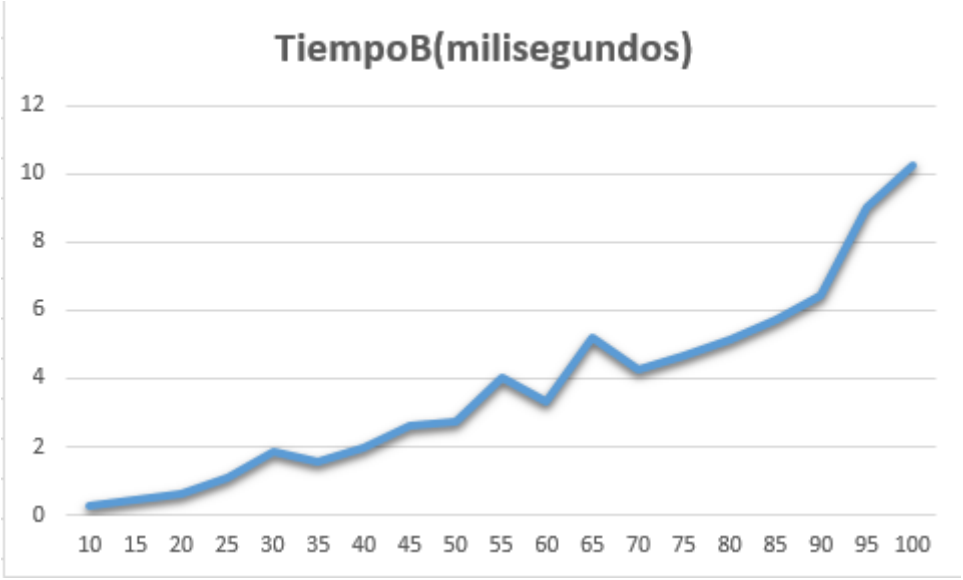
Por lo tanto con lo expuesto anteriormente podemos afirmar que para cualquier caso, la complejidad del algoritmo es n^2. Lo expresaremos de la manera $\Theta(n^2)$, que también coincide con el tiempo que empleara para encontrar la solución.

Comprobaremos de forma experimental que el análisis realizado anteriormente se cumple, para ello realizamos ejecuciones del algoritmo para diferentes tamaños n y mediremos el tiempo que tarda en ejecutarse.

En la siguiente tabla podemos ver con más detalle las diferentes ejecuciones, el número de elementos, las inversiones realizadas (Cuenta) y el tiempo en milisegundos que tarda en resolver el problema nuestro algoritmo. Para todas estas pruebas se han generado números aleatorios para rellenar el vector usando una semilla de 370. El vector resultante se le ha pasado a ambos algoritmos para que la comparación entre ambos fuera coherente.

Elementos	CuentaB	TiempoB(milisegundos)
10	14	0,28
15	53	0,431
20	117	0,64
25	153	1,11
30	225	1,88
35	317	1,56
40	408	1,95
45	536	2,602
50	603	2,73
55	687	4
60	835	3,35
65	913	5,21
70	989	4,29
75	1167	4,68
80	1356	5,14
85	1585	5,72
90	1702	6,42
95	1869	9,03
100	2142	10,23

En la siguiente grafica representamos el tiempo que tarda en resolver el problema para cada uno de los tamaños que le hemos pasado. Vemos como dicho tiempo se va incrementando de manera considerable.



Por otra parte tenemos el algoritmo recursivo. Este algoritmo es una variación del Merge Sort en el cual como explicamos antes, añadimos el conteo de las inversiones para dar información extra con respecto al array y de esa forma también resolver el problema planteado.

En este caso tendremos que ir analizando las diferentes partes del algoritmo para descubrir su complejidad y su tiempo de ejecución teórico y real.

Vamos a analizar la complejidad del algoritmo recursivo. Empezamos analizando el árbol de recursividad, en el cual, observamos que los casos bases se dan en el nivel $\log_2(n)$, por lo tanto nuestros niveles de recursividad irán desde i=0 hasta i = $\log_2(n)$. El tiempo empleado por el algoritmo para combinar los sub-problemas es del orden de Θ (n). Por lo que para resolver el problema el algoritmo emplea un tiempo proporcional a n log (n). Como conclusión a nuestro análisis obtenemos que nuestra complejidad es Θ (n log (n)).

Con el análisis anterior podemos calcular el tiempo que tardara el algoritmo en encontrar la solución.

En cada nivel tenemos 2^j sub-problemas desde j = 1 hasta j = $\log_2(n)$, cada uno de estos sub-problemas tendrán un tamaño m = $\frac{n}{2^j}$. Para una rutina Merge tarda f(m) ≤ 6m. Con esto sabemos que el tiempo de ejecución en el nivel j es: $t(n) \leq 2^j \cdot 6\left(\frac{n}{2^j}\right) = 6n$

En total tenemos $\log_2(n) + 1$ niveles. Con todo lo expuesto podemos asegurar que el tiempo real que empleara el algoritmo para la resolución de un problema de tamaño n es:

$$T(n) \leq 6n \log_2(n) + 6n$$

Teniendo la complejidad y el tiempo, vamos a calcular este último usando la recurrencia general, para abordar el análisis desde todos los frentes posibles. Para ello empezamos a despejar las diferentes variables de la formula.

a = 2. El algoritmo divide el problema inicial siempre por la mitad por lo tanto el número de sub-problemas es 2.

$\frac{1}{b} = \frac{1}{2}$. El tamaño de cada sub-problema siempre es la mitad.

D(n) = Θ (1). Tiempo invertido en dividir cada uno de los problemas.

C(n) = Θ (n). Como explicamos antes el tiempo invertido en combinar los sub-problemas es n.

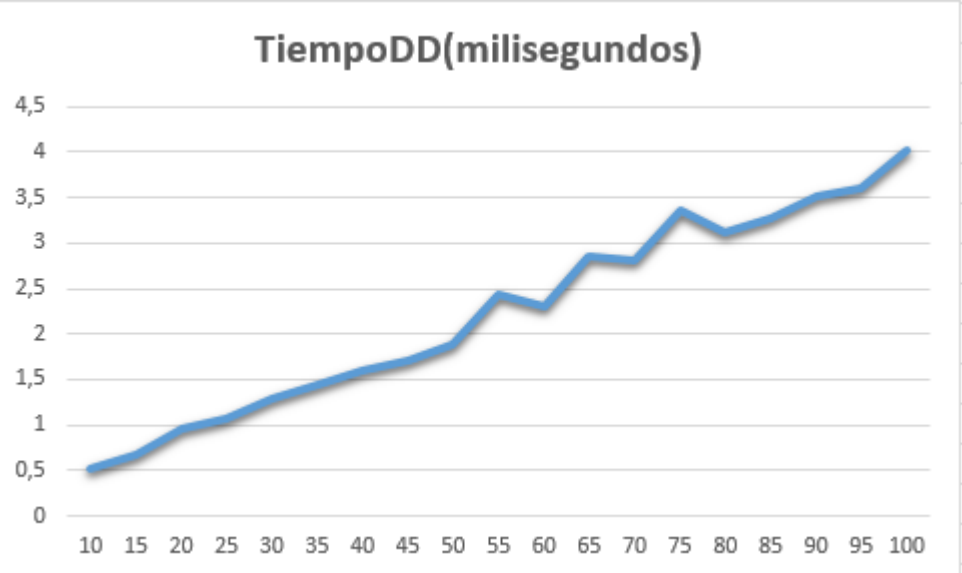
Finalmente:

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

Una vez terminado el análisis teórico vamos a completarlo con los tiempos medidos de forma real con el código, tal y como ya lo hicimos en el algoritmo de fuerza bruta.

Al igual que para el algoritmo de fuerza bruta vamos a mostrar los valores que hemos obtenido en las pruebas en la tabla y ver su evolución con una gráfica.

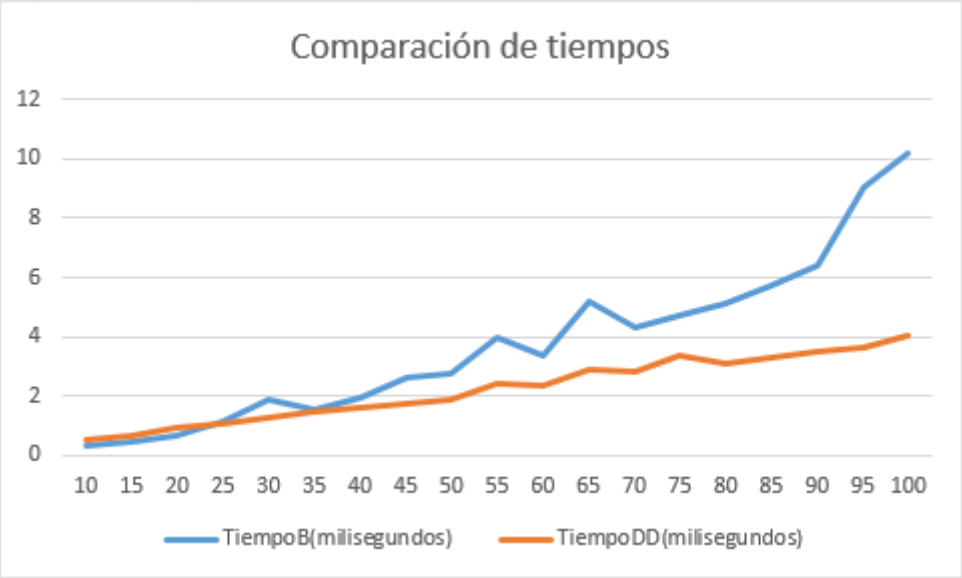
Elementos	CuentaDD	TiempoDD(milisegundos)
10	14	0,51
15	53	0,68
20	117	0,95
25	153	1,07
30	225	1,29
35	317	1,45
40	408	1,6
45	536	1,71
50	603	1,88
55	687	2,44
60	835	2,31
65	913	2,86
70	989	2,82
75	1167	3,35
80	1356	3,11
85	1585	3,27
90	1702	3,51
95	1869	3,6
100	2142	4,01



4. Conclusión

Como era de esperar el algoritmo de fuerza bruta tarda más en completar la tarea que nuestro algoritmo recursivo. Esto lo podemos demostrar comparando sus complejidades, ya que $\Theta(n^2) > \Theta(n \log_2(n))$ para cualquier tamaño de n.

Los análisis de complejidad para casos sencillos como estos, nos demuestran que un buen diseño del algoritmo nos proporcionara unos resultado mucho más óptimos, a pesar de que la implementación de este último es considerablemente más complicado que el primer planteamiento.



En esta gráfica podemos observar la diferencia de tiempos entre el algoritmo de fuerza bruta (azul) y el de Divide y Vencerás (naranja) con lo que demostramos lo expuesto anteriormente.

5. Referencias bibliográficas

1- http://www.geeksforgeeks.org/counting-inversions/