Problema del sub-array de máxima suma

Introducción y descripción del problema

Se pretende estudiar la técnica de resolución de problemas Divide y Vencerás. Esta técnica consiste en ir dividiendo un problema en sub-problemas cada vez más pequeños hasta que sean resolubles rápidamente. Luego las soluciones de los sub-problemas se combinarán para conseguir la solución al problema completo.

En este proyecto se van a realizar dos pequeñas implementaciones del problema del **sub-array de suma máxima** que luego serán comparadas y analizadas.

Este problema pretende encontrar el subvector dentro de un vector en el que la suma de sus números sea la máxima. El array puede contener números negativos (si no fuese así, la solución del problema sería siempre el vector completo).

Resolución

En primer lugar se implementó un algoritmo de fuerza bruta:

```
Array maxSubArrayFB(Array array){
     max, inic, fin = 0;

     para (i = 0 in array.size(){
          suma = 0;
          para (j = i in array.size()){
               suma += array.get(j);
               si (suma > max){
                    max = suma;
                    inic = i;
                    fin = j;
               }
          }
     }

     return array[inic-to-fin];
}
```

El código consta de dos bucles anidados, en los que se definen los límites del sub-array (el primero fija el límite de la izquierda y el segundo, el de la derecha). Se comprueban de forma exhaustiva las sumas de todos los sub-arrays posibles y se devuelve aquel que tenga mayor resultado.

La complejidad de este algoritmo es sencilla de calcular. Al estar basado en dos bucles *for* anidados que dependen del tamaño del vector (n), podemos afirmar que el tiempo de ejecución $T(n) = (n^2)$.

Este tiempo de ejecución es polinomial, así que no está mal del todo, sin embargo, puede ser mejorado significativamente utilizando la técnica *Divide y Vencerás*. El código implementado utilizando esta estrategia es el siguiente:

```
SubArray maxSubArrayD&C(Array array, inic, fin){
     medio = 0;

     si (inic == fin)
          return Array(inic, fin, array[inic]);

     si no {
          medio = floor((inic + fin) / 2);

          SubArray izq = maxSubArrayD&C(array, inic, medio);
          SubArray der = maxSubArrayD&C (array, medio+1, fin);
          SubArray cruce = maxSubArrayCruce(array, inic, medio, fin);

          si(izq.getSuma() >= der.getSuma() && izq.getSuma() >= cruce.getSuma())
               return new SubArray(izq.getInic(), izq.getFin(), izq.getSuma());
          si(der.getSuma() >= izq.getSuma() && der.getSuma() >= cruce.getSuma())
               return new SubArray(der.getInic(), der.getFin(), der.getSuma());
          si no
               return new SubArray(cruce.getInic(), cruce.getFin(), cruce.getSuma());

     }
}
```

Como se puede observar en el pseudocódigo, **maxSubArrayD&C** es una función recursiva que se llama a sí misma múltiples veces dividiendo el problema a la mitad en cada llamada. Para entender la función de manera intuitiva, simplemente tenemos en cuenta que el sub-array de máxima suma puede estar en la mitad izquierda del array principal, en la derecha, o cruzando el punto medio. La función comprueba de forma recursiva la suma estos tres posibles sub-arrays y retorna el mayor. En caso de que el sub-array sólo contenga un elemento simplemente lo retornará.

La recursividad que presenta el algoritmo es la siguiente: $T(n) = \Theta(1) + 2T(n/2) + \Theta(?) + \Theta(1)$ siendo $\Theta(?)$ la complejidad de la función **maxSubArrayCruce** y los $\Theta(1)$ la complejidad del caso base y de los tres condicionales del final. Podemos simplificarla por tanto y dejarla como $T(n) = 2T(n/2) + \Theta(?)$.

Para calcular $\Theta(?)$ se debe analizar la función que calcula la mayor suma cruzando el punto medio:

```
SubArray maxSubArrayCruce(Array array, inic, medio, int fin){
      izqSuma = -infinito;
      suma = 0;
      izqIndice = 0;
      para (i = medio; i >= inic; i--){
            suma += array[i];
            si (suma > izqSuma){
                  izqSuma = suma;
                  izqIndice = i;
            }
      }
      derSuma = -infinito;
      suma = 0;
      derIndice = 0;
      para (i = medio + 1; i <= fin; i++){
            suma += array[i];
            si (suma > derSuma){
                  derSuma = suma;
                  derIndice = i;
            }
      }
      return new SubArray(izqIndice, derIndice, derSuma + izqSuma);
}
```

Esta función busca desde el punto medio hasta los límites del sub-array actual y devuelve aquel cuya suma sea máxima. Al volver hacia arriba en la recursión se van comparando las sumas máximas de cada uno de los sub-problemas y se selecciona la mayor de cada uno.

Consta de dos bucles *for* que recorren cada uno la mitad del array, por lo que la complejidad de la función sería $T(n) = \Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) = \Theta(n)$. Por lo que la complejidad del proceso completo queda de la siguiente forma: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

En cada llamada de la recursión el problema se divide en dos sub-problemas de la mitad de tamaño y se hace un trabajo de complejidad $\Theta(n)$. Esta recursividad puede resolverse tanto por el método maestro como por el árbol de recursividad.

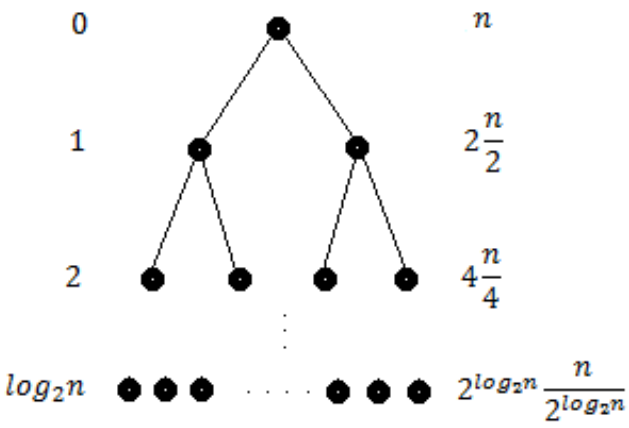
T(n)=2T(n/2)+(n)

- a=2
- b=2
- d=1

Nos encontramos en el primer caso en el que $a = b^d$, por lo tanto:

$$T(n) = n\,logn$$

Si lo resolvemos mediante el árbol de recursividad el resultado debería ser el mismo:



El árbol tendrá $log_2 n$ niveles, cada uno tendrá el doble de nodos que el anterior y cada uno de los nodos hará la mitad del trabajo:

$$T(n) \sum_{i=0}^{log_2 n} n = n\, log_2 n$$

$$T(n) = \Theta(n\, log n)$$

Experimentos

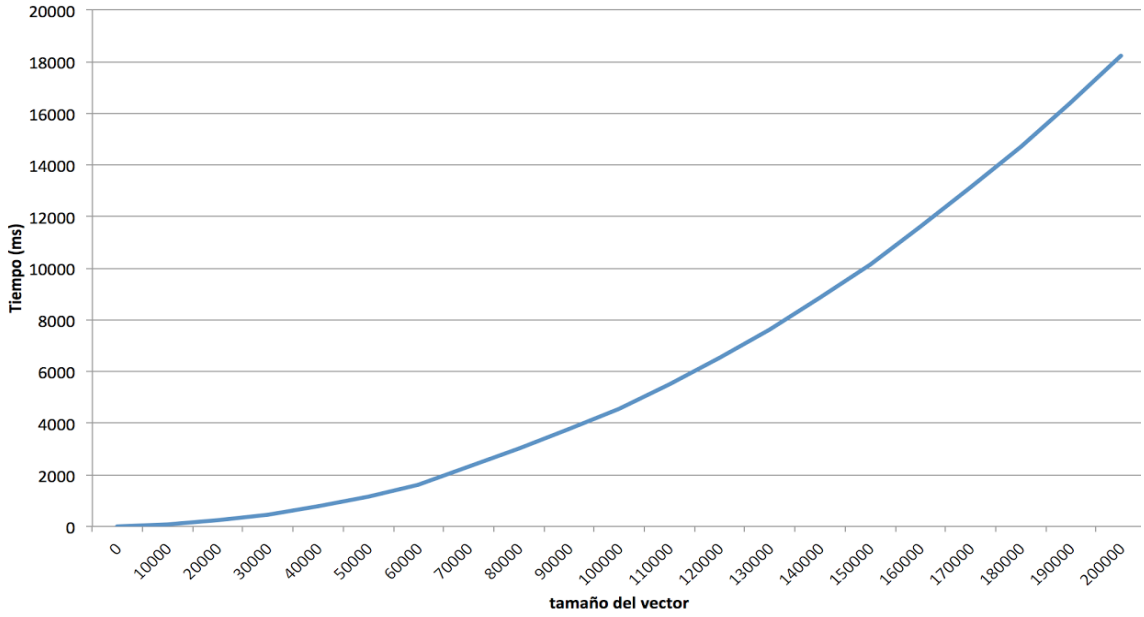
Para comprobar las conclusiones a las que llegamos durante el análisis de los algoritmos, se han hecho una serie de experimentos midiendo los tiempos que tarda la ejecución de cada algoritmo para vectores de distintos tamaños formados por números aleatorios en el intervalo [-10, 10].

Las primeras medidas se realizaron con el algoritmo de fuerza bruta y se obtuvieron los siguientes resultados:

Tamaño del vector	Tiempo (ms)
10000	61
20000	258
30000	464
40000	770
50000	1160
60000	1596
70000	2315
80000	3031
90000	3746
100000	4542

Tamaño del vector	Tiempo (ms)
110000	5512
120000	6526
130000	7637
140000	8856
150000	10166
160000	11606
170000	13139
180000	14711
190000	16390
200000	18248

Si se representan los valores obtenidos en una gráfica obtenemos lo siguiente:



Se aprecia perfectamente el comportamiento cuadrático que habíamos predicho en el análisis.

A continuación se realizó el mismo experimento con el algoritmo *Divide y Vencerás.* Dado que este algoritmo es mucho más eficiente, cada tamaño de entrada se incrementa en 50000 con respecto al anterior, en lugar de en 10000 como en el experimento anterior. Los datos obtenidos son los siguientes:

Tamaño del vector	Tiempo (ms)
50000	31
100000	63
150000	46
200000	45
250000	63
300000	84
350000	140
400000	104
450000	118
500000	111
550000	117
600000	157
650000	150
700000	171
750000	150

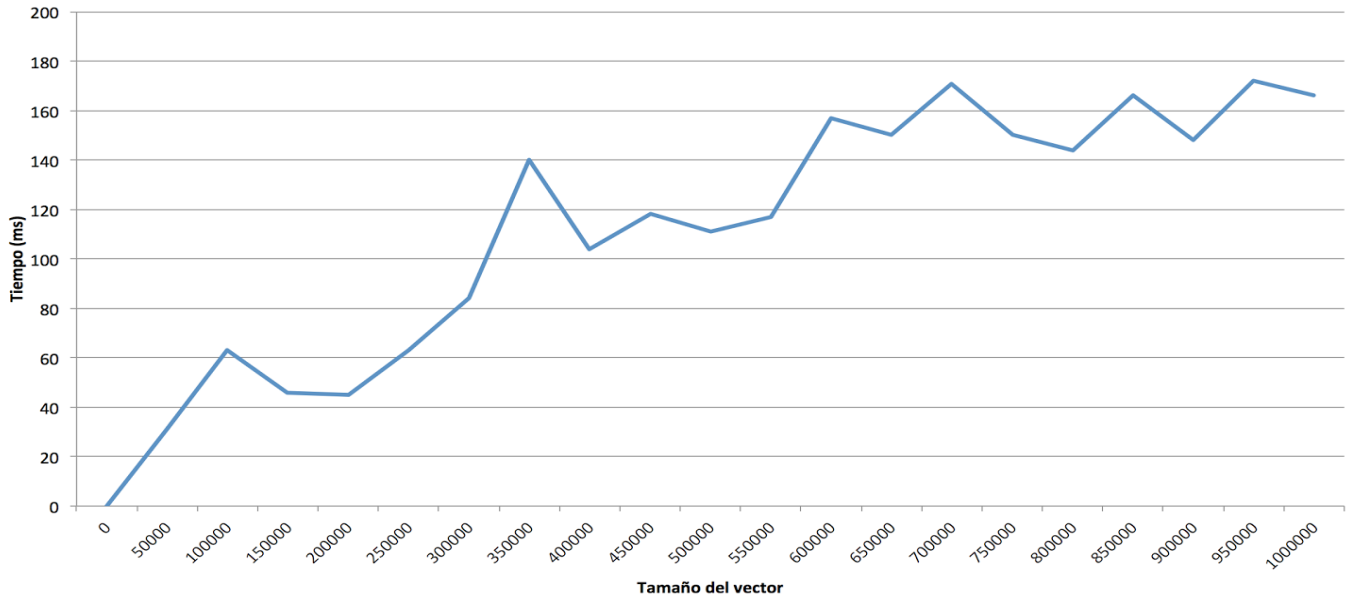
Tamaño del vector	Tiempo (ms)
800000	144
850000	166
900000	148
950000	172
1000000	166
1050000	232
1100000	226
1150000	217
1200000	202
1250000	195
1300000	215
1350000	217
1400000	218
1450000	210
1500000	226

En este caso la gráfica generada no ha cuadrado con nuestra predicción. Los valores no siguen un comportamiento intuitivo. La conclusión a la que se llegó es que este algoritmo es dependiente de la entrada. En el primer algoritmo, el vector de tamaño n se recorre un número n^2 de veces, independientemente de los números que contenga. En el segundo, ocurre algo similar en los bucles *for* pero hay que tener en cuenta que hay una serie de instrucciones condicionales que hacen variar el tiempo de ejecución.

Si un vector contiene números positivos en sus posiciones pares y negativos en sus posiciones impares y a la vez estos números decrecen conforme se avanza en el vector, siempre el subvector de la izquierda va a ser el mayor, por lo tanto, en la función **maxSubArrayD&C**, solo se tendrá que comprobar uno de los condicionales.

De la misma forma, si todos los elementos de un array son negativos, el número de veces que se acceda a los condicionales que se encuentran en la función **maxSubArrayCruce** será cero, por lo tanto el número de instrucciones y el tiempo de ejecución disminuirán.

De cualquier forma, la gráfica obtenida es la siguiente:



Aunque hay una serie de picos en la función, se puede apreciar que su tendencia es más similar a la función $f(n) = log(n)$ que a la $f(n) = nlog(n)$ predicha. Sin embargo, hay que tener en cuenta que, como se ha comentado, el número de instrucciones ejecutadas es muy dependiente de la entrada, por lo que es normal que se produzcan desviaciones. De cualquier forma, ahora que hemos visto cómo se comporta el algoritmo experimentalmente, es necesario rectificar los cálculos realizados en el apartado de análisis:

Aunque los cálculos eran correctos, se determinó el tiempo de ejecución $T(n) = \Theta(n\ logn)$. Sin embargo, como se ha explicado, esto no es correcto, ya que este tiempo define únicamente una cota superior para el algoritmo: $T(n) = O(n\ logn)$.

Conclusiones

- Para un resultado no trivial siempre será necesaria la existencia de enteros **negativos** en el array, de caso contrario el array de suma máxima siempre será el array completo.
- El algoritmo de fuerza bruta es de orden cuadrático (n^2) y de alto coste.
- El algoritmo de *Divide & Conquer* es la mejor solución a este problema, encuentra la solución en un orden $T(n) = O(n\ logn)$.
- Con los experimentos producidos se demuestra que el tiempo de ejecución del algoritmo *D&C* mejora en mucho el desempeño del *Brute Force*.
- Los algoritmos no siempre dan la misma solución en el caso de que haya varios sub-arrays del mismo tamaño.