Algorítmica - Practica 2: Divide y Vencerás

A. Herrera, A. Moya, I. Sevillano, J.L. Suarez

10 de abril de 2015

Contents

1 Organización de la práctica

La práctica 2 trata sobre el desarrollo de algoritmos siguiendo el paradigma divide y vencerás. El problema que nuestro grupo debe resolver es el 6. Hemos desarrollado además varios algoritmos para el problema opcional que se exponen en su correspondiente apartado.

Para cada problema se sigue la siguiente estructura:

- Enunciado del problema
- Resolución teórica del problema (con una subsección por algoritmo)
- Análisis empírico. Análisis de la eficiencia híbrida

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la eficiencia híbrida para los mismos.

Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Toshiba
- **RAM:** 8 GB
- **Procesador:** Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

El código, los resultados de las ejecuciones, las gráficas y los pdf asociados se puede encontrar en GitHub.

2 Problema 6

2.1 Enunciado del problema

Sea un vector v de números con n componentes, todas distintas, de forma que existe un índice p (que no es ni el primero ni el último) tal que a la izquierda de p los números están ordenados de forma creciente y a la derecha de p están ordenados de forma decreciente, es decir:

$$\forall i, j \leq p \text{ con } i < j \Rightarrow v[i] < v[j] \text{ y } \forall i, j \geq p \text{ con } i < j \Rightarrow v[i] > v[j] \quad (1)$$

Lo que implica que el máximo se encuentra en p. Diseñe un algoritmo "divide y vencerás" que permita determinar p. ¿Cuál es la complejidad del algoritmo?. Compárelo con el algoritmo "trivial" para realizar esta tarea. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Un ejemplo de vector unimodal es el dado en la Imagen 1, en la cual se señala el buscado elemento p. Se harán referencias a la imagen a lo largo del texto.

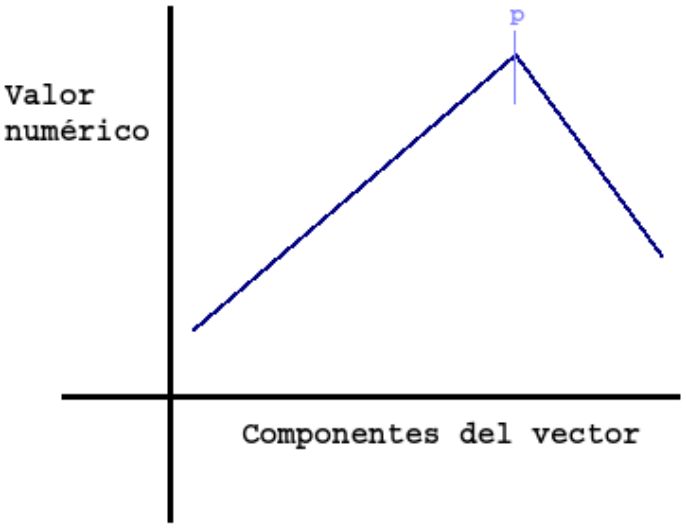

Imagen 1. Representación de un vector unimodal.

2.2 Resolución teórica del problema

Daremos dos soluciones al problema, una trivial y otra basada en divide y vencerás. Como es de esperar, la primera tendrá un orden de eficiencia lineal mientras que con la segunda conseguiremos alcanzar el deseado orden $\log n$.

En primer lugar, nótese que dada la forma del vector, caracterizada por (1), el elemento p es el máximo del vector. Esto implica que p verifica (2):

$$v[p-1] < v[p] > v[p+1] \quad (2)$$

Y, además, es el único elemento del vector que verifica (2) pues en caso caso contrario no se mantendría el invariante (1) del vector. Es por tanto el único máximo local. Utilizaremos este hecho para el desarrollo de ambos algoritmos.

2.2.1 Solución trivial

La idea es sencilla: recorrer el vector hasta encontrar el punto p. Recordemos que la propiedad (2) caracteriza al punto p. Por tanto, en nuestro algoritmo trivial basta recorrer todas las componentes $2, 3, ..., n-1$ y encontrar la primera componente que verifica (2), que será p por unicidad.

Dado que nuestro recorrido es lineal de izquierda a derecha podemos obviar la primera comparación $v[p-1] < v[p]$ pues si no hemos encontrado todavía p es porque $v[p-2] < v[p-1] < v[p-1]$.

2.2.1.1 Pseudocódigo.

Se proporciona el pseudocódigo a continuación:

```
# Pseudocódigo del algoritmo trivial.
# v es el vector unimodal con n componentes.
for i in range(1, n-1):
    if v[i] > v[i+1]:
        p = i
        break
```

En el caso de la Imagen 1, el algoritmo recorrería el vector de izquierda a derecha hasta encontrar el elemento p, donde la pendiente se vuelve negativa.

2.2.1.2 Ejemplo.

Veamos un ejemplo del algoritmo. Consideramos el vector:

$$\text{v} = [1\ 3\ 5\ 7\ 8\ 6\ 4]$$

Empezamos el recorrido en la componente de índice 1. Esta es menor que $v[2] = 5$ luego continuamos la búsqueda. Repetimos este hecho hasta que llegamos a la componente $v[4] = 8$. Tenemos que $v[5] = 6 < v[4] = 8$ y por ello se tiene que $p = 4$.

2.2.2 Solución mediante divide y vencerás

Queremos obtener una solución con un orden de eficiencia $\theta(\log n)$. Para ello deberímos librarnos de la búsqueda lineal. Un problema similar y ampliamente conocido es el de búsqueda en un vector. La búsqueda lineal sirve en vectores desordenados. Pero en el caso de ser ordenados, podemos hacerlo mucho mejor con una búsqueda binaria, que sigue las directrices del paradigma de diseño de algoritmos divide y vencerás. Nos inspiraremos en la búsqueda binaria para el diseño de nuestro algoritmo.

En primer lugar, comprobamos si el elemento en la posición mitad del vector verifica (2). En tal caso, este elemento debe ser p, devolviendo su índice como resultado del algoritmo. En el caso de no ser p recurrimos a divide y vencerás. Para ello, tal y como sucede en la búsqueda binaria, obviaremos uno de los dos vectores mitades (izquierda y derecha) llamando a nuestro algoritmo recursivo sobre el otro subvector.

La cuestión es la siguiente: ¿sobre qué subvector aplicar la recursividad y por qué?.

Consideremos el vector unimodal de la Imagen 1. Los elementos a la izquierda de p están ordenados de menor a mayor y los que se encuentran a su derecha de mayor a menor. Encontramos la componente mitad del vector, llamémosla $m1$. El vector se encuentra dividido tal y como muestra la Imagen 2.

Es claro que la componente p se queda en el subvector derecha. Podemos ahora abstraer la razón de este hecho. Esta es:

$$v[m1-1] < v[m1] < v[m1+1]$$

Como $v[m1] < v[p]$ y nos encontramos en el subvector creciente, p debe estar a la derecha de $m1$ tal y como sucede en la imagen.

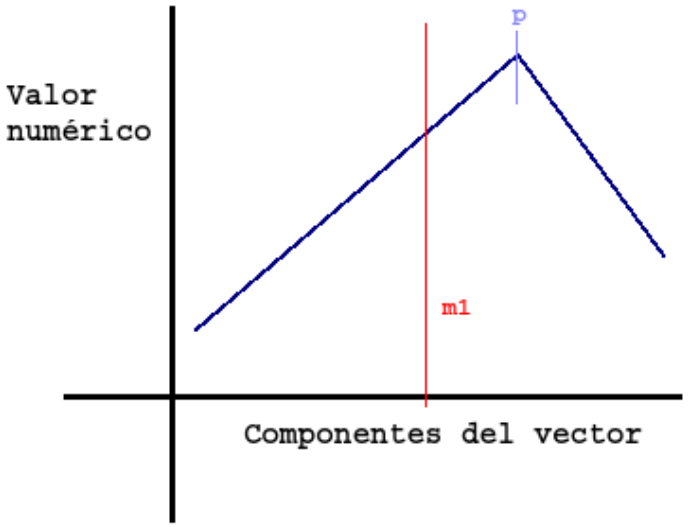

Imagen 2. Componente mitad del vector unimodal.

Utilizamos ahora recursividad sobre las componentes $\{m1+1, ..., n-1\}$. La Imagen 3 muestra esta situación. Obtenemos la componente mitad de este nuevo subvector, denotémosla $m2$, que tampoco verifica (2) y por ello no es p. Pero en este caso se tiene que $m2$ cumple:

$$v[m2-1] > v[m2] > v[m2+1]$$

Por tanto, estamos situados en la zona decreciente del vector v y por ello p se encuentra a la izquierda de $m2$ ($v[p] > v[m2]$). El subvector sobre el que se volvería a aplicar la recursividad se ha resaltado en gris. Se debe repetir el proceso hasta dar con el elemento p.

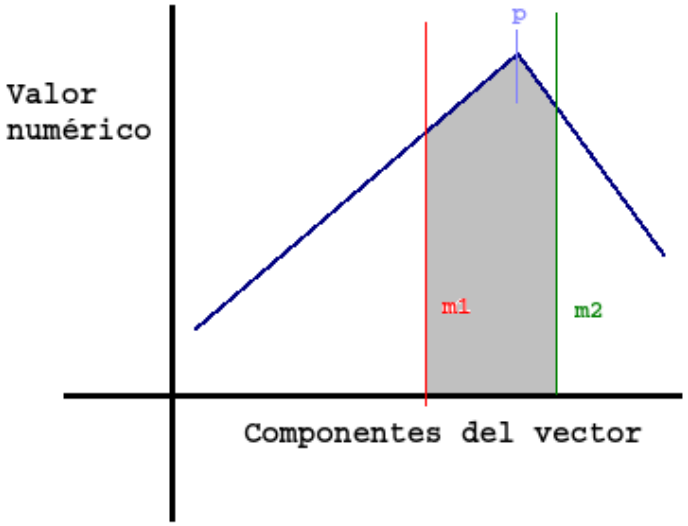

Imagen 3. Segunda iteración sobre el vector unimodal.

2.2.2.1 Pseudocódigo. Formalizando lo anterior, podemos dar el siguiente pseudocódigo:

```
# Algoritmo recursivo que resuelve el problema en O(log n).
# El vector dado como parámetro se presupone unimodal.
# Parametros:
#  - v      : Vector sobre el que se realiza la búsqueda.
#  - inicio : Posición de inicio del subvector a considerar.
#  - final  : Posición siguiente a la última del subvector.
def obtenerP(v, inicio, final):
    mitad = (inicio + final) / 2
    if mitad > inicio:
        if v[mitad-1] < v[mitad] and v[mitad] > v[mitad+1]:
            return mitad
        else if v[mitad-1] < v[mitad]:
            return obtenerP(v, mitad+1, final)
        else:
            return obtenerP(v, inicio, mitad)
    else:
        return inicio
```

2.2.2.2 Ejemplo. Veamos un ejemplo del funcionamiento del algoritmo. Consideramos el mismo vector que para el algoritmo lineal:

$$\text{v} = [1\ 3\ 5\ 7\ 8\ 6\ 4]$$

Este vector consta de 7 componentes. Inicialmente, `inicio = 0` y `final = 7`. Se tiene que la componente mitad tiene índice $3 = (0 + 7)\ /\ 3$. Sin embargo, $v[2] = 5 < v[3] = 7 < v[4] = 8$. Debemos pues aplicar la recursividad en el subvector derecha, `return obtenerP(v, mitad+1, final)`.

$$\text{v} = [-\ -\ -\ -\ 8\ 6\ 4]$$

En este caso, la componente mitad tiene índice 5 y valor 6. Verifica $v[4] = 8 < v[5] = 6 < v[7] = 4$. En este caso se aplica la recursividad sobre el vector izquierda:

$$\text{v} = [-\ -\ -\ -\ 8\ -\ -]$$

Este contiene un único elemento, el 8, cuyo índice se devuelve como p.

Si comparamos el ejemplo con el del algoritmo lineal, aunque sea más antinatural el algoritmo recursivo realiza menos iteraciones (solo 3) en comparación con las 4 realizadas por el lineal. En problemas grandes la diferencia es incluso mayor como se verá en el siguiente apartado.

2.2.2.3 Eficiencia teórica. En cada llamada recursiva del algoritmo se hace un trabajo constante para calcular el punto medio y decidir en cual de los tres posibles casos estamos. En el peor de ellos, nunca se encuentra p y siempre se debe hacer una llamada recursiva a un vector de tamaño mitad. Si denotamos $T(n)$ a función con la eficiencia del algoritmo, en el peor caso esta verifica:

$$T(n) = T(\frac{n}{2}) + \theta(1)$$

Tenemos una ecuación en diferencias lineal de fácil solución. Tomando $n = 2^k$:

$$T(n) = T(2^k) = T(2^{k-1}) + \theta(1) = \ldots = T(1) + (k-1)\theta(1) \in \theta(k) = \theta(\log_2 n)$$

Efectivamente, como ya se había adelantado, el algoritmo en el peor caso es $\theta(\log_2 n)$.

2.3 Análisis empírico. Análisis de la eficiencia híbrida

Para analizar de forma empírica el comportamiento de ambos algoritmos hemos recurrido al pograma `generar_unimodal.cpp` proporcionado para la práctica que genera vectores unimodales, es decir, verifican (1), de un tamaño dado como parámetro. Para su uso hemos desarrollado un script `generar_test_cases.sh` que compila y utiliza el programa anterior para crear 100 vectores unimodales cuyo tamaño varía de 1000 en 1000 desde 1000 hasta 100000.

Mediante otro script llamado `comparar_algoritmos.sh` se ejecutan ambos algoritmos sobre todos los vectores generados, obteniendo el tiempo de ejecución para cada uno de ellos. Los resultados en un archivo para cada algoritmo compuesto de una columna con el tamaño de los vectores y otra con el tiempo de ejecución. Posteriormente, el script llama a `plot_resultados.sh`, que utiliza GNU-PLOT para generar la Imagen -.

Hemos realizado x ejecuciones de ambos algoritmos para tama

3 Opcional: Problema 2

3.1 Enunciado del problema

Muchos sitios web intentan comparar las preferencias de dos usuarios para realizar sugerencias a partir de las preferencias de usuarios con gustos similares a los nuestros. Dado un ranking de n productos (p.ej. películas) mediante el cual los usuarios indicamos nuestras preferencias, un algoritmo puede medir la similitud de nuestras preferencias contando el número de inversiones: dos productos i y j están "invertidos"" en las preferencias de A y B si el usuario A prefiere el producto i antes que el j, mientras que el usuario B prefiere el producto j antes que el i. Esto es, cuantas menos inversiones existan entre dos rankings, más similares serán las preferencias de los usuarios representados por esos rankings.

Por simplicidad podemos suponer que los productos se pueden identificar mediante enteros $1, \ldots, n$, y que uno de los rankings siempre es $1, \ldots, n$ (si no fuese así bastaría reenumerarlos) y el otro es $a_1, a_2, \ldots, a_n$, de forma que dos productos i y j están invertidos si $i < j$ pero $a_i > a_j$. De esta forma nuestra representación del problema será un vector de enteros v de tamaño n, de forma que $v[i] = ai, \forall i = 1, \ldots, n$. El objetivo es diseñar, analizar la eficiencia e implementar un algoritmo "divide y vencerás" para medir la similitud entre dos rankings. Compararlo con el algoritmo de "fuerza bruta" obvio. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.