

ALGORÍTMICA

PRÁCTICA 2: DIVIDE Y VENCERÁS



AUTORES:

**Pablo Moreno Megías, Diego Lerena García, Manuel Vallejo Felipe, Ángel Díaz de la Torre,
Francisco Navarro Morales, Marcel Kemp Muñoz y David Redondo Correa**

Algorítmica [PRACTICAS]
Segundo curso del Grado de Ingeniería Informática.
Universidad de Granada.
curso 2016-2017.

Índice

1. Problema	2
2. Algoritmo Obvio. Fuerza Bruta.	2
2.1. Algoritmo	2
2.2. Eficiencia teórica.	3
2.3. Eficiencia empírica.	3
2.4. Eficiencia híbrida.	4
3. Algoritmo Divide y Vencerás.	5
3.1. Algoritmo	5
3.2. Eficiencia teórica.	6
3.3. Eficiencia empírica.	6
3.4. Eficiencia híbrida.	6
4. Comparación y conclusiones.	8

1 Problema

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 ello, el objetivo del problema sería realizar un algoritmo que compruebe el número de inversiones que existen entre un vector ordenado, y otro desordenado, sabiendo que sus datos van desde 0 hasta n-1 sin que ninguno se repita.

2 Algoritmo Obvio. Fuerza Bruta.

2.1. Algoritmo

El algoritmo de fuerza bruta comienza comparando la primera posición del vector que nos han dado con todas las demás posiciones del vector, comprobando si el número elegido (primera posición) es mayor que el número con el que se ha comparado. Si se cumpliese esta condición incrementaríamos una variable auxiliar para saber que está mal posicionado. Esta comparación hay que realizarla n veces con todos los números que se encuentren dentro del vector (primer bucle), y además el número elegido debe de compararse con todos los demás números que aún no se hayan comprobado (segundo bucle). El problema de este algoritmo es que tiene una eficiencia muy mala, ya que daría $O(n^2)$, por culpa de los dos bucles anteriormente explicados.

Listing 1: Algoritmo de fuerza bruta

```
1 int inversiones = 0;
2 clock_t ini, fin;
3 ini = clock();
4 for (int k = 0; k < 10000; k++) {
5     for (int i = 0; i < n; i++) {
6         for (int j = i + 1; j < n; j++) {
7             if (T[j] > T[i]) {
8                 inversiones++;
9             }
10        }
11    }
12 }
13
14 fin = clock();
15 double tiempo = (double)((fin-ini)/(CLOCKS_PER_SEC)*(double)10000);
16 cout << tiempo << endl;
17 }
```

2.2. Eficiencia teórica.

Tenemos dos bucles anidados, uno va desde 0 hasta el número de elementos y el otro (interior) va desde el índice del exterior más uno hasta el final, luego tenemos:

$$\sum_{i=0}^n \sum_{i+1}^n 1$$

donde $\sum_{i+1}^n 1 = n - i$ y, al final tenemos, por tanto, $\sum_{i=0}^n \sum_{i+1}^n 1 = \sum_{i=0}^n n - i$ Luego tenemos:

$$\sum_{i=0}^n n - \sum_{i=0}^n i = n(n+1) - \frac{(n+1)n}{2} = \frac{(n+1)n}{2} = \frac{n^2 + n}{2}$$

Luego la eficiencia teórica del algoritmo de fuerza bruta es $O(n^2)$.

2.3. Eficiencia empírica.

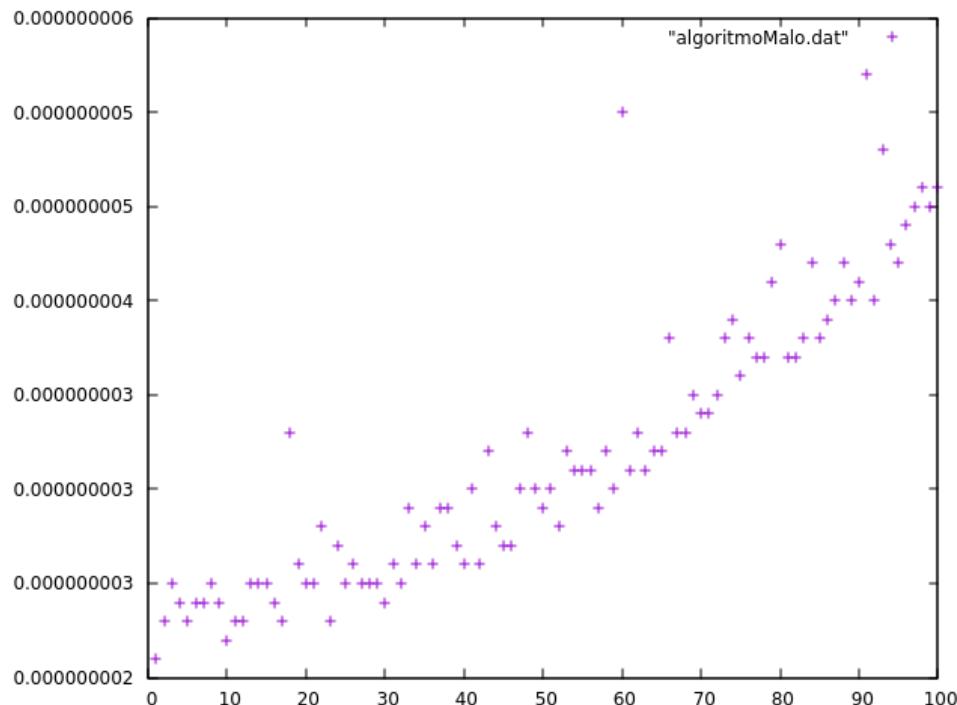


Figura 1: Algoritmo de fuerza bruta eficiencia empírica

2.4. Eficiencia híbrida.

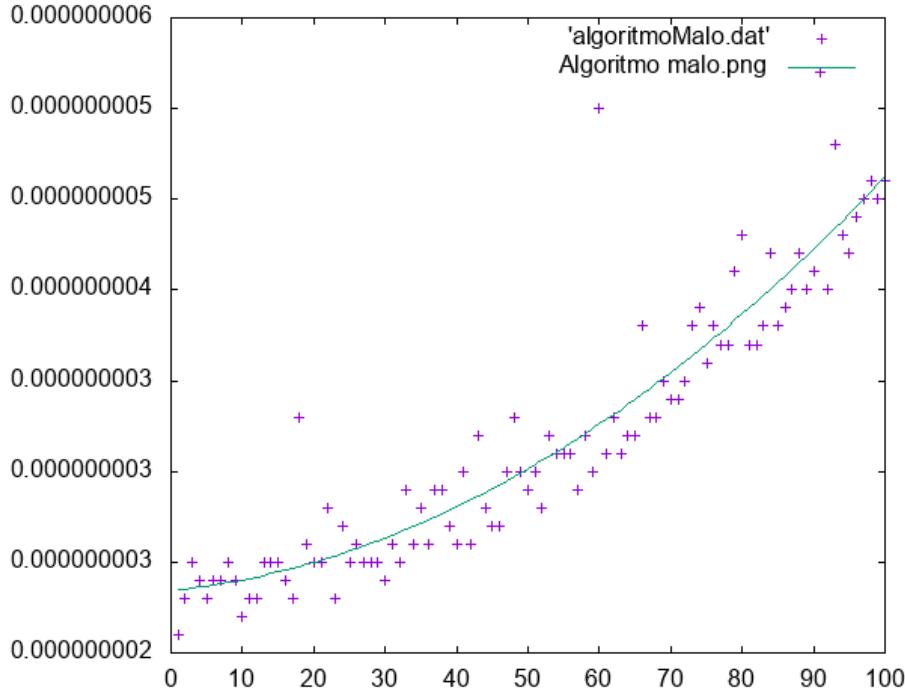


Figura 2: Algoritmo de fuerza bruta eficiencia híbrida

```
gnuplot> fit f(x) 'algoritmoMalo.dat' via a0,a1,a2
[iter      chisq      delta/lim      lambda      a0          a1          a2
 0 2.10236335800e+09  0.000e+00  2.61e+03  1.000000e+00  1.000000e+00  1.000000e+00
 1 4.5388083358e+04 -4.63e+09  2.61e+02 -9.043652e-03  9.842816e-01  9.997059e-01
 2 1.2935174877e+04 -2.51e+05  2.61e+01 -9.397920e-03  7.424351e-01  9.899261e-01
 3 2.2153824265e+01 -5.83e+07  2.61e+00  8.002882e-06 -1.318253e-02  9.450939e-01
 4 1.4528931145e+00 -1.42e+06  2.61e-01  1.217558e-04 -1.468399e-02  3.689627e-01
 5 5.9029213303e-05 -2.46e+09  2.61e-02  7.767048e-07 -9.364698e-05  2.351795e-03
 6 2.4289153079e-13 -2.43e+13  2.61e-03  5.000962e-11 -6.002983e-09  1.531999e-07
 7 7.3859097463e-18 -3.29e+09  2.61e-04  1.875345e-13  4.045488e-12  2.343190e-09
 8 7.3859096464e-18 -1.35e-03  2.61e-05  1.875025e-13  4.049342e-12  2.343093e-09
iter      chisq      delta/lim      lambda      a0          a1          a2
After 8 iterations the fit converged.
final sum of squares of residuals : 7.38591e-18
rel. change during last iteration : -1.35326e-08

degrees of freedom      (FIT_NDF) : 97
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf) : 2.75941e-10
variance of residuals (reduced chisquare) = WSSR/ndf : 7.61434e-20

Final set of parameters            Asymptotic Standard Error
=====================================
a0      = 1.87503e-13    +/- 3.703e-14   (19.75%)
a1      = 4.04934e-12    +/- 3.86e-12    (95.33%)
a2      = 2.34309e-09    +/- 8.447e-11   (3.605%)

correlation matrix of the fit parameters:
      a0      a1      a2
a0      1.000
a1      -0.969  1.000
a2      0.753 -0.871  1.000
```

Figura 3: Ajuste con n^2 del algoritmo fuerza bruta

3 Algoritmo Divide y Vencerás.

3.1. Algoritmo

Listing 2: Algoritmo ‘Divide y vencerás’

```
1 int cuenta_y_combina(int vector[], int izquierda, int derecha) {
2     int mid = (izquierda+derecha)/2;
3     int ret = 0 ;
4     int a = izquierda ;
5     int b = mid+1;
6     int resultado[derecha-izquierda+1];
7     int r = 0;
8
9     while(a <= mid && b <= derecha){
10         if(vector[a] <= vector[b]){
11             resultado[r++] = vector[a++];
12         }else{
13             resultado[r++] = vector[b++];
14             ret += mid - a + 1;
15         }
16     }
17
18     while(a <= mid)
19         resultado[r++] = vector[a++];
20     while(b <= derecha)
21         resultado[r++] = vector[b++];
22
23     for(int i = 0 ; i < derecha-izquierda+1 ; i++){
24         vector[i+izquierda] = resultado[i];
25     }
26
27     return ret;
28 }
29
30 int ContarInversiones(int vector[], int izquierda, int derecha){
31     int x,y,z,mid;
32     if(izquierda >= derecha) return 0 ;
33     mid = (izquierda+derecha)/2;
34
35     x = ContarInversiones(vector, izquierda, mid);
36     y = ContarInversiones(vector, mid+1, derecha);
37     z = cuenta_y_combina(vector, izquierda, derecha);
38
39     return x+y+z;
40 }
```

Nuestra solución consiste en atacar el problema con un esquema de divide y vencerás clásico, basado en dos llamadas recursivas de tamaño $n/2$ y una función para combinar ambas soluciones. La mecánica es: dividimos el problema en dos partes, y cada parte a su vez en dos partes hasta que tengamos problemas de tamaño uno (que tendrán cero inversiones) o de tamaño dos (que tendrán cero o una inversión, dependiendo de si el número mayor está a la izquierda o no). En el caso de tamaño dos, si existe inversión, además de contar dicha inversión, invertimos los dos números para ordenar el vector. Cuando la recursividad ‘se rompe’, nos vemos en la necesidad de, no solo sumar las inversiones en cada mitad en la que dividimos el problema, sino también de contabilizar las inversiones que hay entre los números de una mitad y los de la otra. Para ello, tenemos en cuenta que los números de la mitad de la derecha solo

podrán tener inversiones con los de la izquierda si los de la izquierda son **mayores**; y, dado que además nos interesa ir ordenando el vector conforme contamos inversiones, realizamos lo siguiente: Como ambas mitades están ordenadas, podemos combinarlas en un vector ordenado en tiempo $O(n)$ simplemente manteniendo un índice para cada parte (empezando en el mínimo de cada una) e ir comparando ambos índices. El de menor valor se introduce en un nuevo vector y se aumenta en uno el valor del índice. Cuando uno de los dos vectores se termine, se introducirán en el vector los elementos restantes del otro hasta que este también termine. Ahora bien, para contar las inversiones tenemos que tener en cuenta el detalle de que **los vectores están ordenados**. Así, si el elemento del vector de la derecha es menor que un elemento del de la izquierda (que será menor que todos los elementos restantes de este vector, a su derecha); también es menor que todos los elementos que restan de la parte izquierda. Como hemos dicho, tenemos una inversión por cada número de la parte izquierda que sea menor que algún número de la parte de la derecha, así pues, contabilizamos tantas inversiones como números haya a la derecha del índice de la parte izquierda siempre que el valor del vector en dicho índice sea mayor que en el índice de la parte derecha. Así, al mismo tiempo ordenamos el vector (para facilitar cambios posteriores) y contamos las inversiones.

3.2. Eficiencia teórica.

El algoritmo de basa en dos llamadas recursivas de tamaño $n/2$ y la combinación de los dos subvectores ordenados obtenidos de ello en $O(n)$. La condición de parada es si el tamaño es 0, 1 o 2, y en tal caso tarda $O(1)$. Luego tenemos:

$$t(n) = 1 \text{ para } n < 3 \quad t(n) = 2t(n/2) + n$$

Si realizamos el cambio $n = b^k \rightarrow k = \log_b n$ nos queda:

$$t(2^k) = 2t(2^{k-1} + 2^k) \rightarrow t(2^k) - 2t(2^{k-1}) = 2^k \rightarrow x - 2 = 2^k$$

Tenemos una ecuación lineal no homogénea. El polinomio característico de la parte homogénea es $x-2$, y de la parte no homogénea $x-2$, también. Luego tenemos una raíz única de multiplicidad 2: $pc = (x - 2)^2$ luego la solución general sería:

$$t(2^k) = c_1 2^k + c_2 k 2^k \rightarrow t(n) = c_1 2^{\log_2 n} + c_2 \log_2 n \times 2^{\log_2 n} = c_1 n + c_2 \log_2 n \times n$$

Como no tiene sentido que c_1 y c_2 sean negativos, concluimos que la eficiencia del algoritmo es

$$O(n \times \log_2 n)$$

3.3. Eficiencia empírica.

Al medir los tiempos de forma empírica obtuvimos los resultados mostrados en la figura 4

3.4. Eficiencia híbrida.

Los datos se ajustan perfectamente a la función obtenida de forma teórica, como se aprecia en las figuras 5 y 6

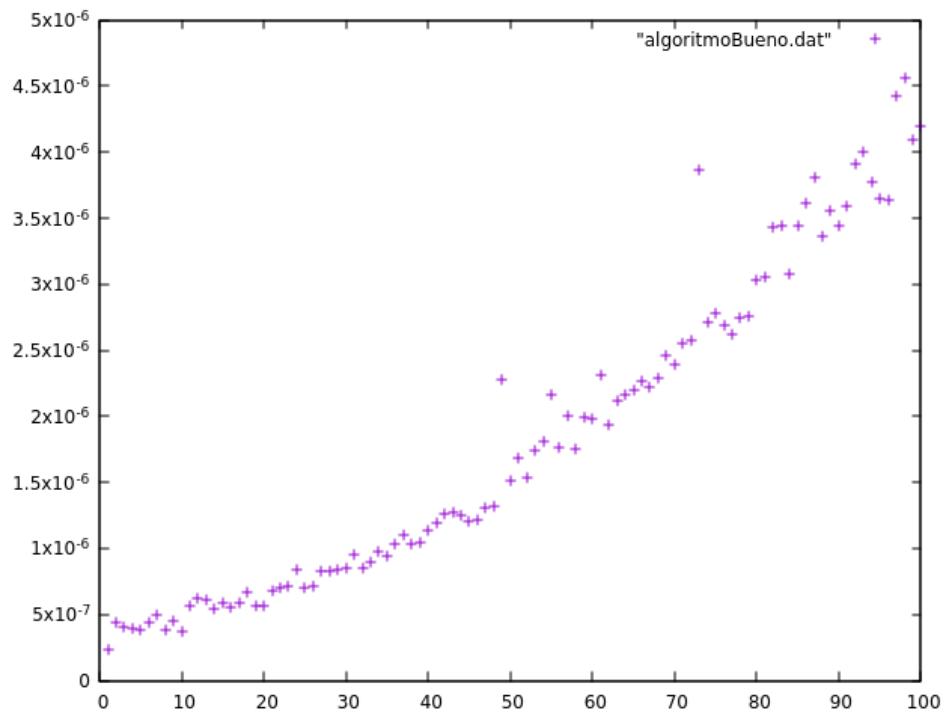


Figura 4: Algoritmo divide y vencerás sin ajuste

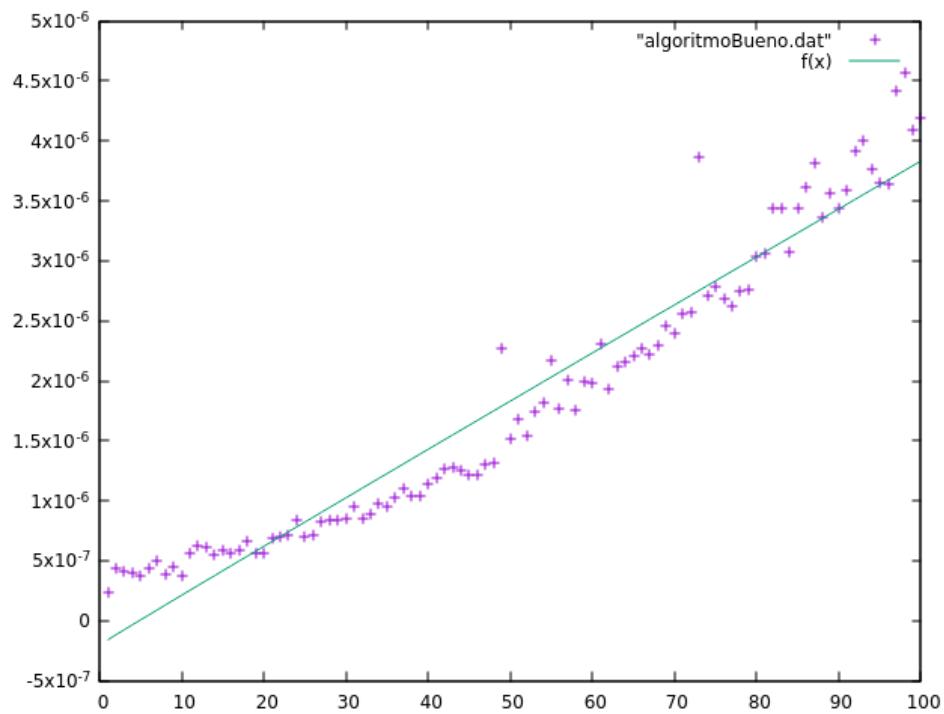


Figura 5: Algoritmo divide y vencerás con ajuste $\log(n)$

```

gnuplot> f(x)=a*x*log10(b*x)+c
gnuplot> fit f(x) 'algoritmoBueno.dat' via a,b,c
iter      chisq      delta/lim   lambda    a          b          c
  0  1.1924949603e+06  0.00e+00  6.42e+01  1.000000e+00  1.000000e+00
  1  3.2693322429e+03 -3.64e+07  6.42e+00  4.774502e-02  7.761336e-01  9.906175e-01
  2  1.2333682355e+01 -2.64e+07  6.42e-01  -3.328431e-03  6.893066e-01  5.715828e-01
  3  1.9230853368e-03 -6.41e+08  6.42e-02  -5.176952e-05  6.986789e-01  7.566151e-03
  4  3.7004784543e-11 -5.20e+12  6.42e-03  1.286127e-08  6.986811e-01  1.103376e-06
  5  5.9114756876e-12 -5.26e+05  6.42e-04  2.111843e-08  6.986811e-01  1.082209e-07
  6  5.9114749889e-12 -1.18e-02  6.42e-05  2.111845e-08  6.986801e-01  1.082197e-07
iter      chisq      delta/lim   lambda    a          b          c

After 6 iterations the fit converged.
final sum of squares of residuals : 5.91147e-12
rel. change during last iteration : -1.18189e-07

degrees of freedom (FIT_NDF) : 97
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 2.46866e-07
variance of residuals (reduced chisquare) = WSSR/ndf : 6.0943e-14

Final set of parameters      Asymptotic Standard Error
=====
a      = 2.11184e-08      +/- 6.038e-09  (28.59%)
b      = 0.69868      +/- 0.8873  (127%)
c      = 1.0822e-07      +/- 1.024e-07 (94.67%)

correlation matrix of the fit parameters:
      a      b      c
a      1.000
b      -0.997  1.000
c      0.874  -0.903  1.000

```

Figura 6: Ajuste con $n \log n$ del algoritmo DyV

4 Comparación y conclusiones.

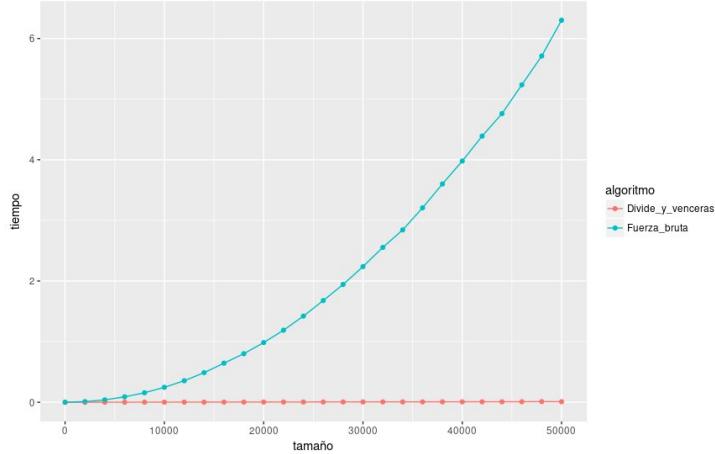


Figura 7: Comparación 2

Aunque desarrollar un algoritmo según el paradigma de divide y vencerás puede ser algo más complejo que utilizar una idea evidente como la propuesta con dos bucles anidados, los resultados obtenidos son mucho mejores utilizando técnicas de divide y vencerás y el orden de eficiencia pasa de ser cuadrático a casi lineal. Es importante realizar bien los análisis teóricos de eficiencia al plantear una solución divide y vencerás porque es fácil cometer errores obteniendo el orden del algoritmo debido a la recursividad o a las estructuras de datos necesarias para implementar el algoritmo. Aunque podría darse otro esquema distinto al de dos llamadas recursivas de n medios, es importante advertir que llamadas recursivas con problemas demasiado grandes (por ejemplo de tamaño $n-1$) no suelen dar buenos resultados. Del mismo modo, este problema ilustra bien la necesidad de realizar la combinación de los resultados obtenidos en las llamadas recursivas, y hay que señalar que se obtienen resultados adecuados porque el tiempo de combinación es lineal. Si la combinación fuera más lenta el algoritmo no funcionaría en los tiempos que queremos.