

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.	2
2.3. Eficiencia empírica.	3
2.4. Eficiencia híbrida.	3
3. Algoritmo Divide y Vencerás.	4
3.1. Algoritmo	4
3.2. Eficiencia teórica.	6
3.3. Eficiencia empírica.	6
3.4. Eficiencia híbrida.	6
4. Comparación	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.

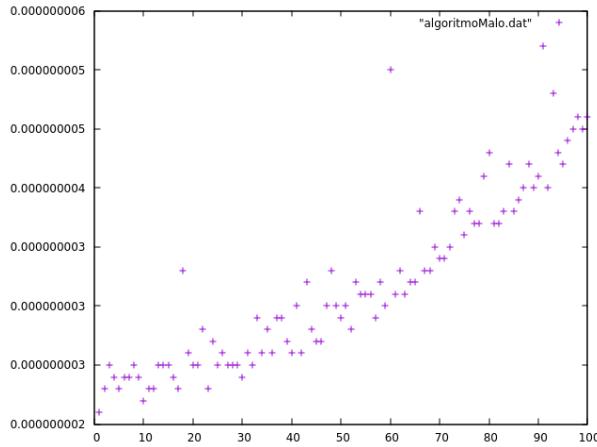


Figura 1: Algoritmo de fuerza bruta eficiencia empírica

2.4. Eficiencia híbrida.

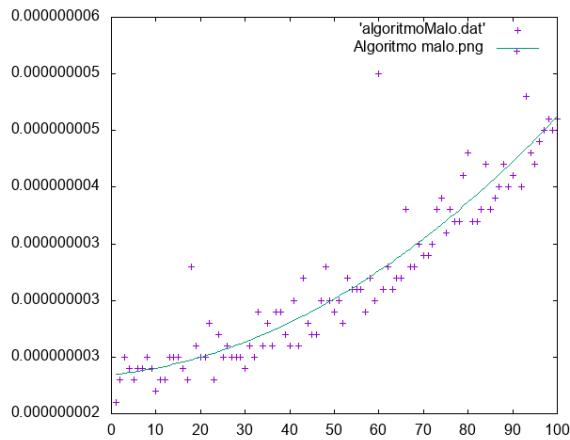


Figura 2: Algoritmo de fuerza bruta eficiencia híbrida

```

gnuplot> fit f(x) 'algoritmoMalo.dat' via a0,a1,a2
iter      chisq      delta/lim   lambda   a0          a1          a2
 0  2.1023635800e+09  0.00e+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 Posicion (vector<int> &vO, int ini, int fin, int &x) {
2     int resultado = 0;
3     if(ini <= fin) {
4         int m = ((fin - ini)/2)+ ini;
5         // cout << "x: " << x << " m: " << vO[m] << " " ;
6         if(x < vO[m]) resultado = Posicion(vO, ini, m, x);
7         else if (x>vO[m]) resultado = Posicion(vO, m+1, fin, x);
8         else resultado = m;
9     }
10    else resultado = -1;
11
12    // cout << "esto es el resultado: " << resultado << endl;
13    return resultado;
14
15 }
16
17 int Preferencias(vector<int> vO, vector<int> vD) {
18     int res = 0;
19
20     while(vD.size() > 0) {
21         int x = vD.front();
22
23         int pos = Posicion(vO, 0, vO.size()-1, x);
24         // cout << "pos " << pos ;
25         res = res + pos ;
26
27 //     cout << "\n";
28

```

```

29         vO.erase(find(vO.begin(),vO.end(),x));
30         vD.erase(vD.begin());
31     }
32
33 // cout << "resultado: " << res ;
34
35     return res;
36
37 }
38
39 vector<int> desordenado ;
40 vector<int> ordenado;
41
42
43
44 for(int i = 0 ; i < n ; i++) {
45     desordenado.push_back(T[i]) ;
46     ordenado.push_back(i);
47 }
48
49
50 double tiempo_transcurrido;
51 const int NUM_VECES=10000;
52 clock_t tantes=clock();
53 int inversiones;
54 for (int i=0; i<NUM_VECES;i++)
55     inversiones = Preferencias(ordenado,desordenado) ;
56
57 clock_t tdespues = clock();
58
59 tiempo_transcurrido=((double)(tdespues-tantes) / (CLOCKS_PER_SEC* (double)
      NUM_VECES));
60
61 //cout << "\nInversiones = " << inversiones ;
62 cout <<           << n    <<           << tiempo_transcurrido << endl;
63
64
65
66 }

```

La solución que hemos planteado se basa en suponer que, dado el elemento más a la izquierda del vector no ordenado, el número de inversiones que tienen que ver con este será igual al número de elementos que hay a la izquierda del elemento en el vector ordenado (que es igual a la posición que ocupa en este). Ahora bien, si eliminamos de ambos vectores(ordenado y desordenado) el número que estaba más a la izquierda en el desordenado, este razonamiento se puede repetir hasta que no queden números en el vector.

En un principio este algoritmo nos pareció acertado y decidimos implementarlo. Sin embargo, durante el proceso pudimos comprobar que la idea tenía un fallo teórico en la base y al medir la eficiencia empírica vimos que no se correspondía con los resultados esperados.

Como no hemos tenido tiempo de plantear otro algoritmo y repetir el proceso, hemos decidido analizar el

error y comentarlo.

3.2. Eficiencia teórica.

El algoritmo se basa en la función preferencias, que tiene n iteraciones y en cada iteración llama a la función posición (que implementa una búsqueda binaria) que tiene una eficiencia $O(\log(n))$ (aquí también se aplica el divide y vencerás). El problema del algoritmo es que necesita ir eliminando los elementos usados de los dos vectores que emplea y la eliminación en la estructura de datos *vector* es $O(n)$ en lugar de $O(1)$ (que era lo que supusimos en un principio). También planteamos algunas alternativas para solucionar esto, como utilizar listas enlazadas (en las que la eliminación es $O(1)$), pero entonces no se podía implementar la búsqueda binaria y el tiempo de búsqueda pasaba a ser lineal(aún así la cosa mejoraría porque tendríamos una llamada $O(n)$ y una $O(1)$ en lugar de una $O(\log(n))$ y otra $O(n)$) como es el caso, pero dado que la mejora no suponía superar al algoritmo que llamamos de fuerza bruta y que no hemos dispuesto de tiempo suficiente para repetir todo el proceso de desarrollo, hemos decidido dejarlo así. La mejor opción que se nos ocurrió fue implementar el algoritmo desde los dos extremos del vector (ya que la propiedad descrita es simétrica) y, haciendo la implementación con listas enlazadas, mantener un puntero a la mitad de la lista para hacer que la búsqueda fuera $\frac{n}{2}$. Así se mejoraría un poco el rendimiento pues se eliminarían los vectores de dos en dos y si estuvieran en la misma mitad del vector no habría que recorrerlo entero. Sin embargo, en el peor caso seguía siendo una búsqueda en $O(n)$ y el algoritmo seguía siendo igual o peor que el de fuerza bruta.

3.3. Eficiencia empírica.

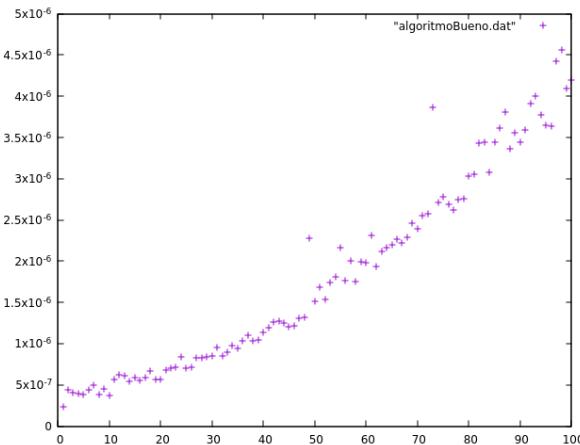


Figura 4: Algoritmo divide y vencerás sin ajuste

Al medir los tiempos de forma empírica ya pudimos intuir que algo no funcionaba como esperábamos. La curva de ejecución se adivina cuadrática.

3.4. Eficiencia híbrida.

Como es lógico, al intentar ajustar los tiempos a una función $n \log n$ los resultados obtenidos son evidentemente malos.

Como se puede apreciar en la figura 6, el error es bastante grande (127 % en el parámetro b), por eso probamos a ajustarla con una función cuadrática y los resultados fueron más próximos, como era de esperar, pues el algoritmo tenía un mal fundamento teórico.

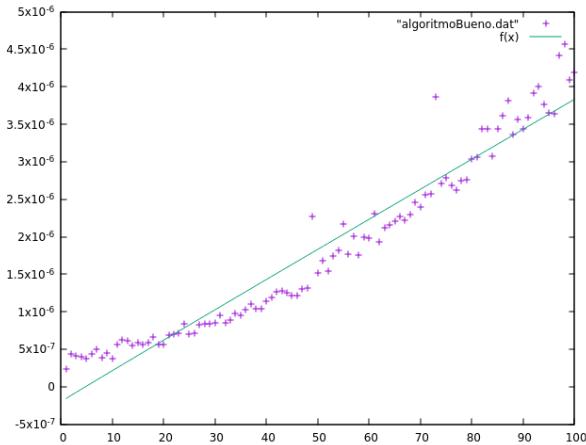


Figura 5: Algoritmo divide y vencerás con ajuste $\log(n)$

```

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 chi-square) = 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
gnuplot>

```

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

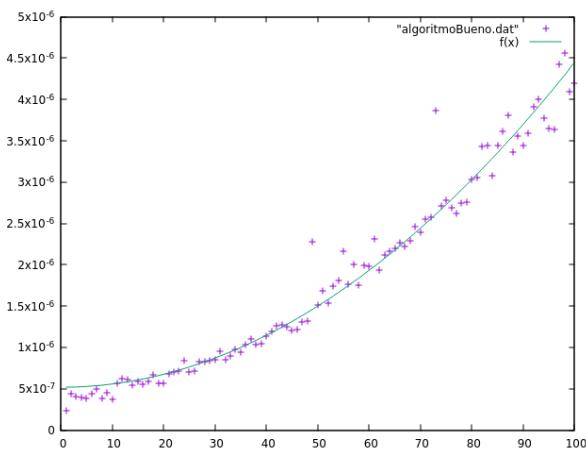


Figura 7: Algoritmo divide y vencerás con ajuste cuadrático

```

gnuplot> fit f(x) 'algoritmoBueno.dat' via a,b,c
iter      chisq      delta/lim  lambda   a          b          c
  0  1.6516762167e+05  0.00e+00  2.35e+01  2.111845e-08  6.986801e-01  1.082197e-07
  1  1.8230221160e+00 -9.06e+09  2.35e+00  2.111840e-08  2.319646e-03  1.082197e-07
  2  5.7943101574e-08 -3.15e+12  2.35e-01  2.111840e-08 -1.478538e-06  1.082197e-07
  3  5.5917540498e-08 -3.62e+03  2.35e-02  2.111838e-08 -1.555907e-06  1.082197e-07
  4  5.5905935366e-08 -2.08e+01  2.35e-03  2.111621e-08 -1.555744e-06  1.082197e-07
  5  5.4763302843e-08 -2.09e+03  2.35e-04  2.090164e-08 -1.539571e-06  1.082210e-07
  6  1.3189127582e-08 -3.15e+05  2.35e-05  1.037275e-08 -7.459770e-07  1.083134e-07
  7  5.6474979859e-12 -2.33e+08  2.35e-06  3.255908e-10  1.130688e-08  1.084512e-07
  8  4.4234927498e-12 -2.77e+04  2.35e-07  2.304392e-10  1.840467e-08  1.134149e-07
  9  3.9611702030e-12 -1.17e+04  2.35e-08  2.805116e-10  1.236552e-08  2.651722e-07
 10 3.9134829603e-12 -1.22e+03  2.35e-09  3.024746e-10  9.716786e-09  3.317243e-07
 11 3.9134820431e-12 -2.34e-02  2.35e-10  3.025713e-10  9.705119e-09  3.320175e-07
iter      chisq      delta/lim  lambda   a          b          c

After 11 iterations the fit converged.
final sum of squares of residuals : 3.91348e-12
rel. change during last iteration : -2.34354e-07

degrees of freedom (FIT_NDF) : 97
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 2.00861e-07
variance of residuals (reduced chisquare) = WSSR/ndf : 4.03452e-14

Final set of parameters      Asymptotic Standard Error
=====
a      = 3.02571e-10    +/- 2.696e-11  (8.90%)
b      = 9.70512e-09    +/- 2.81e-09  (28.95%)
c      = 3.32017e-07    +/- 6.148e-08  (18.52%)

correlation matrix of the fit parameters:
      a      b      c
a      1.000
b      -0.969  1.000
c      0.753 -0.871  1.000

```

Figura 8: Ajuste con n^2 del algoritmo DyV

En este caso los tiempos medidos sí que se ajustan a la función, dando evidencia de que el algoritmo genera tiempos igual o peores que los del algoritmo base que planteamos como "mala solución". Como se puede apreciar en las fotos de los ajustes, los índices de error ahora son mucho menores por lo que concluimos que la función tiene un tiempo cuadrático. Basta con deducir que si el tiempo de la eliminación es los vectores es lineal y el de la búsqueda logarítmico, por la regla de la suma prevalece la parte lineal de la eliminación y el algoritmo en cada iteración es lineal. Como se ejecuta en todos los elementos, tenemos $n \times n$ y aparece esa eficiencia cuadrática.

4 Comparación y conclusiones.

La comparación es obvia, la idea que hemos propuesto es incluso peor que la inicial (no las hemos comparado gráficamente porque la diferencia es irrelevante, la conclusión es que la solución propuesta es totalmente desecharable).

De esta practica hemos aprendido que hay que darle más importancia al análisis teórico de los algoritmos antes de molestarse en implementarlos y mucho menos en realizar mediciones, solucionar errores y demás.