Algorítmica

Práctica 2

Comparación de Preferencias

Fc.Javier Azpeitia Muñoz
Christian Andrades Molina
Guillermo Muriel Sánchez lafuente
Miguel Keane Cañizares
Mercedes Alba Moyano

GRUPO 1

- Índice:

1. Introducción.

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 numero 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,…,n, y que uno de los rankings siempre es 1,...,n (si no fuese as bastara reenumerarlos) y el otro es a1,a2, …, an, de forma que dos productos i y j están invertidos si i < j pero ai > aj . De esta forma nuestra representación del problema será un vector de enteros v de tamaño n, de forma que v[i] = ai; i = 1,...,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.

2. Definición del problema

Una página web compara las preferencias de un usuario con los otros mediante los rankings establecidos por ellos de “n” productos. La medida de similitud entre dos usuarios es el número de inversiones entre ellos:

Ranking Usuario A : 1, 2, … , n.
Ranking Usuario B: a1, a2, … , an.

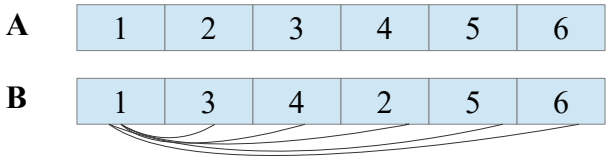
Se producen inversiones si i < j pero ai > aj

3.- Formas de resolver el problema

Vamos a resolver este problema mediante dos algoritmos: **Fuerza bruta** y **Divide y vencerás**.

3.1.- Fuerza bruta O(n²)

Para el algoritmo de fuerza bruta hemos contado directamente las “inversiones” que hay dentro de un mismo ranking. Este “ranking”, es un array de caracteres creado aleatoriamente gracias al archivo generador suffle.cpp.



Comparamos todos los pares (i, j). En este caso tenemos 2 inversiones: 3-2 y 4-2.

Código “suffle.cpp”

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

//generador de ejemplos para el problema de la comparación de preferencias. Simplemente se genera una permutación aleatoria del vector 0,1,2,...,n-2,n-1

double uniforme() //Genera un número uniformemente distribuido en el
                  //intervalo [0,1) a partir de uno de los generadores
                  //disponibles en C.
{
 int t = rand();
 double f = ((double)RAND_MAX+1.0);
 return (double)t/f;
}

int main(int argc, char * argv[])
{

  if (argc != 2)
    {
     cerr << "Formato " << argv[0] << " <num_elem>" << endl;
     return -1;
    }

  int n = atoi(argv[1]);
```

```
  int * T = new int[n];
  assert(T);

srand(time(0));

for (int j=0; j<n; j++) T[j]=j;
//for (int j=0; j<n; j++) {cout << T[j] << " ";}
//algoritmo de random shuffling the Knuth (permutación aleatoria)
for (int j=n-1; j>0; j--) {
   double u=uniforme();
   int k=(int)(j*u);
   int tmp=T[j];
   T[j]=T[k];
   T[k]=tmp;
}

for (int j=0; j<n; j++) {cout << T[j] << " ";}
cout << endl;

}
```

Una vez creado aleatoriamente el ranking por suffle.cpp , se lo introducimos a fuerza_bruta.cpp y éste, se dispone a contar las “inversiones” que el ranking tiene. (Recordar que una inversión es aquella vez en la que un número es mayor que el otro que le sigue después en un ranking).

Ahora fuerza_bruta.cpp mediante la función “atoi”, transforma esos caracteres procedentes del código aleatorio en enteros, para que así sea más fácil la comparación.

Esta comparación, no es más que dos bucles “for” anidados, en los que cada elemento de la cadena se compara con todos los elementos siguientes y, por cada vez que ese elemento esté “invertido” se actualiza un contador entero llamado “inversión”, que mostraremos al final como el número total de inversiones.

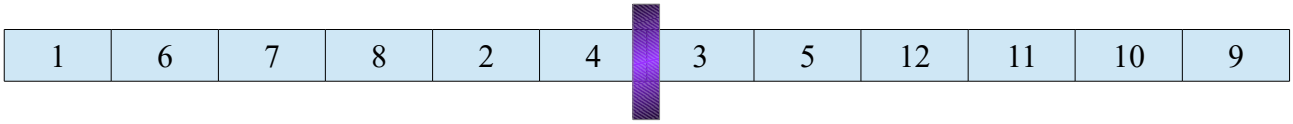
Código “fuerza_bruta.cpp”

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

int main(int argc, char * argv[]){

int inversion = 0;
int num;

cout << "argc :" <<argc;
```

```
        for(int i=1; i<argc; i++){
                num = atoi(argv[i]);
                for(int j=i; j<argc; j++){
                        if(num>atoi(argv[j])){
                                inversion++;
                        }
                }

        }


cout << "\nLa inversión es: "<< inversion << endl;
cout<< endl;

}
```

3.2. Divide y vencerás O(n*log(n))

En esta forma, vamos a dividir la lista o ranking en dos mitades. Después, contaremos recursivamente el número de inversiones de cada una.



-Mitad A

1	6	7	8	2	4

Inversiones: 6-2, 6-4, 7-2, 8-2, 8-4

-Mitad B

3	5	12	11	10	9

Inversiones: 12-11, 12-10, 12-9, 11-10, 11-9, 10-9

Una vez dividido, **combinamos**, es decir, contamos las inversiones en las que ai y aj están en diferentes mitades partiendo de que estén ordenadas. Mezclamos y devolvemos el conjunto ordenado.

1	6	7	8	2	4

Inversiones: 6 + 6 +7 = 19

3	5	12	11	10	9
4	**3**	**0**	**0**	**0**	**0**

Con esta procedimiento, finalmente obtenemos un algoritmo de orden (n*log(n)).

4.- Implementación en Pseudocódigo

El pseudocódigo en el que nos basaremos para realizar la forma divide y vencerás es el siguiente:

Pseudocódigo

```
Sort-and-Count (L)
{
  if (L.length==1) //Comprobamos si estamos en un caso base o no
     return (0,L);

// Dividir la lista en dos vectores A y B (mitades) y contar inversiones
        (rA, A) ← Sort-and-Count(A)
        (rB, B) ← Sort-and-Count(B)
// Unir mitades ordenadas y contar las inversiones en diferentes mitades
        (r, L) ← Merge-and-Count(A, B)
return ( rA+ rB + r, L ); //Devolvemos la suma de las inversiones de A, B y ambas.
```

Código dyvsortcount.cpp (Equivalente a Sort-and-Count)

```
#include <iostream>
#include <vector>
#include <utility>

using namespace std;

pair<int,vector<int> > ord_cont(vector v){
        pair<int,vector<int> > p1;
        pair<int,vector<int> > p2;
        pair<int,vector<int> > p3;
        pair<int,vector<int> > finalp;

        //Comprobamos si estamos en un caso base o no
        if (v.size()==1){{
                pair <int, vector<int> > mip;
                mip.first=0;
                mip.second=v;
                return mip;
        }

        //Division del vector en 2 vectores
```

```
        vector<string> a;
        vector<string> b;
        for (int i = 0; i < v.size()/2; i++){  //size() retorna valores enteros sin decimales,
eliminando la parte decimal
                p1.second.push_back(v[i]);

        for (int i =v.size()/2 ;i < v.size(); i++)
                p2.second.push_back(v[i]);

        //fin de la division, los vectores resultantes son a y b
        //llamadas recursivas
        p1=ord_cont(a);
        p2=ord_cont(b);
        p3=contarOrdena(a,b);


        finalp.first=p1.first+p2.first+p3.first;
        finalp.second=v;

        return (finalp);

}
```

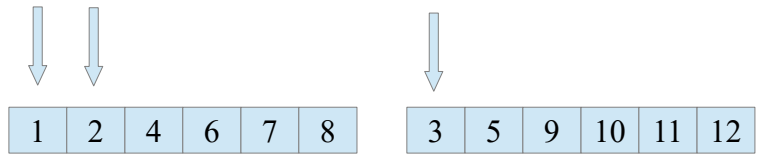
Código contarOrdena.cpp (Equivalente a Merge-and-Count)

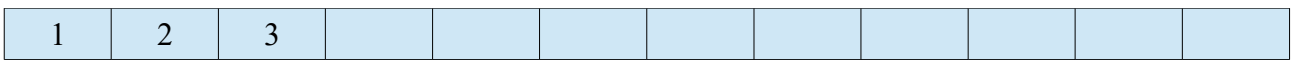
```
#include <iostream>
#include <utility>
#include <vector>

using namespace std;

pair<int, vector<int> > contarOrdena(vector<int> u, vector<int> v){//recibe las dos mitades
ordenadas
        int cont=0,i=0,j=0,min;
        vector<int> z;
        pair<int, vector<int> > sol; //Par solucion

        while(i<u.size() && j<v.size()){ //bucle, hasta que uno de los dos vectores se acabe.
                if(u[i]<v[j]){          //Si el elemento i-esimo de u es menor que el j-esimo
                        z.push_back(u[i]);  //Mete el elemento de u en el vector auxiliar
                        i++;                //incrementa el contador del vector u
                }
                else{
                        z.push_back(v[j]);  //Si no, mete el elemento de v e incrementa v
```

```
                    j++;
                    cont+=u.size()-i;   //Aumenta el contador de intercambios lo equivalente a el tamaño de u menos las posiciones que ya ha sido avanzado.
                }
        }
   //Completamos nuestro vector auxiliar con el resto de elementos que nos queden en alguno de los dos vectores.
        if(i==u.size()){
                while(j<v.size()){
                        z.push_back(v[j]);
                        j++;
                }
        }
        else{
                while(i<u.size()){
                        z.push_back(u[i]);
                        i++;
                }
        }
        sol.first=cont;
        sol.second=z;

        return sol; }
```

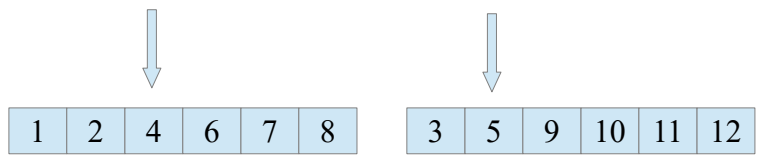
5.- Funcionamiento Merge and Count

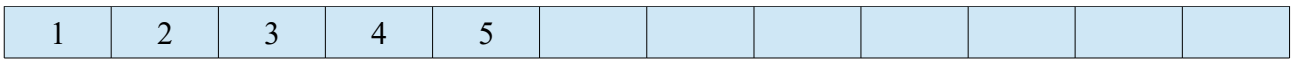
Paso 1



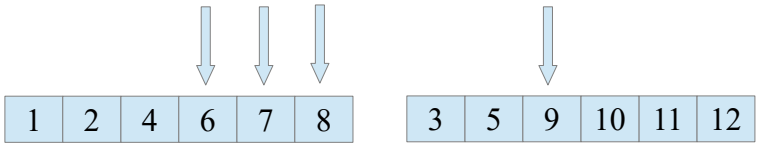


Paso 2

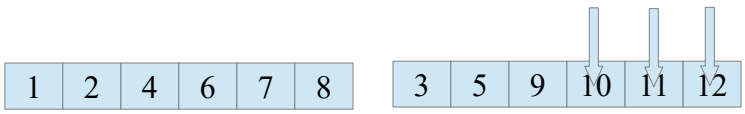




Paso 3



Paso 4, 5 y 6



6.- Comparativa de Fuerza bruta y Divide y vencerás

Una vez implementada la librería “chrono” en los dos algoritmos, tomamos los tiempos usando una macro.sh.

Macro.sh (Para fuerza bruta)

```
#!/bin/csh -vx
echo "" >> salida.dat
@ argc = 800
while ( $argc <= 24000 )
./ejecutable $argc >> salida.dat
@ argc += 800
end
```

Como vemos en la macro comenzamos con 800, y vamos sumando 800 hasta llegar a 24000.

Macro.sh (Para Divide y vencerás)

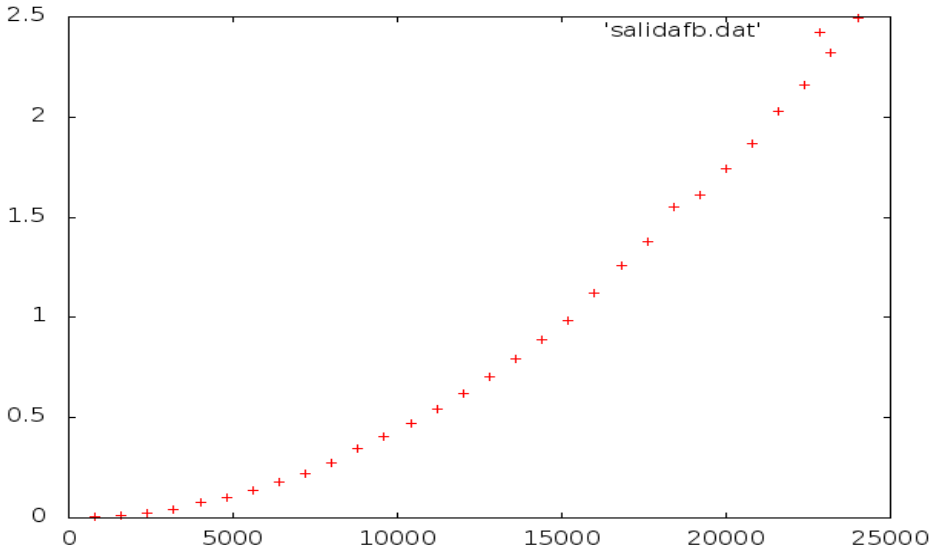
```
#!/bin/csh -vx
echo "" >> salida.dat
@ argc = 90000
while ( $argc <= 2700000)
./ejecutable $argc >> salida.dat
@ argc += 90000
end
```

En este caso comenzamos a partir de 90000, y le vamos sumando ese mismo valor cada iteración hasta llegar a 2700000.

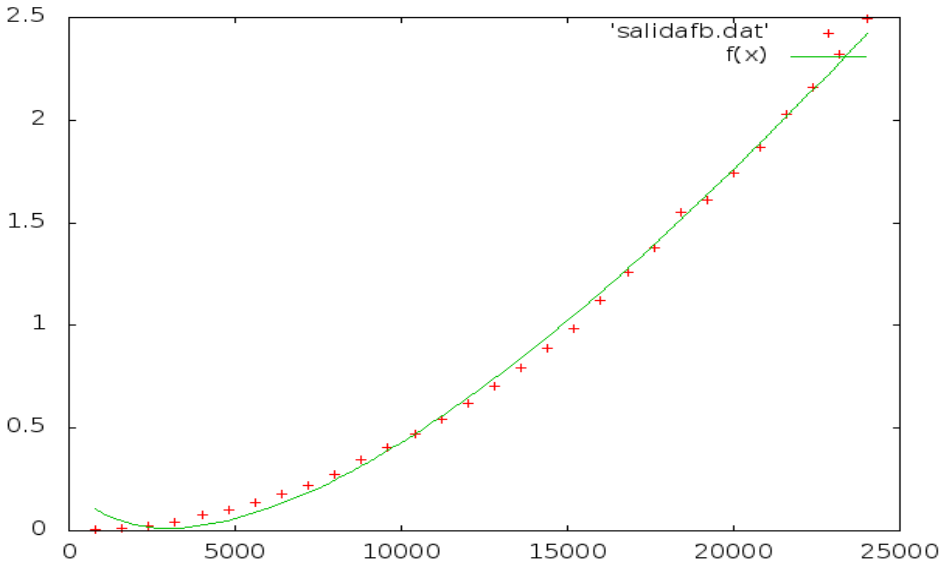
Tiempos Fuerza bruta

Tamaño	Tiempo(seg)	Tamaño	Tiempo(seg)
800	0,004285	**12800**	0,706803
1600	0,01468	**13600**	0,793469
2400	0,025523	**14400**	0,8877
3200	0,044742	**15200**	0,986821
4000	0,077253	**16000**	1,12243
4800	0,100034	**16800**	1,25774
5600	0,134947	**17600**	1,37679
6400	0,176217	**18400**	1,55339
7200	0,223684	**19200**	1,61142
8000	0,276205	**20000**	1,74504
8800	0,344716	**20800**	1,8655
9600	0,404738	**21600**	2,,03142
10400	0,47213	**22400**	2,15824
11200	0,543367	**23200**	2,32161
12000	0,620896	**24000**	2,49592

Gráfica Fuerza bruta sin ajuste



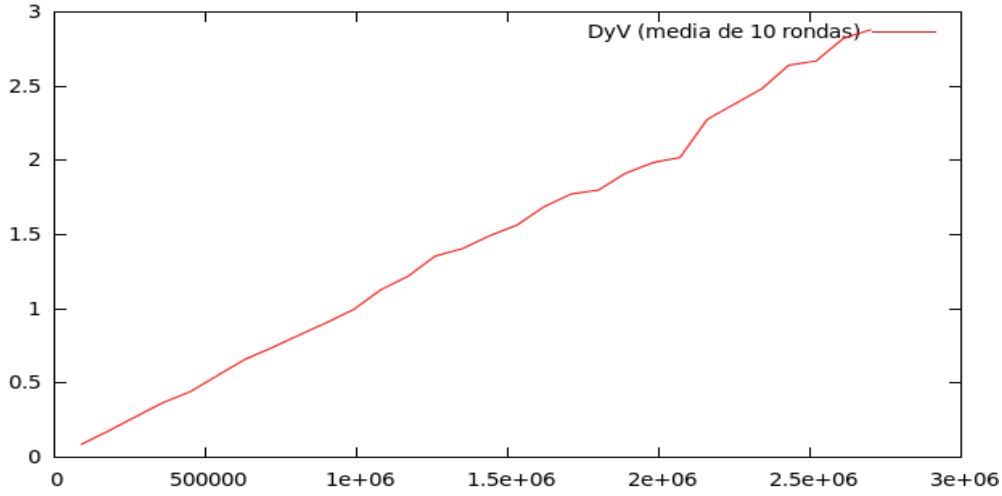
Gráfica Fuerza bruta con ajuste



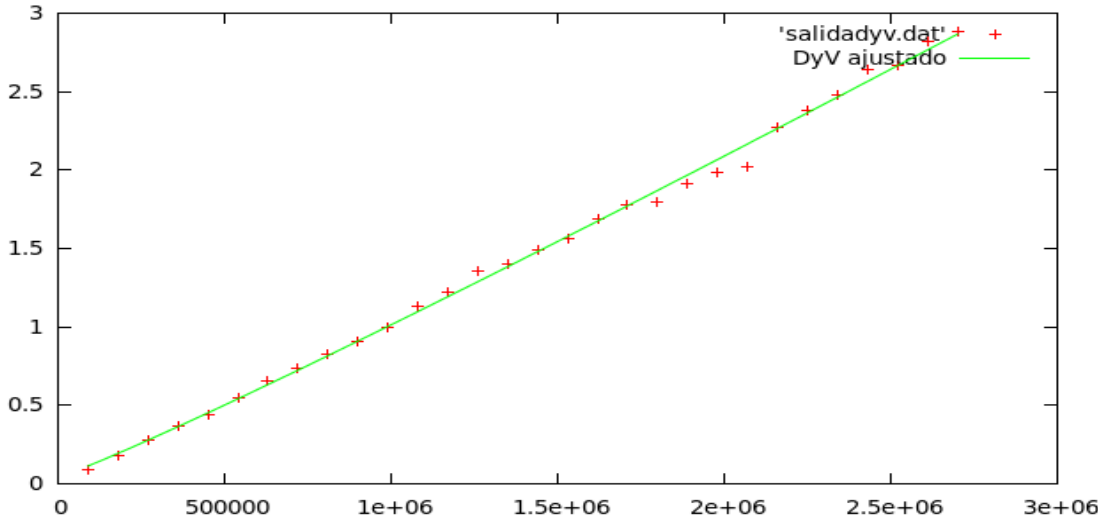
Tiempos Divide y Vencerás

Tamaño	Tiempo(seg)	Tamaño	Tiempo(seg)
90000	0,088456	**1440000**	1,49315
180000	0,179525	**1530000**	1,56418
270000	0,275039	**1620000**	1,68835
360000	0,369337	**1710000**	1,7746
450000	0,443038	**1800000**	1,80031
540000	0,551783	**1890000**	1,91426
630000	0,66006	**1980000**	1,98497
720000	0,73925	**2070000**	2,02055
810000	0,824464	**2160000**	2,27641
900000	0,907129	**2250000**	2,37947
990000	0,996319	**2340000**	2,48176
1080000	1,12873	**2430000**	2,66979
1170000	1,21991	**2520000**	2,66979
1260000	1,35588	**2610000**	2,82171
1350000	1,40592	**2700000**	2,87998

Gráfica Divide y vencerás sin ajuste



Gráfica Divide y vencerás con ajuste



7.- Conclusión

Como vemos en el gráfico, o si queremos, en los últimos valores de las tablas de tiempos, llegamos a la conclusión de que el algoritmo “Divide y vencerás” es mucho más eficiente que el algoritmo de “Fuerza bruta”

Gráfico comparativo de los dos algoritmos