Memoria de divide y vencerás

Rubén Morales Pérez Francisco Javier Morales Piqueras
Bruno Santindrian Manzanedo Ignacio de Loyola Barragan Lozano
Francisco Leopoldo Gallego Salido

12 de abril de 2016

Índice

1. Introducción

Divide y vencerás es una técnica algorítmica que consiste en resolver un problema diviéndolo en problemas más pequeños y combinando las soluciones. El proceso de división continúa hasta que los subproblemas llegan a ser lo suficientemente sencillos como para una resolución directa.
El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales. El coste computacional se determina resolviendo relaciones de recurrencia.

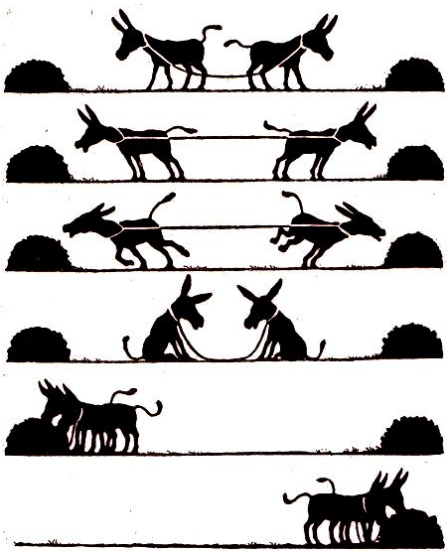

Figura 1: Subproblemas

Una forma de dirigir las cartas siguiendo este método (creada por Donald Knuth, autor de The Art of Computer Programming) es separarlas en bolsas diferentes en función de su área geográfica y cada una a su vez se ordena en lotes para subregiones más pequeñas.
Un ejemplo anterior a Cristo donde se usa ¨divide y vencerás¨ con un único subproblema es el algoritmo de Euclides para computar el máximo común divisor de dos números.

$$
\begin{aligned}
11312 &= 500 \cdot 22 + 312, \ldots [1] \\
500 &= 312 \cdot 1 + 188, \ldots\ldots [2] \\
312 &= 188 \cdot 1 + 124, \ldots\ldots [3] \\
188 &= 124 \cdot 1 + 64, \ldots\ldots\ldots [4] \\
124 &= 64 \cdot 1 + 60, \ldots\ldots\ldots [5] \\
64 &= 60 \cdot 1 + 4 \ldots\ldots\ldots\ldots [6]
\end{aligned}
$$

Figura 2: Euclides

2. Mezclando k vectores ordenados

Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos). Una posible alternativa consiste en, utilizando un algoritmo clásico, mezclar los dos primeros vectores, posteriormente mezclar el resultado con el tercero, y así sucesivamente.

- ¿Cuál sería el tiempo de ejecución de este algoritmo?
- Diseñe, analice la eficiencia e implemente un algoritmo de mezcla más eficiente, basado en d̈ivide y vencerás.
- Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

2.1. Estudio preliminar

Planteándo el problema es posible imponer una cota superior teórica a la mezcla. Teniendo en cuenta que hay kn elementos, si aplicásemos un algoritmo con eficiencia $O(n) = nlog(n)$ deducimos que podemos encontrar un algoritmo de ordenación básica con eficiencia $O(k, n) = nklog(nk)$. En este caso estaríamos representando los k vectores de n elementos como un único vector, sin aprovechar aún el hecho de que parte del ¨ vector ëstá ordenado.

2.2. Código

2.2.1. Divide y vencerás

```
// Funcion que llama a todo el algoritmo
int* MezclaDyV(int** vectors, int n_vec, int n_elem){
  int * full_vector = GeneraVector(vectors, n_vec, n_elem); // Generamos el
     vector a devolver
  MergeKPartitions(full_vector, n_elem, 0, n_vec*n_elem-1); // Aplicamos la
     particion y mezcla

  return full_vector;
}
```

```
// Funcion que genera un vector a partir de una matriz, colocando cada
   vector
// seguido de otro.
int * GeneraVector(int** &matriz, int n_vec, int n_elem){
  int* vector = new int[n_vec*n_elem];

  for (int i=0; i<n_vec; ++i){
    for (int j=0; j<n_elem; ++j){
      vector[i*n_elem+j] = matriz[i][j];
    }
  }

  return vector;
}
```

```
// Funcion que hace la particion del vector, teniendo en cuenta donde empieza,
// donde termina y el numero de elementos de la particion, y devuelve ese trozo
// ya ordenado
void MergeKPartitions(int* &vector, int n_elem, int ini, int fin){
  int size = fin - ini + 1;
  int partitions = size/n_elem;
  // Caso base: Si hay 2 particiones de igual tamanio, las mezcla
  if(partitions == 2){
    Merge(vector, ini, ini+n_elem, fin);
  }
  // Si hay mas de dos particiones:
  else if(partitions > 2){
    int division = ini + (partitions/2)*n_elem; // Calculo de la division
    // Aqui es basicamente donde aplicamos divide y venceras, el cual nos acaba
    // proporcionando la eficiencia deseada para esta situacion.
    MergeKPartitions(vector, n_elem, ini, division-1);  // Llamada al primer conjunto de particiones
    MergeKPartitions(vector, n_elem, division, fin);    // Llamada al segundo conjunto de particiones
    Merge(vector, ini, division, fin);  // Mezclamos los dos conjuntos de particiones, ya ordenados
  }
}
```

```
// Funcion que mezcla dos vectores ordenados en uno ordenado
void Merge(int* &vector, int ini, int pivot, int fin){
  int size = fin-ini+1;
  int f_count = 0, s_count = 0;// Contadores de cada vector
  int selected;
  int* aux = new int[size];

  // Hasta que agotemos el tamanio del vector auxiliar
  for (int i=0; i<size; ++i){
    // Si hemos agotado el primer vector:
    if(ini+f_count == pivot){
      selected = vector[pivot+s_count];
      s_count++;
    }
    // Si hemos agotado el segundo:
    else if(s_count+pivot == fin+1){
      selected = vector[ini+f_count];
      f_count++;
    }
    // En cualquier otro caso:
    else{
      // Si el del primer vector es mayor que el del segundo:
      if(vector[ini+f_count] <= vector[pivot+s_count]){
        selected = vector[ini+f_count];
        f_count++;
```

```
      }
      // Si es al reves:
      else{
        selected = vector[pivot+s_count];
        s_count++;
      }
    }
    // Asignamos valor a la posicion correspondiente del vector
    aux[i] = selected;
  }
  // Asignamos el vector auxiliar al trozo del vector real
  for (int i=0; i<size; ++i){
    vector[ini+i] = aux[i];
  }
  delete[] aux;
}
```

2.2.2. Fuerza bruta

```
int* MezclaNoDyV(int** M, int num_vec, int num_elem){
  int* vec_indices = new int[num_vec]; //Vector que almacena los indices
  for (int i=0; i<num_vec; ++i)
    vec_indices[i]=0;

  int pos_escritas = 0; //Numero de posiciones escritas
  const int MAX_POS_ESCRITAS = num_vec*num_elem; //Tope de las posiciones
     que se pueden escribir (n*k)
  int* vec_ordenado = new int[MAX_POS_ESCRITAS]; //Variable donde se va
     almacenando el vector resultante ordenado.

  while (pos_escritas < MAX_POS_ESCRITAS){
    int pos_min = f_pos_min(vec_indices, M, num_vec);
    vec_ordenado[pos_escritas] = M[pos_min][vec_indices[pos_min]];
    if (vec_indices[pos_min] >= num_elem-1)
      vec_indices[pos_min] = -1;
    else
      ++vec_indices[pos_min];
    ++pos_escritas;
  }

  delete[] vec_indices;
  return vec_ordenado;
}
```

```

//Funcion que calcula el minimo de entre los elementos apuntados por los
   indices

int f_pos_min(int* vec_indices, int** M, int num_vec){
  int pos_min = 0;
  while((vec_indices[pos_min] <0) && (pos_min < num_vec))
```

```
7       ++pos_min;
8     for (int i=pos_min+1; i<num_vec; ++i){
9       if (vec_indices[i] >= 0){
10        if (M[i][vec_indices[i]] < M[pos_min][vec_indices[pos_min]])
11          pos_min = i;
12      }
13    }
14    return pos_min;
15  }
```

2.3. Tiempo de ejecución

2.4. Estudio empírico e híbrido

2.4.1. Mezcla con furza bruta

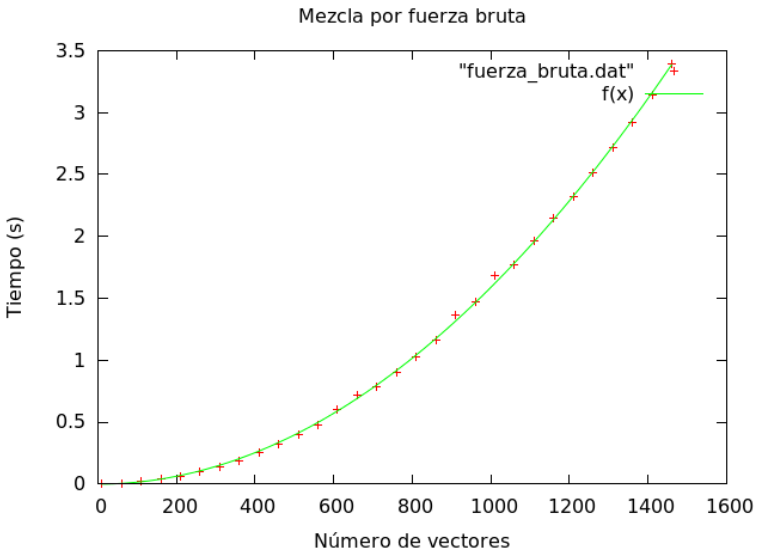

Figura 3: Fuerza bruta con 100 elementos cada vector

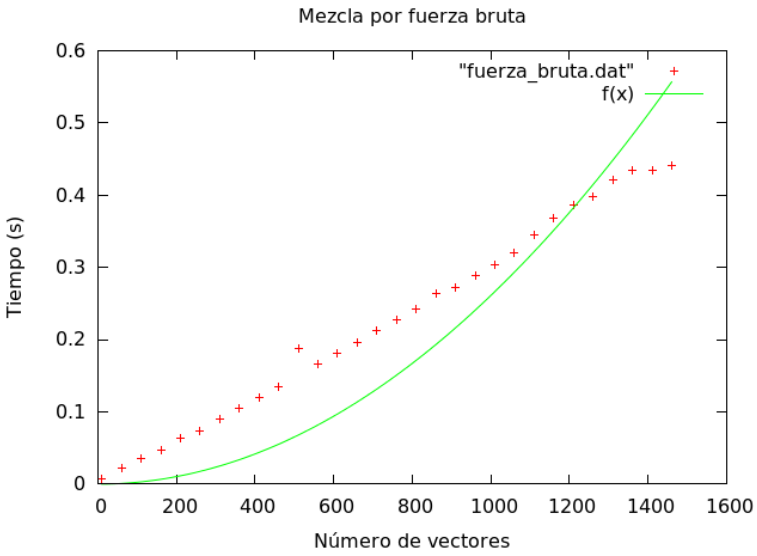

Figura 4: Fuerza bruta con 100 vectores

2.4.2. Mezcla con divide y vencerás

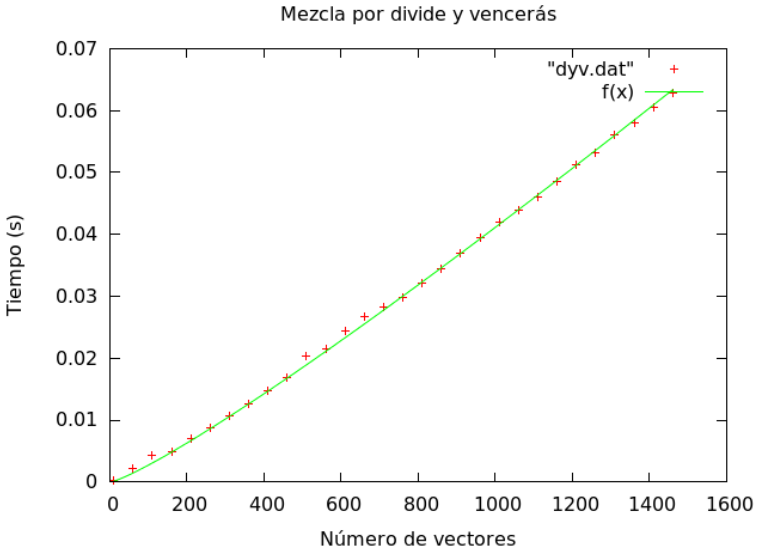

Figura 5: Divide y vencerás con 100 elementos cada vector

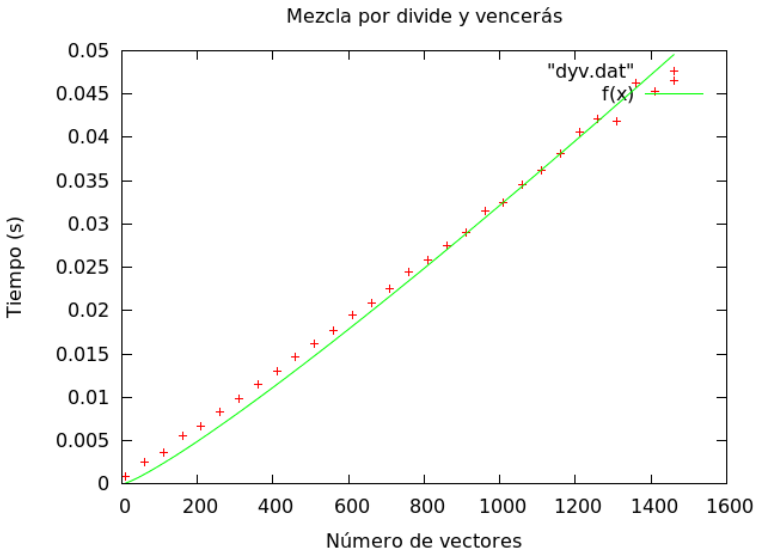

Figura 6: Divide y vencerás con 100 vectores

2.4.3. Comparativa

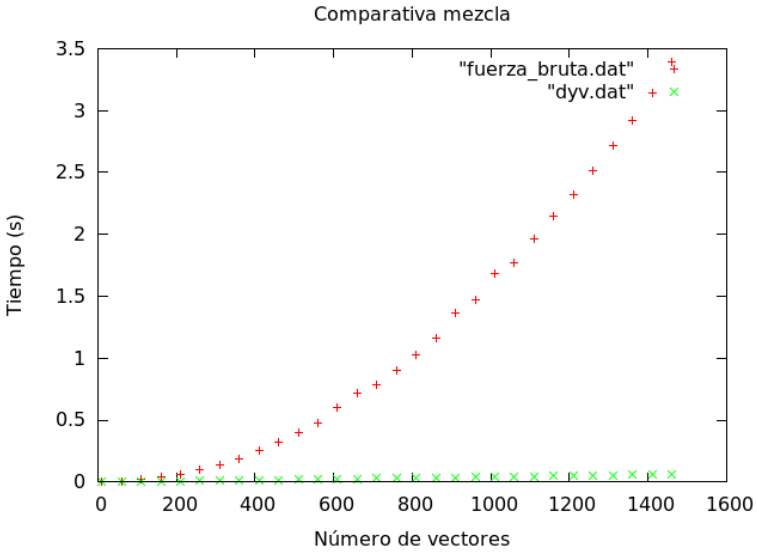

Figura 7: Comparativa con 100 elementos cada vector

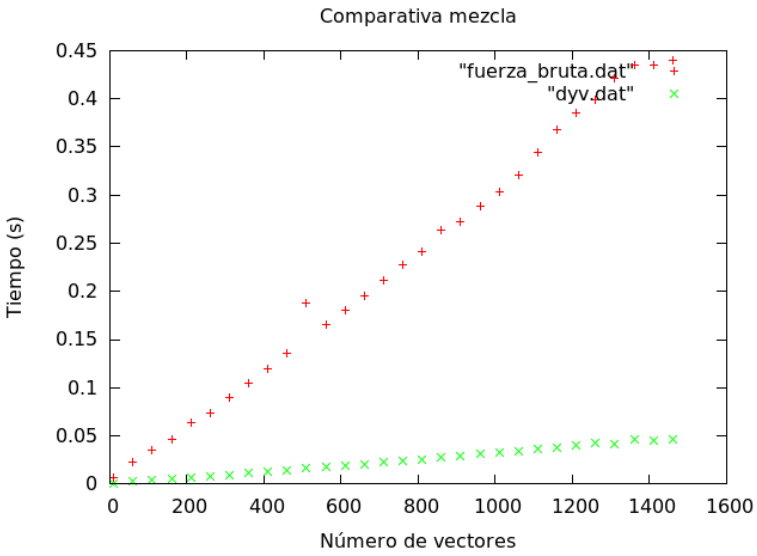

Figura 8: Comparativa con 100 vectores

3. Comparación de preferencias (opcional)

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 ïnvertidosën 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 $1, \cdots, n$, y que uno de los rankings siempre es $1, \cdots, n$ (si no fuese así bastaría reenumerarlos) y el otro es $a_1, a_2, \cdots, a_n$, 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 $v[i] = a_i \ / \ i = 1, \cdots, n$.
El objetivo es diseñar, analizar la eficiencia e implementar un algoritmo d̈ivide y vencerásp̈ara medir la similitud entre dos rankings. Compararlo con el algoritmo de f̈uerza brutaöbvio. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

3.1. Automatizando el problema

A la hora de generar los ejecutables, los datos y las gráficas hemos usado los siguientes scripts:

script.sh

```
1  #!/bin/bash
2
3  if [ $# -ne 1 ]
4  then
```

```
    echo "Uso: $0 <nombre>"
    exit 1
fi

mkdir ../Graficas 2> /dev/null
mkdir ../Datos 2> /dev/null

# Fuerza bruta
g++ -std=c++11 -D GP_OUT ../src/fuerza_bruta.cpp
nelementos=10
echo "" > datos.dat
while [ $nelementos -lt 5000 ]; do
    ./a.out $nelementos >> datos.dat
    let nelementos=nelementos+10
done
gnuplot ./gnuplot/fuerza_bruta.gp

mv grafica.png ../Graficas/fuerza_bruta_$1.png
mv datos.dat ../Datos/fuerza_bruta_$1.dat
mv fit.log ../Datos/fit_fuerza_bruta_$1.log
echo "Fuerza bruta completado"

# DyV
g++ -std=c++11 -D GP_OUT ../src/dyv.cpp
nelementos=10
echo "" > datos.dat
while [ $nelementos -lt 5000 ]; do
    ./a.out $nelementos >> datos.dat
    let nelementos=nelementos+10
done
gnuplot ./gnuplot/dyv.gp

mv grafica.png ../Graficas/dyv_$1.png
mv datos.dat ../Datos/dyv_$1.dat
mv fit.log ../Datos/fit_dyv_$1.log
echo "DyV completado"
```

Para conseguir los datos y gráficas hemos usado:

$$fuerza_bruta.gp$$

```
set terminal pngcairo
set output "grafica.png"

set title "Fuerza bruta"
set xlabel "Tamanio del vector"
set ylabel "Tiempo (s)"
set fit quiet
f(x) = a*x*x + b*x + c
fit f(x) "datos.dat" via a, b, c
plot "datos.dat", f(x)
```

dyv.gp

```
set terminal pngcairo
set output "grafica.png"

set title "Divide y vencer\'as"
set xlabel "Tamanio del vector"
set ylabel "Tiempo (s)"
set fit quiet
f(x) = a*x*x + b*x + c
fit f(x) "datos.dat" via a, b, c
plot "datos.dat", f(x)
```

3.2. Fuerza bruta

El programa usado para calcular los tiempos del algoritmo de fuerza bruta es:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <chrono>
using namespace std;


int inversiones(vector<int> v){
  int count = 0;
  int size = v.size();
  for (int i=0; i < size; i++)
    for (int j=i+1; j < size; j++)
      if (v[i] > v[j])
        count++;

  return count;
}


int main(int argc, char** argv){
  if (argc < 2){
    cerr << "Formato " << argv[0] << " num_elem" << endl;
    return -1;
  }

  int n = atoi(argv[1]);
  vector<int> rankings(n);
  srandom(time(0));
  for (int i=0; i < n; i++)
    rankings[i] = i;

  random_shuffle(rankings.begin(), rankings.end());

  chrono::high_resolution_clock::time_point tantes, tdespues;
  chrono::duration<double> transcurrido;
```

```
38    int n_inv;
39
40    tantes = chrono::high_resolution_clock::now();
41    n_inv = inversiones(rankings);
42    tdespues = chrono::high_resolution_clock::now();
43
44    transcurrido = chrono::duration_cast<chrono::duration<double>>(tdespues -
          tantes);
45
46    #ifndef GP_OUT
47      cout << "ranking: ";
48      for (int i=0; i<n; i++)
49        cout << rankings[i] << " ";
50      cout << endl;
51      cout << "Num inversiones: " << n_inv << endl;
52    #else
53      cout << n << " " << transcurrido.count() << endl;
54    #endif
55
56  }
```

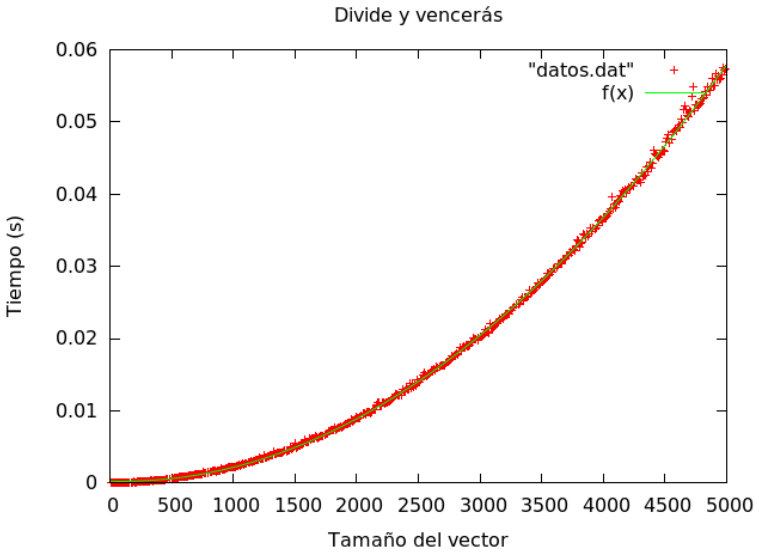

Figura 9: Subproblemas

3.3. Divide y vencerás

El mismo problema usando un programa que utiliza divide y vencerás:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <ctime>
5  #include <climits>
6  #include <cstdlib>
```

```
7  #include <chrono>
8  using namespace std;
9
10
11
12 /* ********************************************************** */
13 /*  M\'etodo de ordenaci\'on por mezcla  */
14
15 /**
16    @brief Devuelve el numero de inversiones de un ranking.
17
18    @param T: vector de elementos. Debe tener num_elem elementos.
19              Es MODIFICADO.
20    @param num_elem: n\'umero de elementos. num_elem > 0.
21
22    @return el numero de inversiones
23
24    Calcula el numero de inversiones aplicando el algoritmo de mezcla.
25    Como consecuencia tambien ordena el vector T.
26 */
27 inline static
28 int inversiones(int T[], int num_elem);
29
30
31
32 /**
33    @brief Devuelve el n\'umero de inversiones de un ranking.
34
35    @param T: vector de elementos. Tiene un n\'umero de elementos
36                   mayor o igual a final. Es MODIFICADO.
37    @param inicial: Posici\'on que marca el incio de la parte del
38                   vector a ordenar.
39    @param final: Posici\'on detr\'as de la \'ultima de la parte del
40                   vector a ordenar.
41                   inicial < final.
42
43   Calcula el n\'umero de inversiones aplicando el algoritmo de mezcla.
44   Como consecuencia tambien ordena el vector T.
45 */
46 void inversiones_recursivo(int T[], int inicial, int final, int&
      num_inversiones);
47
48
49 /**
50    @brief Ordena un vector por el m\'etodo de inserci\'on.
51
52    @param T: vector de elementos. Debe tener num_elem elementos.
53              Es MODIFICADO.
54    @param num_elem: n\'umero de elementos. num_elem > 0.
55
56    Cambia el orden de los elementos de T de forma que los dispone
57    en sentido creciente de menor a mayor.
58    Aplica el algoritmo de inserci\'on.
59 */
```

```
60  inline static
61  void insercion(int T[], int num_elem);
62
63
64  /**
65     @brief Ordena parte de un vector por el m\'etodo de inserci\'on.
66
67     @param T: vector de elementos. Tiene un n\'umero de elementos
68                     mayor o igual a final. Es MODIFICADO.
69     @param inicial: Posici\'on que marca el incio de la parte del
70                     vector a ordenar.
71     @param final: Posici\'on detr\'as de la \'ultima de la parte del
72                     vector a ordenar.
73                     inicial < final.
74
75     Cambia el orden de los elementos de T entre las posiciones
76     inicial y final - 1 de forma que los dispone en sentido creciente
77     de menor a mayor.
78     Aplica el algoritmo de la inserci\'on.
79  */
80  static void insercion_lims(int T[], int inicial, int final);
81
82
83  /**
84     @brief Mezcla dos vectores ordenados sobre otro.
85
86     @param T: vector de elementos. Tiene un n\'umero de elementos
87                     mayor o igual a final. Es MODIFICADO.
88     @param inicial: Posici\'on que marca el incio de la parte del
89                     vector a escribir.
90     @param final: Posici\'on detr\'as de la \'ultima de la parte del
91                     vector a escribir
92                     inicial < final.
93     @param U: Vector con los elementos ordenados.
94     @param V: Vector con los elementos ordenados.
95               El n\'umero de elementos de U y V sumados debe coincidir
96               con final - inicial.
97
98     @return: El n\'umero de elementos que estaban desordenados
99
100    En los elementos de T entre las posiciones inicial y final - 1
101    pone ordenados en sentido creciente, de menor a mayor, los
102    elementos de los vectores U y V.
103 */
104 int fusion(int T[], int inicial, int final, int U[], int V[]);
105
106
107
108 /**
109    Implementaci\'on de las funciones
110 **/
111
112
113 int inversiones(int T[], int num_elem)
```

```
114 {
115   int n = 0;
116   inversiones_recursivo(T, 0, num_elem, n);
117   return n;
118 }
119
120 void inversiones_recursivo(int T[], int inicial, int final, int&
      num_inversiones)
121 {
122   if (final - inicial < 2)
123     {
124       return;
125     } else {
126       int k = (final - inicial)/2;
127
128       int * U = new int [k - inicial];
129       int l, l2;
130       for (l = 0, l2 = inicial; l < k; l++, l2++)
131              U[l] = T[l2];
132
133       int * V = new int [final - k];
134       for (l = 0, l2 = k; l < final - k; l++, l2++)
135             V[l] = T[l2];
136
137       inversiones_recursivo(U, 0, k, num_inversiones);
138       inversiones_recursivo(V, 0, final - k, num_inversiones);
139       num_inversiones += fusion(T, inicial, final, U, V);
140       delete [] U;
141       delete [] V;
142     };
143 }
144
145
146 int fusion(int T[], int inicial, int final, int U[], int V[])
147 {
148   int n_desordenados = 0;
149   int k = (final-inicial)/2;
150   for (int i =0; i < k-inicial; i++)
151     for (int j=0; j < final-k; j++)
152       if (U[i] > V[j])
153         n_desordenados++;
154
155   int i, j;
156   for (i =0; i < k-inicial; i++)
157     T[i] = U[i];
158   for (i=0, j=k-inicial; j < final-k; i++, j++)
159     T[j] = V[i];
160
161   return n_desordenados;
162 }
163
164
165 int main(int argc, char** argv){
166   if (argc < 2){
```

```
167       cerr << "Formato " << argv[0] << " num_elem" << endl;
168       return -1;
169     }
170 
171     int n = atoi(argv[1]);
172     int* rankings = new int[n];
173     srandom(time(0));
174     for (int i=0; i < n; i++){
175       rankings[i] = i;
176     }
177     random_shuffle(rankings, rankings+n);
178 
179   #ifndef GP_OUT
180     cout << "ranking: ";
181     for (int i=0; i<n; i++)
182       cout << rankings[i] << " ";
183     cout << endl;
184   #endif
185 
186     chrono::high_resolution_clock::time_point tantes, tdespues;
187     chrono::duration<double> transcurrido;
188     int n_inv;
189 
190     tantes = chrono::high_resolution_clock::now();
191     n_inv = inversiones(rankings, n);
192     tdespues = chrono::high_resolution_clock::now();
193     transcurrido = chrono::duration_cast<chrono::duration<double>>(tdespues -
          tantes);
194 
195   #ifndef GP_OUT
196     cout << "Num inversiones: " << n_inv << endl;
197   #else
198     cout << n << " " << transcurrido.count() << endl;
199   #endif
200   }
```

3.4. Comparación

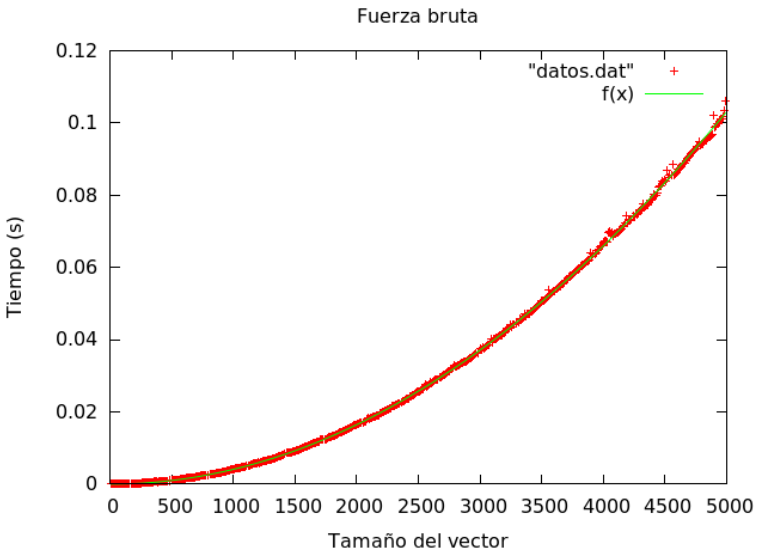

Figura 10: Subproblemas

4. Bibliografia

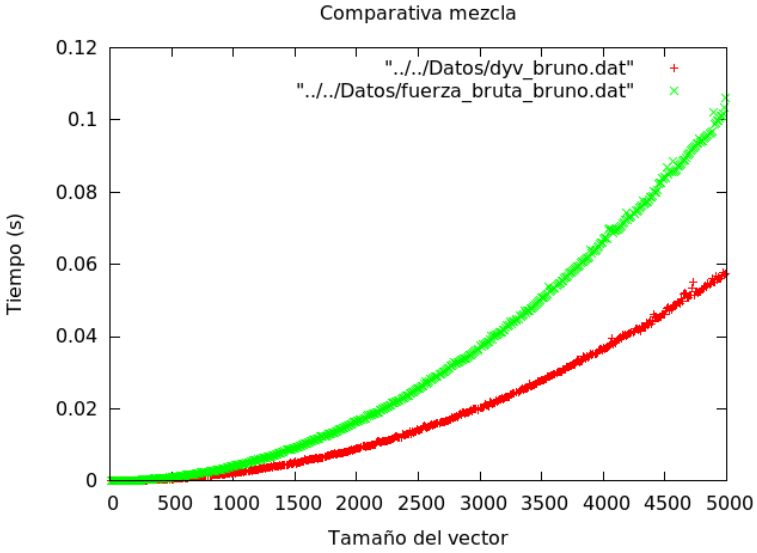

Figura 11: Subproblemas