Práctica 2: Algoritmos Divide y Vencerás

Por: Sara Bellarabi El Fazazi, Manuel Villatoro Guevara , Arturo Cortés Sánchez, Sergio Vargas Martin

1.1. Traspuesta de una matriz: Análisis teórico fuerza bruta

trasponerV1() realiza n/2 intercambios.

trasponerV2() realiza n copias.

Ambos son de orden lineal, pero trasponerV2() puede procesar matrices rectangulares.

```
1   typedef vector<vector<char>> matriz;
2
3   void transponerV1(matriz &a) {
4    for (int i = 0; i < a.size() - 2; i++)
5      for (int k = i + 1; k < a.size() - 1; k++)
6        swap(a[i][k], a[k][i]);
7   }
8
9   matriz trasponerV2(const matriz &m) {
10   matriz m2(m[0].size(),
    vector<char>(m.size()));
11
12   for (int i = 0; i < m.size(); i++)
13     for (int j = 0; j < m[0].size(); j++)
14       m2[j][i] = m[i][j];
15
16   return m2;
17  }
18
19
```

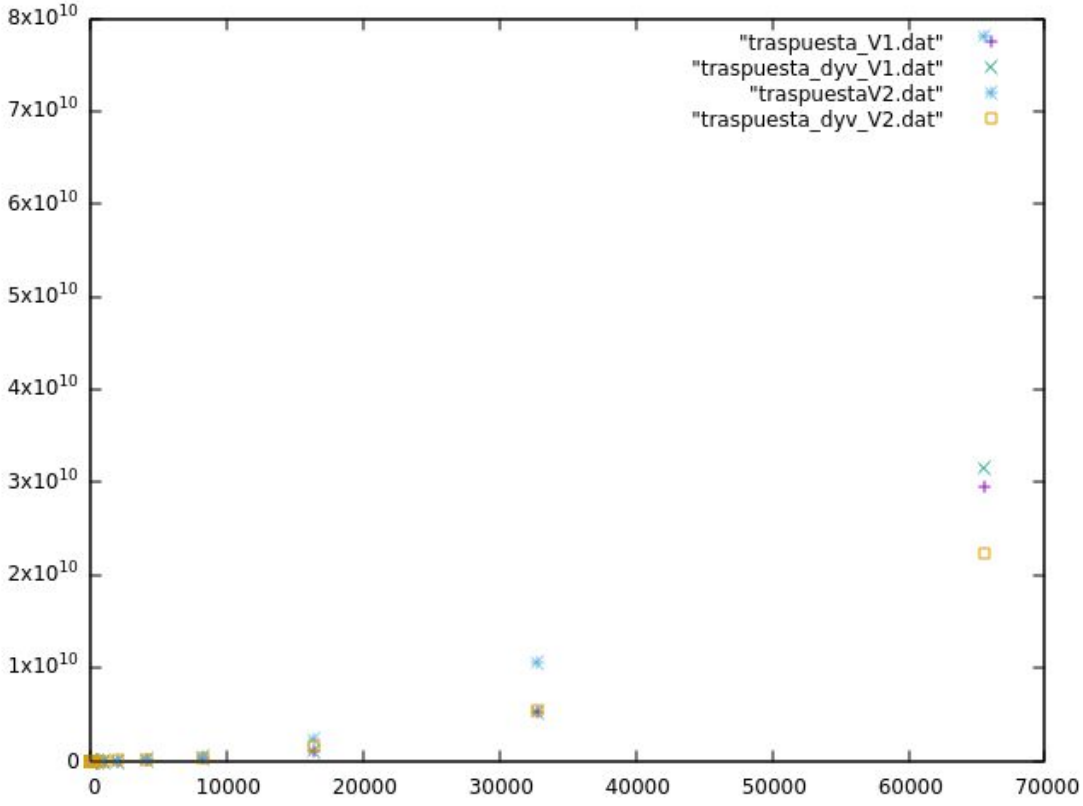
1.2. Traspuesta de una matriz: Análisis teórico DyV

De la misma forma que antes trasponer_dyvV1() realiza n/2 intercambios y trasponer_dyvV2() realiza n copias.

Ambos siguen siendo de orden lineal pero igual que antes trasponer_dyvV2() puede procesar matrices rectangulares.

```
1   void transponer_dyvV1(matriz &a, int i, int j) {
2    if (abs(i - j) == 1) {
3      for (int k = i + 1; k < a.size() ; k++)
4        swap(a[i][k], a[k][i]);
5    } else {
6      transponer_dyvV1(a, i, (j + i) / 2);
7      transponer_dyvV1(a, (j + i) / 2, j);
8    }
9   }
10  void traspuesta_dyvV2(const matriz &entrada, matriz &salida,
11   int x0, int x1,int y0, int y1) {
12   if (abs(y0 - y1) == 0) {
13     salida[x1][y1] = entrada[y1][x1];
14     salida[x0][y0] = entrada[y0][x0];
15   } else {
16     int x_mid = (x1 + x0) / 2;
17     int y_mid = (y1 + y0) / 2;
18     traspuesta_dyvV2(entrada, salida, x0, x_mid +1, y0, y_mid);
19     traspuesta_dyvV2(entrada, salida, x0, x_mid +1, y_mid + 1, y1);
20     traspuesta_dyvV2(entrada, salida, x_mid, x1, y0, y_mid);
21     traspuesta_dyvV2(entrada, salida, x_mid, x1, y_mid +1, y1);
22   }
23  }
```

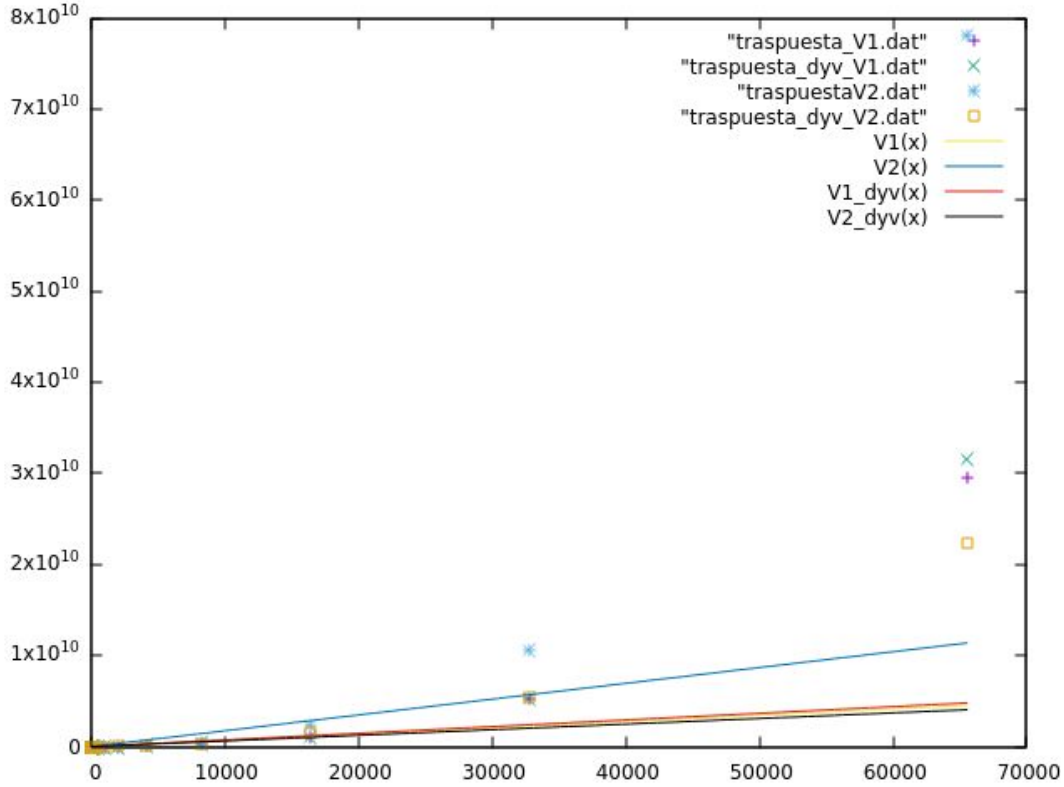
1.3. Traspuesta de una matriz: Análisis empírico



Cálculo de constantes ocultas

```
1   #include <cmath>
2   #include <fstream>
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   template <class T> double media_vector(vector<T> v) {
8     double k = 0;
9     for (auto i : v)
10      k += i;
11    return k / v.size();
12  }
13  double orden(double f) {
14    return f; // Orden del algoritmo
15  }
16  int main(int argc, char *argv[]) {
17   fstream file;
18  for (int i = 1; i <= argc; i++) {
19    vector<double> vec;
20    double fx, tx;
21    file.open(argv[i]);
22    while (file) {
23      file >> fx >> tx;
24      vec.push_back(tx / orden(fx));
25    }
26    cout << fixed << argv[i] <<" K: " << media_vector(vec) <<
    endl;
27    file.close();
28  }
29  }
```

1.4. Traspuesta de una matriz: Análisis híbrido



traspuesta_dyv_V1.dat K: 73118.824188
traspuesta_dyv_V2.dat K: 61643.950594
traspuesta_V1.dat K: 69046.233163
traspuestaV2.dat K: 173564.245605

1.5. Estudio del umbral V1

Al ser de orden lineal, las gráficas nunca van a tener un punto de corte. Para hallar el umbral miramos directamente la tabla de tiempos. En este caso vemos que la version de fuerza bruta siempre es mejor que la divide y vencerás, por lo que no hay umbral.

N	Fuerza bruta(t)	Divide y venceras (t)
2	347	387
4	244	320
8	412	703
16	987	1387
32	2507	3730
64	8236	11350
128	32040	40290
256	146720	147943
512	713686	718640
1024	863345	2578256
2048	3995161	4918089
4096	49497357	51697661
8192	250841178	251244280
16384	1096548907	1110516472
32768	5189665245	5254973261
65536	29498126890	31575964762

1.6. Estudio del umbral V2

De igual manera que antes, las gráficas son lineales por lo tanto no tienen un punto en común.

Para ver el umbral tenemos que fijarnos en la tabla de tiempos.

A partir de N=4096, es más eficiente utilizar el algoritmo Divide y vencerás.

N	Fuerza bruta(t)	Divide y venceras (t)
2	4303	9232
4	1892	2856
8	2248	4348
16	4080	9576
32	9100	31832
64	40528	109166
128	112830	396717
256	425701	1427165
512	2260010	2498899
1024	2043310	5083034
2048	8737627	22465887
4096	111298673	83556186
8192	513980989	369877279
16384	2241478308	1481458946
32768	10534747606	5522607984
65536	78154152038	22438807118

1.7. Descripción caso de ejecución

Algoritmo fuerza bruta: matrices cuadradas

i = 0
k = 1 hasta < a.size() (2)

swap a[0][1] = a[1][0]

FIN

Algoritmo fuerza bruta: matrices no cuadradas

2	4	5
6	7	8

m1.size()=2 m1[0].size = 3

i=0; j=0

m2 =[j] [i] = m1[i] [j]

2	6
4	7
5	8

1.8. Descripción caso de ejecución

Algoritmo divide y vencerás: matrices cuadradas

i=0, j=2 (inicio, y fin)
abs(i-j) != 1
por tanto se ejecuta el bloque else:

trasponer_dyv(a, i, (j+i)/2); i=0, j=1 → abs(i-j) ==1 → swap a [0][1] 0 a[1][0]
trasponer_dyv(a, i, (j+i)/2); i=1, j=2 → abs(i-j) ==1 → swap a [1][2] 0 a[2][1]

Algoritmo divide y vencerás: matrices no cuadradas

2	4	5
6	7	8

m1.size()=2
m1[0].size()=3

m2:

x0=0

x1=2

y0=0

y1=1

x_mid=1

y_mid=0

Llamadas a función:

x0=0; x1=2; y0=0; y1=0

2	
5	

1

x0=0; x1=2; y0=1; y1=1

2	6
5	8

2

---

x0=1; x1=2; y0=0; y1=0

2	6
4	
5	8

3

x0=1; x1=2; y0=1; y1=1

2	6
4	7
5	8

4

2.1. Comparación de preferencias: Análisis teórico fuerza bruta

El algoritmo va comprobando todos los pares (i,j)

- $O(n^2)$

En el mejor caso, los usuarios tendrían la misma preferencia de rankings.

El algoritmo no comprueba este caso:

- $\Omega(n^2)$

```
1  int inversiones(vector<int> v) {
2   int inv = 0;
3   for (int i = 0; i < v.size() - 1; i++)
4     for (int j = i + 1; j < v.size(); j++)
5       if (v[i] > v[j])
6         inv++;
7
8   return inv;
9  }
```

2.2. Comparación de preferencias: D y V

```
1   int inversiones(vector<int> &vec, int iz,
    int der) {
2    int centro = (iz + der) / 2;
3    if (iz >= der) {
4      return 0;
5    } else {
6      return inversiones(vec, iz, centro) +
    inversiones(vec, centro + 1, der) +
7             fusion(vec, iz, der);
8    }
9   }
```

```
1   int fusion(vector<int> &vec, int iz, int der) {
2    vector<int> aux(der - iz + 1);
3    int centro = (iz + der) / 2, inv = 0, i = iz, j = centro
    + 1, k;
4
5    for (k = 0; i <= centro && j <= der; k++) {
6      if (vec[i] <= vec[j]) {
7        aux[k] = vec[i];
8        i++;
9      } else {
10       aux[k] = vec[j];
11       j++;
12       inv += centro - i + 1;
13     }
14   }
15   for (; i <= centro; k++, i++)
16     aux[k] = vec[i];
17   for (; j <= der; k++, j++)
18     aux[k] = vec[j];
19
20   copy(aux.begin(), aux.begin() + der - iz + 1,
    vec.begin() + iz);
21
22   return inv;
23  }
24
```

2.2.1. Comparación de preferencias: Análisis teórico D y V

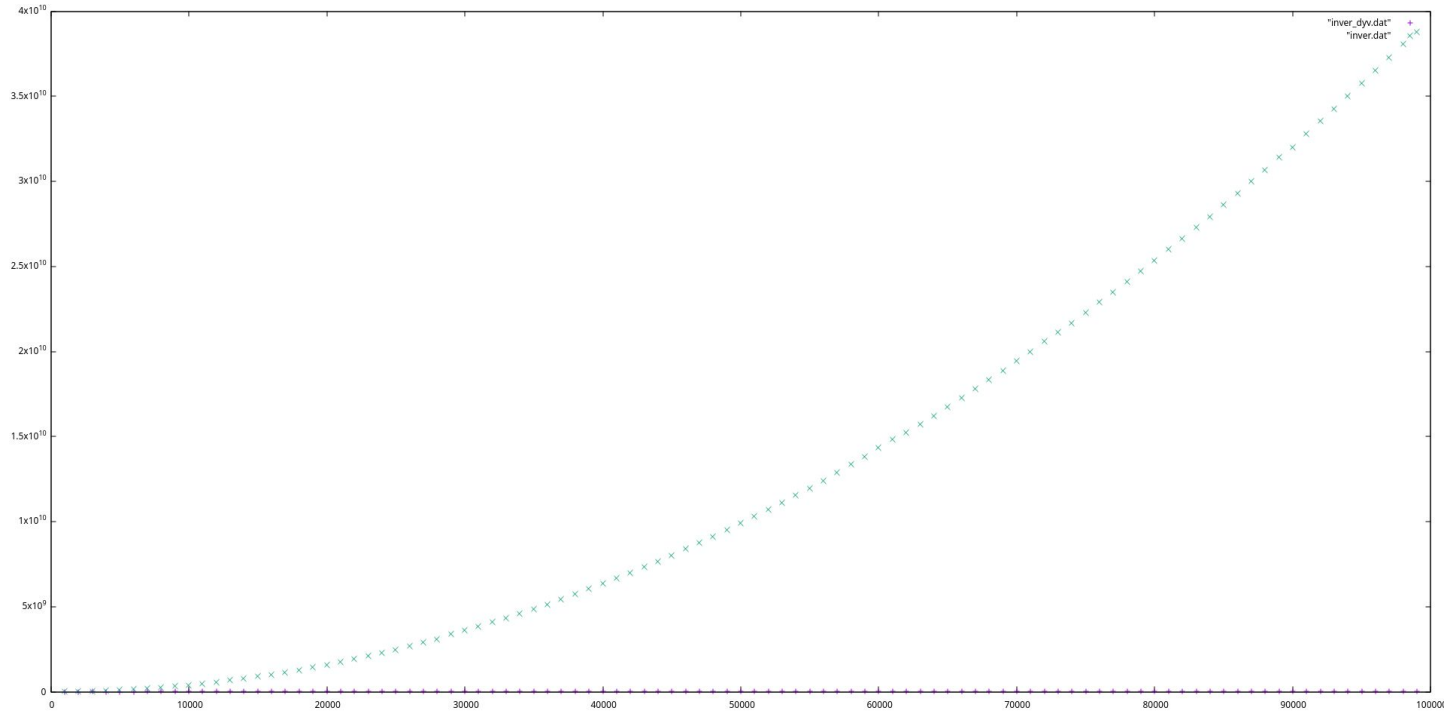
Ecuación del tiempo de ejecución:

$T(n) <= T(n/2) + T(n/2) + O(n) \longrightarrow T(n) = O(n\log(n))$

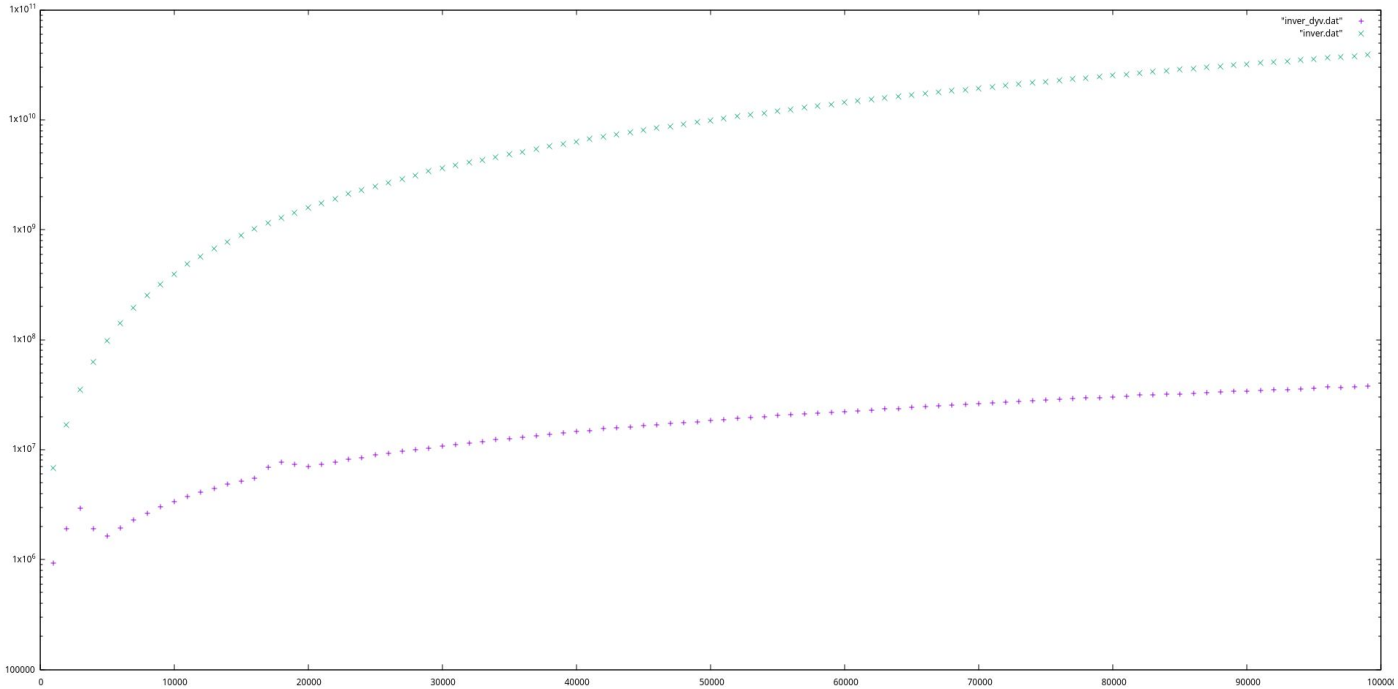
En el mejor caso:

- $\Omega(n\log(n))$

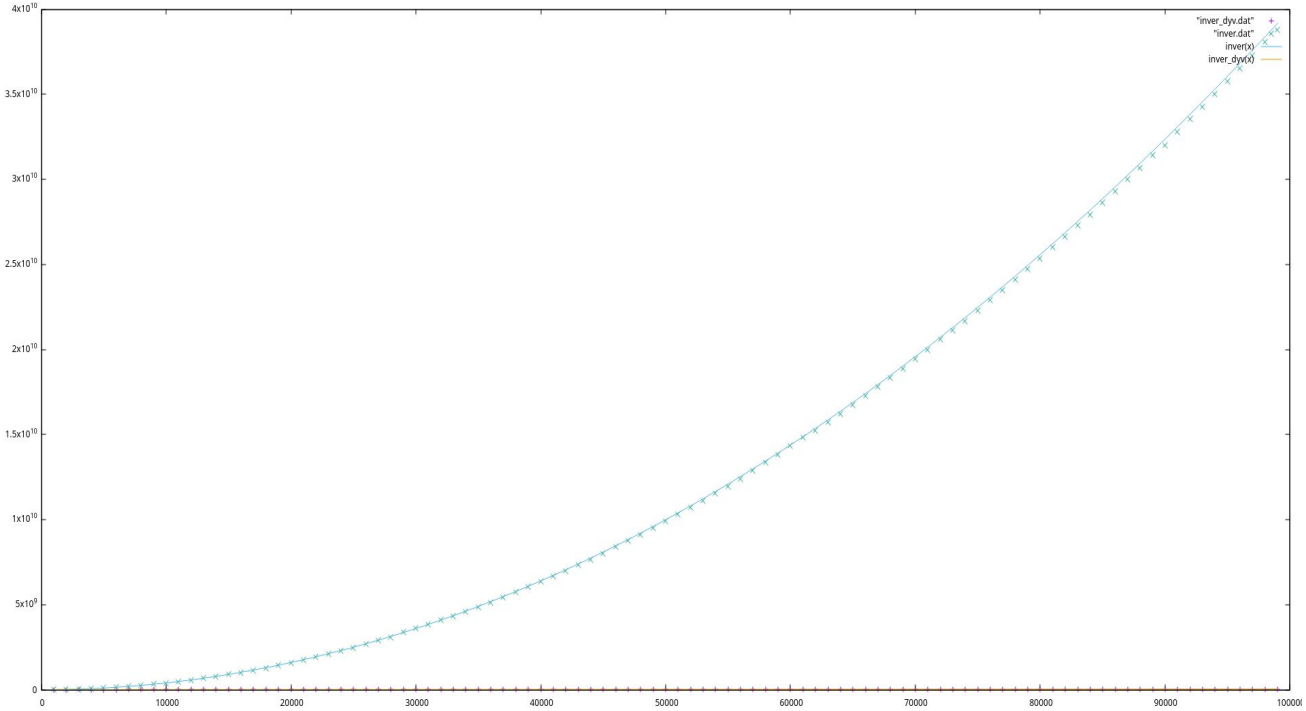
2.3 Comparación de preferencias: Análisis empírico



2.4. Comparación de preferencias: Análisis empírico en escala log



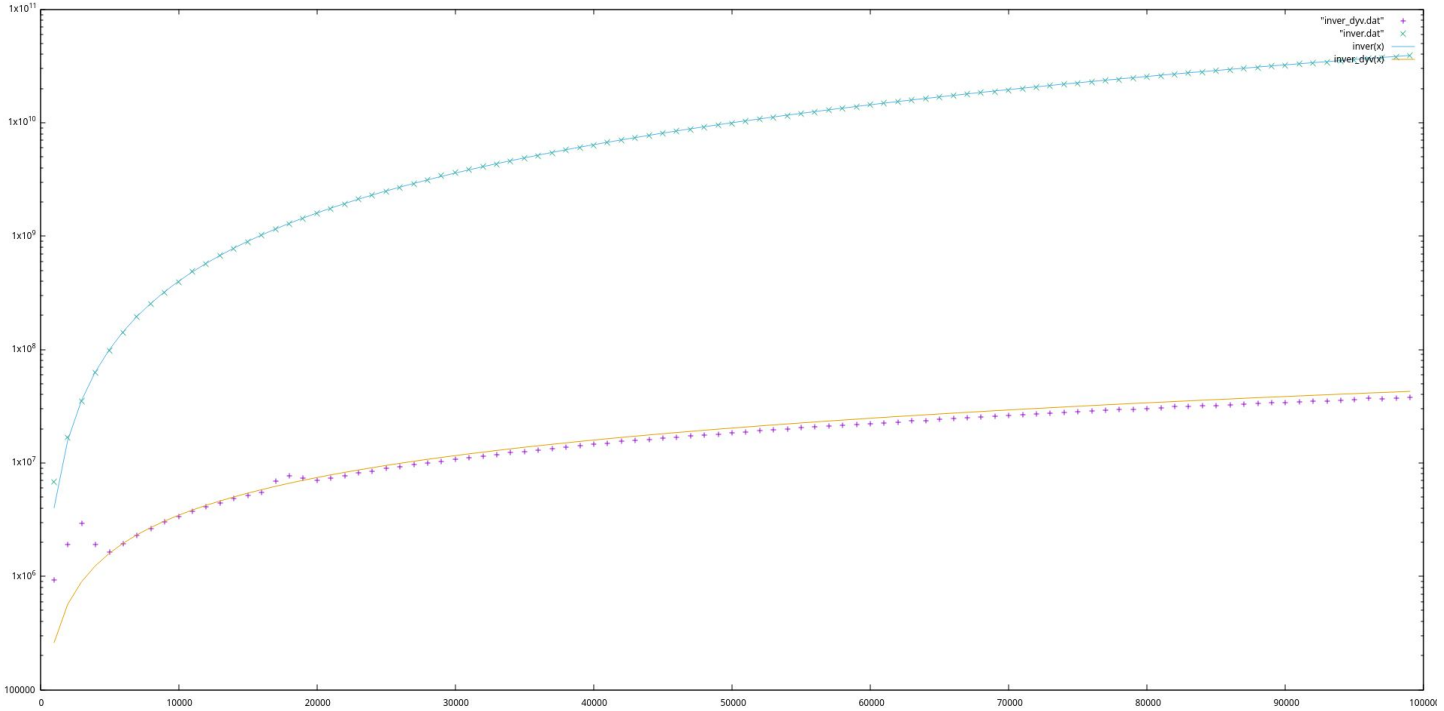
2.5. Comparación de preferencias: Análisis híbrido



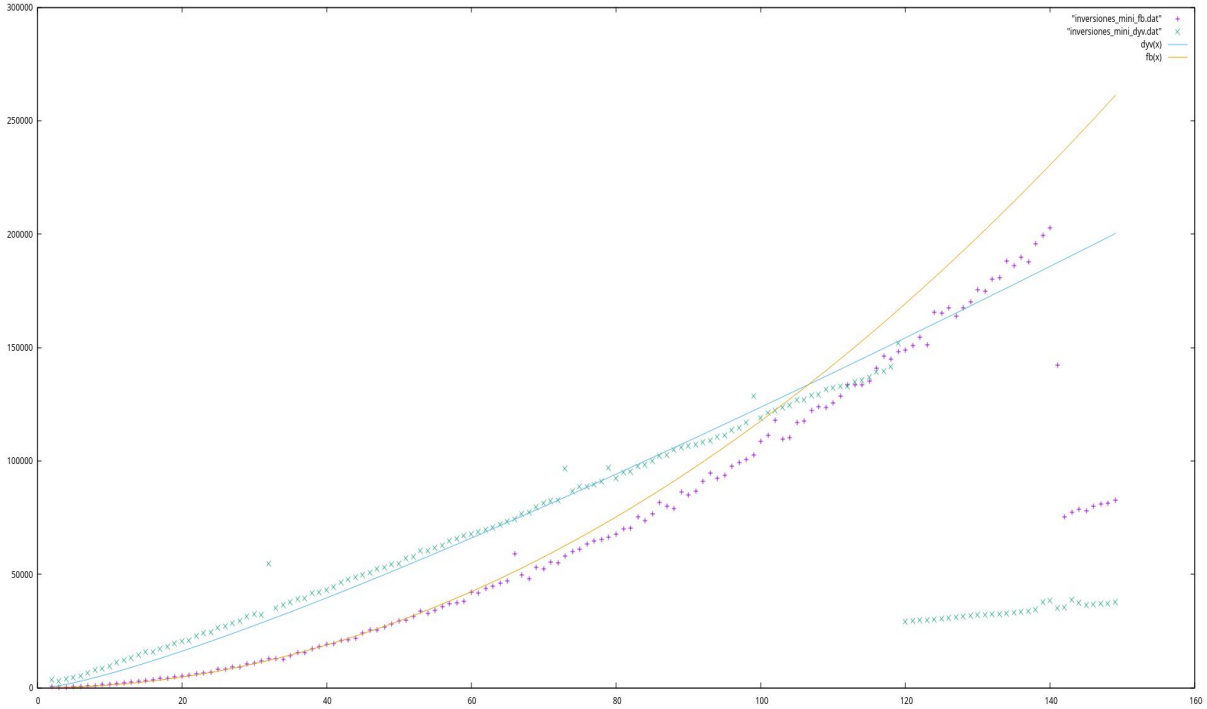
inver.dat K: 3.996632

inver_dyv.dat K: 37.512211

2.6. Comparación de preferencias: Análisis híbrido en escala log



2.7. Estudio del umbral



inversiones_mini_fb.dat K: 11.763392
inversiones_mini_dyv.dat K: 268.567322

Resolvemos sistema de ecuaciones, compuesto por las dos funciones híbridas de los algoritmos:

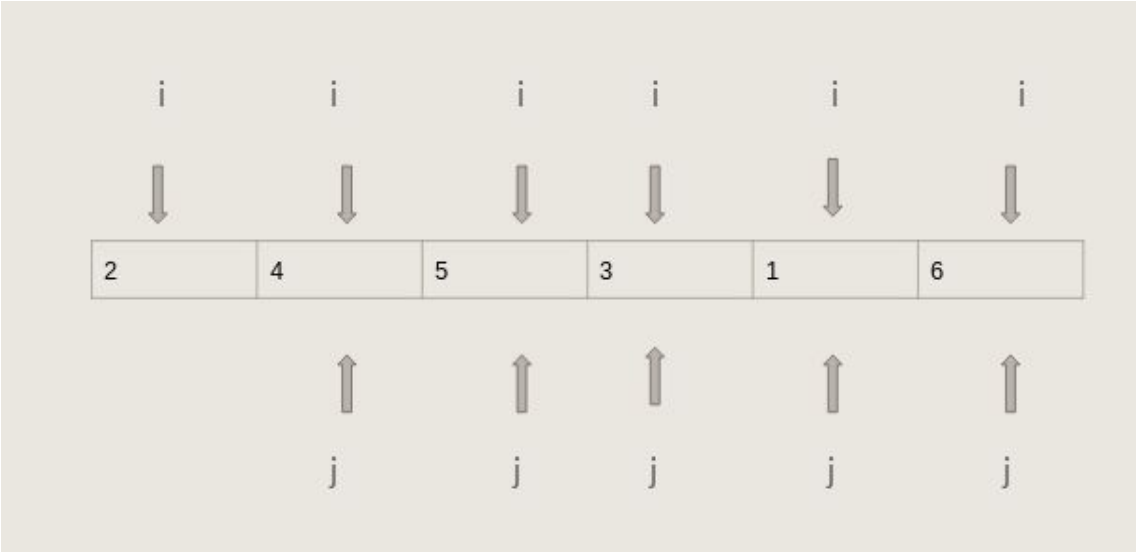
$y=x \cdot \log(x) \cdot 268.567322$
$y=11.763392 \cdot x^2$

Resultado: x = 106.598, y = 133670

Por tanto para tamaños de problema superiores a 106 es mejor el algoritmo divide y vencerás, para tamaños inferiores es mejor el fuerza bruta.

2.8. Descripción caso de ejecución

Algoritmo fuerza bruta:

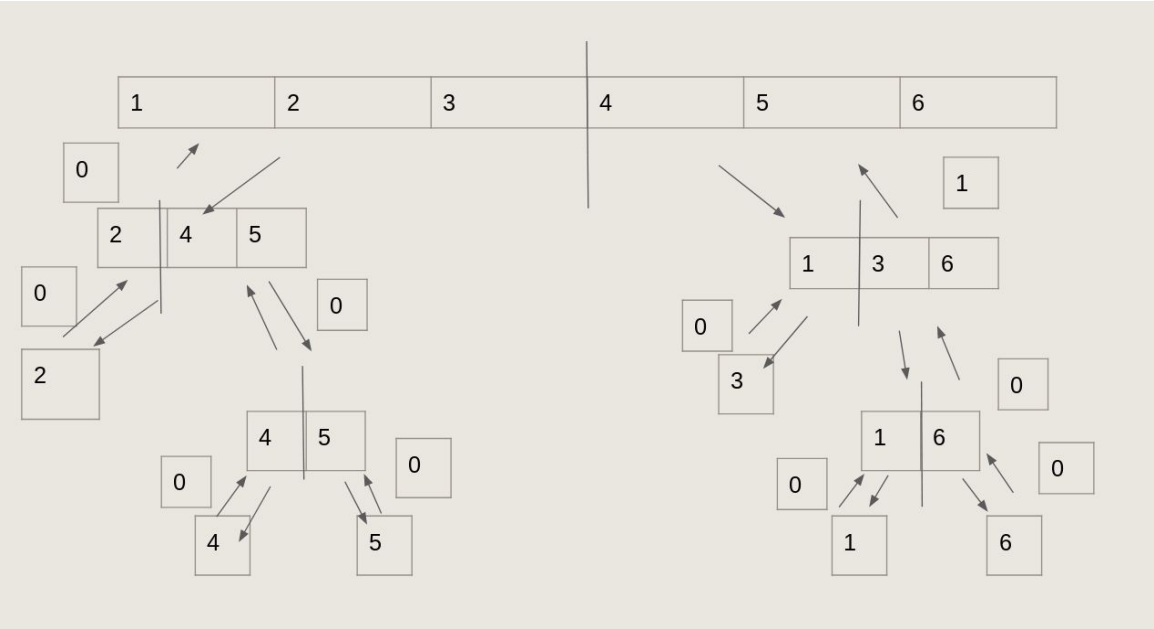


v[i]>v[j]? → No → Inversión = 6

Número total de inversiones= 6

2.9. Descripción caso de ejecución

Algoritmo divide y vencerás:



Número total de inversiones= 6