

**ALGORÍTMICA (2018-2019)**  
DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS  
UNIVERSIDAD DE GRANADA

---

# Memoria Práctica 3

## Grupo Azul

---

Irene Huertas González,  
Javier Alcántara García,  
César Muñoz Reinoso

May 14, 2019

# Contents

|  |           |
|--|-----------|
| <b>1 Problema común. TSP</b>   | <b>3</b>  |
| 1.1 Enunciado . . . . .  | 3         |
| 1.2 Algoritmos . . . . .   | 3         |
| 1.2.1 Basado en cercanía . . . . .                                   | 3         |
| 1.2.2 Basado en mejor inserción . . . . .                            | 4         |
| 1.2.3 Basado en mejor arista . . . . .                               | 6         |
| 1.3 Complejidad . . . . .  | 7         |
| 1.3.1 Traspuesta sencilla . . . . .                                  | 7         |
| 1.3.2 Traspuesta Divide y Vencerás . . . . .                         | 7         |
| 1.4 Análisis empírico . . . . .                                      | 7         |
| 1.4.1 Comparación algoritmo sencillo con Divide y Vencerás . . . . . | 7         |
| 1.4.2 Comparación entre ordenadores de la empírica . . . . .         | 9         |
| 1.5 Análisis híbrido . . . . .                                       | 10        |
| 1.5.1 Análisis híbrido Traspuesta . . . . .                          | 10        |
| 1.5.2 Análisis híbrido Traspuesta Divide y Veceras . . . . .         | 11        |
| 1.6 Ejemplo de ejecución . . . . .                                   | 13        |
| 1.6.1 Traspuesta sencilla . . . . .                                  | 13        |
| 1.6.2 Traspuesta Divide y Venceras . . . . .                         | 16        |
| <b>2 Problema asignado: Serie unimodal de números</b>                | <b>19</b> |
| 2.1 Enunciado . . . . .  | 19        |
| 2.2 Algoritmos . . . . .   | 19        |
| 2.2.1 Unimodal sencillo . . . . .                                    | 19        |
| 2.2.2 Unimodal Divide y Vencerás . . . . .                           | 20        |
| 2.3 Complejidad . . . . .  | 22        |
| 2.3.1 Unimodal sencillo . . . . .                                    | 22        |
| 2.3.2 Unimodal Divide y vencerás . . . . .                           | 22        |
| 2.4 Análisis empírico . . . . .                                      | 23        |
| 2.4.1 Comparación algoritmo sencillo con Divide y vencerás . . . . . | 23        |
| 2.4.2 Comparación entre ordenadores de la empírica . . . . .         | 26        |
| 2.5 Análisis híbrido . . . . .                                       | 28        |
| 2.5.1 Análisis híbrido Unimodal . . . . .                            | 28        |
| 2.5.2 Análisis híbrido Unimodal Divide y Vencerás . . . . .          | 29        |
| 2.6 Ejemplo de ejecución . . . . .                                   | 31        |
| 2.6.1 Unimodal sencilla . . . . .                                    | 31        |
| 2.6.2 Unimodal Divide y Vencerás . . . . .                           | 31        |

# 1 Problema común. TSP

## 1.1 Enunciado

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Mas formalmente, dado un grafo  $G$ , conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

## 1.2 Algoritmos

### 1.2.1 Basado en cercanía

Comenzamos seleccionando la ciudad inicial como la 1. Recorremos la fila 1 y seleccionamos la ciudad a menor distancia (columnas). Cuando hacemos esto saltamos a la fila de la ciudad seleccionada y volvemos a realizar este procedimiento. Tenemos en cuenta que no podemos introducir ciudades repetidas.

Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

### 1.2.2 Basado en mejor inserción

Primero creamos una matriz con la distancias entre las ciudades, utilizando el algoritmo de Euclides. Comenzamos seleccionando la ciudad inicial como la 1. Recorremos la fila 1 y seleccionamos la ciudad a menor distancia (columnas). Cuando hacemos esto saltamos a la fila de la ciudad seleccionada y volvemos a realizar este procedimiento. Tenemos en cuenta que no podemos introducir ciudades repetidas. Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

```
list<int> TSP(const vector<vector<int>> &matriz, const map<int,
pair<double, double>> &m, int &distancia_final) {
    int n=1;
    int s=1;
    int w=1;

    list<int> salida;
    map<int, pair<double, double>>::const_iterator it = m.begin();
    list<int> ciudades_restantes;
    ciudades_restantes.push_back(1);

    pair<double, double> pn=(*it).second,
    ps=(*it).second, pw=(*it).second;
    it++;

    for(it; it != m.end(); it++) {
        ciudades_restantes.push_back((*it).first);
        pair<double, double> p=(*it).second;
        int ciudad_actual=(*it).first;

        if(p.first<pw.first) { //Elige la de mas al oeste
            pw=p;
            w=ciudad_actual;
        }
        else if(p.second>pn.second || (n==w && n==1)) {
            pn=p;
            n=ciudad_actual;
        }
        else if(p.second<ps.second || (s==w && s==1)) {
            ps=p;
            s=ciudad_actual;
        }
    }
}
```

```

    salida.push_back(n);
    salida.push_back(s);
    salida.push_back(w);

    ciudades_restantes.remove(n);
    ciudades_restantes.remove(s);
    ciudades_restantes.remove(w);

    while(ciudades_restantes.size() > 0) {
        pair<int ,list<int >::iterator> pres;
        pres = ElegirCiudad(salida , matriz , ciudades_restantes);
        InsertarCiudad(salida , pres.first , pres.second);
    }

    salida.push_back(*(salida.begin()));
    distancia_final = DistanciaTotal(salida , matriz);

    return salida;
}

```

### 1.2.3 Basado en mejor arista

Creamos un mapa formado por la distancia entre dos ciudades y un par con esas dos ciudades. Vamos seleccionando las distancias menores entre dos ciudades y añadiendo estas al conjunto de seleccionados. Para poder insertar una nueva ciudad, esta no puede tener más de 2 aristas ni crear un ciclo. Cómo el camino resultante de este algoritmo no es cerrado, tenemos que recorrer el camino y añadir la distancia entre las 2 ciudades que únicamente tienen una arista.

```
vector<int> TSP(multimap<int , pair<int ,int> >
& aristas_par_ciudades , int & distancia) {
    vector<int> salida;
    list<pair<int ,int> > aux;
    multimap<int , pair<int ,int> >::iterator it =
    aristas_par_ciudades.begin();
    aux.push_back((*it).second);
    distancia += (*it).first;
    it++;
    while(it != aristas_par_ciudades.end()) {
        if (CiudadFactible(aux, (*it).second) && !ExistenCiclos(aux,
        (*it).second)) {
            aux.push_back((*it).second);
            distancia += (*it).first;
        }
        it++;
    }
    CerrarCiclo(aux, aristas_par_ciudades, distancia);
    salida = TransformaACamino(aux);

    return salida;
}
```

## 1.3 Complejidad

### 1.3.1 Traspuesta sencilla

Cuando se llama a la función traspuesta, nos encontramos con dos bucles for anidados tras los cuales se llama a la función swap que son operaciones elementales sin ninguna relevancia para el calculo del orden de eficiencia. Se nos queda por lo tanto una sumatoria del tipo

$$\sum_{i=0}^n \left( \sum_{i+1}^n (11) \right) = \sum_{i=0}^n (11(n-i-1)) = 11 * (n^2 - n(i+1))$$

Por lo tanto la eficiencia teórica es de orden  $O(n^2)$

### 1.3.2 Traspuesta Divide y Vencerás

Analizamos linea por linea la eficiencia y la expresamos en una formula

```
void traspuestaDyV (vector< vector<double> >& m, int fini , int fult ,
int cini , int cult) {
    if (fini < fult) {
        int fmed = (fini + fult) / 2;           O(1)
        int cmed = (cini + cult) / 2;           O(1)
        traspuestaDyV (m, fini , fmed, cini , cmed);   O(n/2)
        traspuestaDyV (m, fini , fmed, cmed+1, cult);  O(n/2)
        traspuestaDyV (m, fmed+1, fult , cini , cmed);  O(n/2)
        traspuestaDyV (m, fmed+1, fult , cmed+1, cult); O(n/2)
        swap (m, fmed+1, cini , fini , cmed+1, fult - fmed); O(n)
    }
}
```

$$T(n) = 4T(n/2) + n \in O(n^2)$$

Luego la eficiencia es cuadrática, igual que la del algoritmo sencillo

## 1.4 Análisis empírico

### 1.4.1 Comparación algoritmo sencillo con Divide y Vencerás

Se ve claramente que el orden es cuadrático y que es notablemente mejor emplear el sencillo si la matriz es grande.

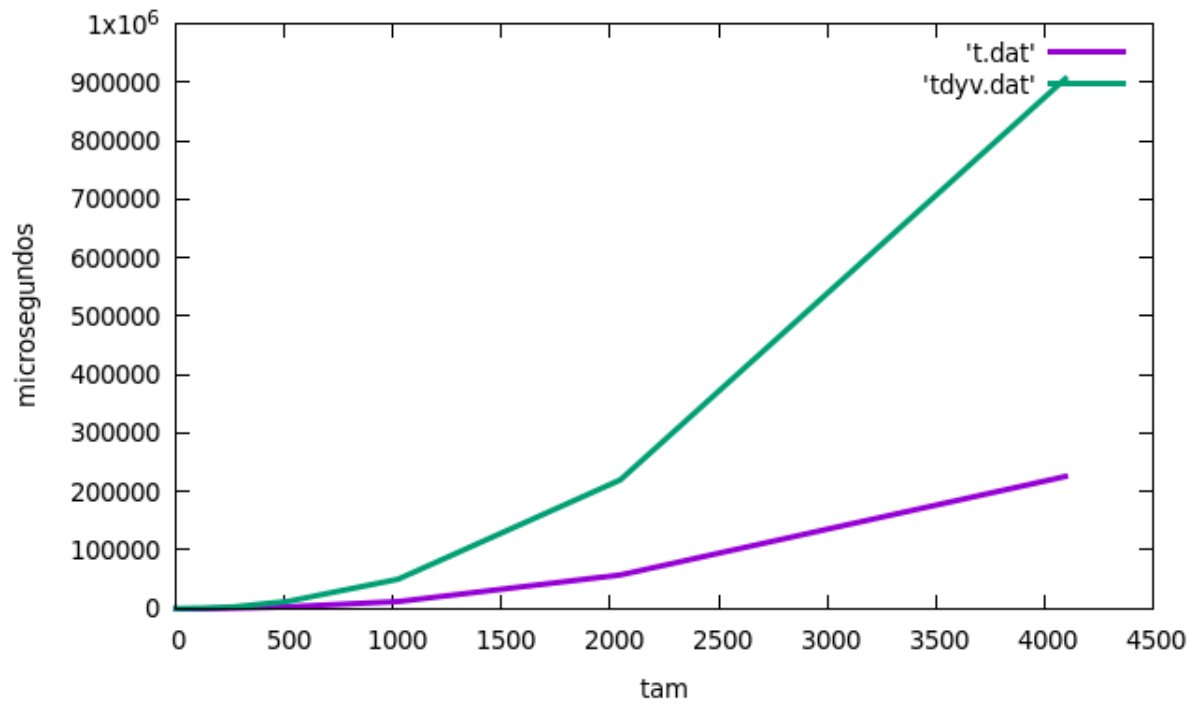


Figure 1.1: Traspuesta sencilla y DyV de Irene

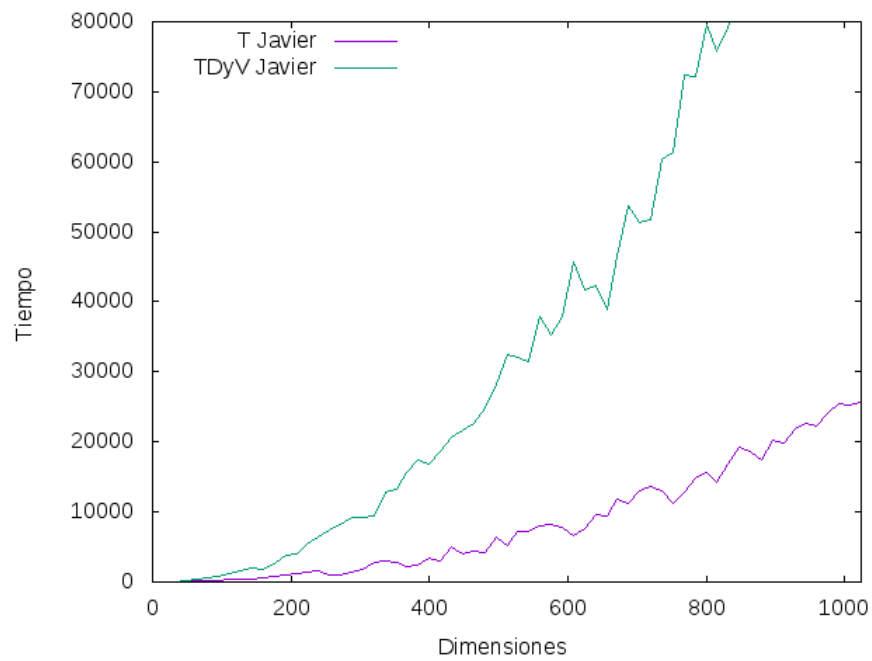


Figure 1.2: Traspuesta sencilla y DyV de Javier



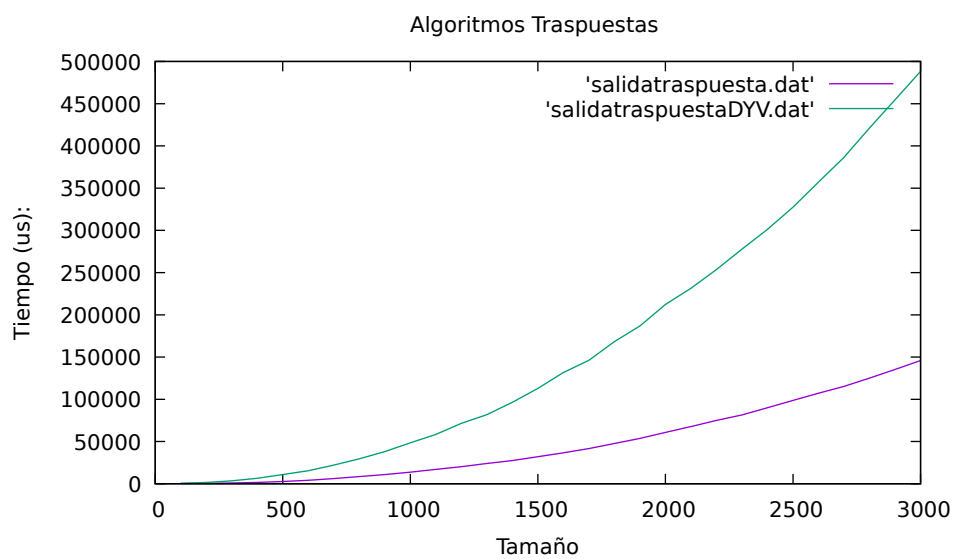


Figure 1.3: Traspuesta sencilla y DyV de César

#### 1.4.2 Comparación entre ordenadores de la empírica

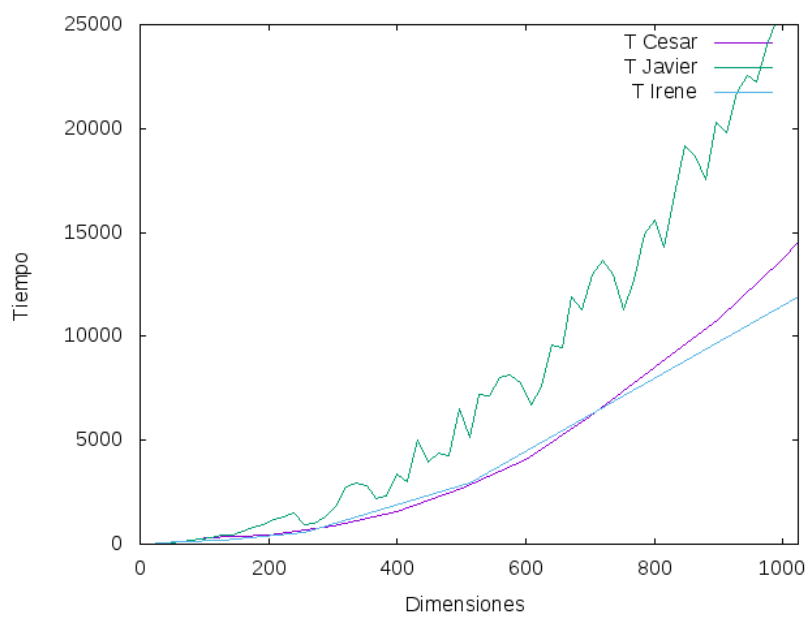


Figure 1.4: Traspuestas

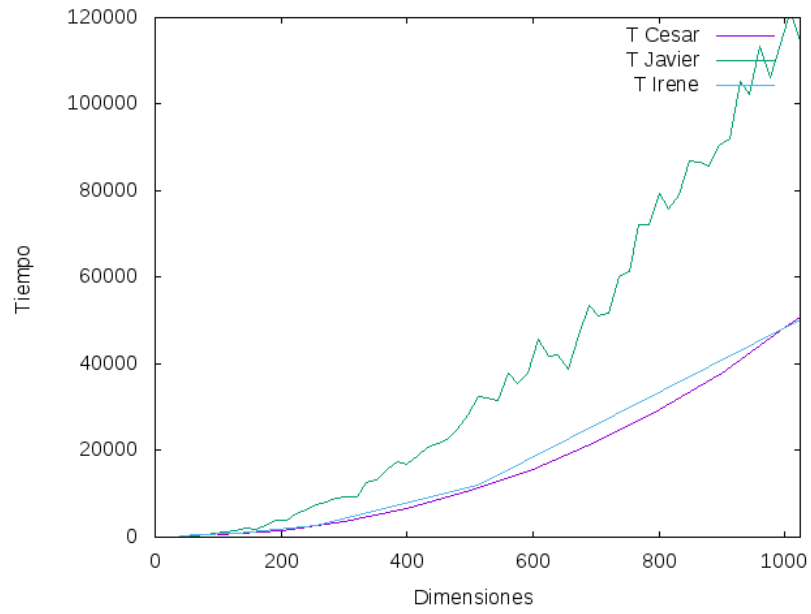


Figure 1.5: Traspuestas DyV

## 1.5 Análisis híbrido

### 1.5.1 Análisis híbrido Traspuesta

La función con la que la ajustamos es

$$f(x) = a_0x^2 + a_1x + a_2$$

Aquí los valores de las constantes tras el ajuste

```

gnuplot> f(x)=a0*x*x+a1*x+a2
gnuplot> fit f(x) 't.dat' via a0,a1,a2
iter    chisq    delta/lim    lambda    a0          a1          a2
0 2.9236237362e+14 0.00e+00 2.89e+06 1.000000e+00 1.000000e+00 1.000000e+00
1 2.1356929459e+11 -1.37e+08 2.89e+05 3.987623e-02 9.997485e-01 9.999999e-01
2 1.0240852519e+07 -2.09e+09 2.89e+04 1.321354e-02 9.997081e-01 9.999999e-01
3 1.0205792796e+07 -3.44e+02 2.89e+03 1.320701e-02 9.963715e-01 9.999946e-01
4 8.8191656301e+06 -1.57e+04 2.89e+02 1.327861e-02 7.226418e-01 9.995395e-01
5 5.9705910108e+06 -4.77e+04 2.89e+01 1.359134e-02 -4.729012e-01 9.875622e-01
6 5.9633501785e+06 -1.21e+02 2.89e+00 1.360535e-02 -5.261527e-01 -4.801685e-02
7 5.8904749951e+06 -1.24e+03 2.89e-01 1.359101e-02 -4.556357e-01 -5.513094e+01
8 5.8617203282e+06 -4.91e+02 2.89e-02 1.357488e-02 -3.763670e-01 -1.170282e+02
9 5.8617166969e+06 -6.20e-02 2.89e-03 1.357470e-02 -3.754663e-01 -1.177316e+02
iter    chisq    delta/lim    lambda    a0          a1          a2
After 9 iterations the fit converged.
final sum of squares of residuals : 5.86172e+06
rel. change during last iteration : -6.19508e-07

degrees of freedom (FIT_NDF) : 9
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 807.033
variance of residuals (reduced chisquare) = WSSR/ndf : 651302

Final set of parameters          Asymptotic Standard Error
=====
a0 = 0.0135747 +/- 0.0001805 (1.33%)
a1 = -0.375466 +/- 0.7087 (188.7%)
a2 = -117.732 +/- 297.9 (253%)

correlation matrix of the fit parameters:
a0 a1 a2
a0 1.000
a1 -0.961 1.000
a2 0.430 -0.538 1.000

```

Figure 1.6: Fit traspuesta sencilla

Y la representación gráfica

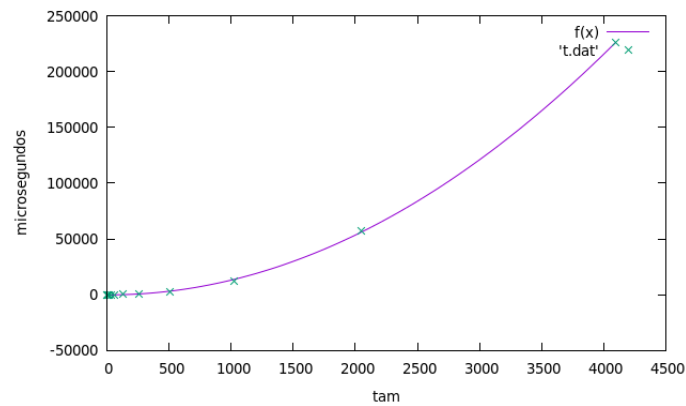


Figure 1.7: Híbrida traspuesta sencilla

### 1.5.2 Análisis híbrido Traspuesta Divide y Veceras

Usamos la misma función que para el algoritmo sencillo

$$f(x) = a0x^2 + a1x + a2$$

Aquí los valores de las constantes tras el ajuste. Se ajustan mejor que el algoritmo sencillo, los porcentajes de error son mas pequeños.

```
gnuplot> f(x)=a0*x*x+a1*x+a2
gnuplot> fit f(x) 'tdyv.dat' via a0,a1,a2
iter   chisq      delta/lim  lambda  a0          a1          a2
0 2.6890476533e+14  0.00e+00  2.89e+06  1.000000e+00  1.000000e+00  1.000000e+00
1 1.9653360154e+11  -1.37e+08  2.89e+05  7.919928e-02  9.997575e-01  9.999999e-01
2 1.0946375031e+08  -1.79e+08  2.89e+04  5.362863e-02  9.995845e-01  9.999997e-01
3 1.0898866604e+08  -4.36e+02  2.89e+03  5.362587e-02  9.829881e-01  9.999814e-01
4 7.4683042497e+07  -4.59e+04  2.89e+02  5.398201e-02  -3.785351e-01  9.985282e-01
5 4.2086494100e+06  -1.67e+06  2.89e+01  5.553754e-02  -6.325229e+00  1.019951e+00
6 4.0600719179e+06  -3.66e+03  2.89e+00  5.560930e-02  -6.600383e+00  3.898076e+00
7 3.4969580658e+06  -1.61e+04  2.89e-01  5.564922e-02  -6.796597e+00  1.570157e+02
8 3.2747679930e+06  -6.78e+03  2.89e-02  5.569404e-02  -7.016945e+00  3.290756e+02
9 3.2747399330e+06  -8.57e-01  2.89e-03  5.569455e-02  -7.019449e+00  3.310308e+02
iter   chisq      delta/lim  lambda  a0          a1          a2

After 9 iterations the fit converged.
final sum of squares of residuals : 3.27474e+06
rel. change during last iteration : -8.56863e-06

degrees of freedom (FIT_NDF) : 9
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 603.208
variance of residuals (reduced chisquare) = WSSR/ndf : 363860

Final set of parameters          Asymptotic Standard Error
=====
a0 = 0.0556945 +/- 0.0001349 (0.2423%)
a1 = -7.01945 +/- 0.5297 (7.546%)
a2 = 331.031 +/- 222.7 (67.27%)

correlation matrix of the fit parameters:
a0      a1      a2
a0      1.000
a1      -0.961  1.000
a2      0.430 -0.538  1.000
```

Figure 1.8: Fit traspuesta Divide y Vencerás

Y la representación gráfica

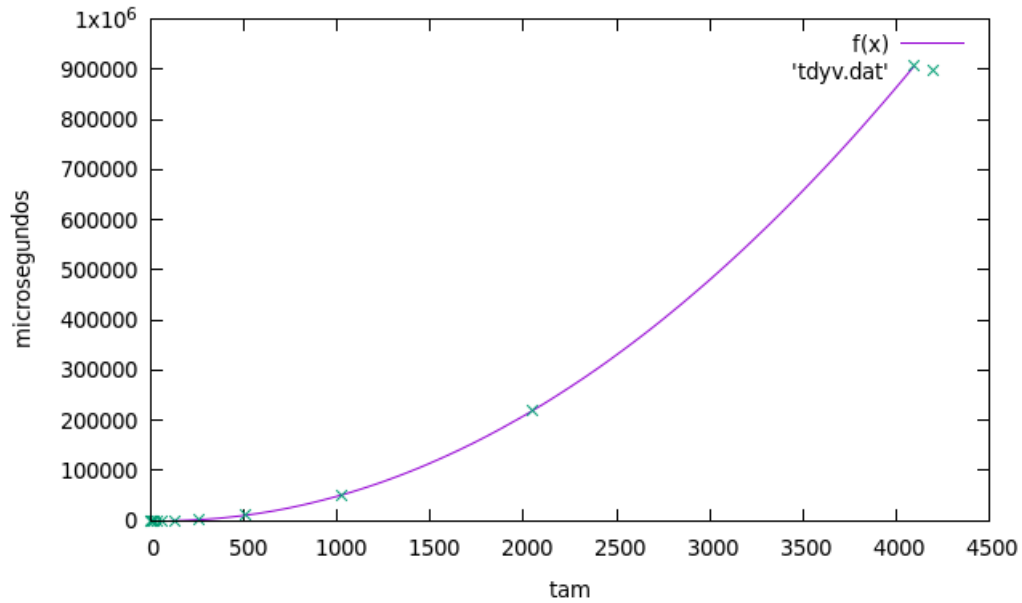


Figure 1.9: Híbrida traspuesta Divide y Vencerás

## 1.6 Ejemplo de ejecución

En ambos ejemplos la matriz sera de 8x8

### 1.6.1 Traspuesta sencilla

En primer lugar se inicializa la matriz con enteros aleatorios

```
irene@DellXPS:~/Escritorio$ ./traspuesta b 8

Esta es la matriz inicial:
3 6 7 5 3 5 6 2
9 1 2 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
```

Figure 1.10: Matriz inicial

Y ya se llama a la función traspuesta que empieza a intercambiar elementos de la diagonal superior.

Comienza cambiando el de la posición  $i=0$   $j=1$  con el de la posición  $i=1$   $j=0$  Así

sucesivamente con los de la misma fila 0 y prosigue con las siguientes de igual manera

```
Ejecutando Traspuesta para tam. caso: 8
Intercambio el elemento a_01 por el a_10
3 9 7 5 3 5 6 2
6 1 2 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
Intercambio el elemento a_02 por el a_20
3 9 0 5 3 5 6 2
6 1 2 7 0 9 3 6
7 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
Intercambio el elemento a_03 por el a_30
3 9 0 2 3 5 6 2
6 1 2 7 0 9 3 6
7 6 2 6 1 8 7 9
5 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
Intercambio el elemento a_04 por el a_40
3 9 0 2 2 5 6 2
6 1 2 7 0 9 3 6
7 6 2 6 1 8 7 9
5 0 2 3 7 5 9 2
3 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
Intercambio el elemento a_05 por el a_50
3 9 0 2 2 9 6 2
6 1 2 7 0 9 3 6
7 6 2 6 1 8 7 9
5 0 2 3 7 5 9 2
3 8 9 7 3 6 1 2
5 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7
```

Figure 1.11: Intercambio elementos fila 0

```

Intercambio el elemento a_45 por el a_54
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 2
5 9 8 5 6 7 8 4
6 3 7 9 1 0 6 3
2 6 9 2 5 5 4 7
Intercambio el elemento a_46 por el a_64
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 2
5 9 8 5 6 7 8 4
6 3 7 9 1 0 6 3
2 6 9 2 5 5 4 7
Intercambio el elemento a_47 por el a_74
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 8 4
6 3 7 9 1 0 6 3
2 6 9 2 2 5 4 7

```

Figure 1.12: Intercambio elementos fila 4

Así hasta llegar a la fila 7.

Finalmente se nos queda esta matriz, que coincide con la traspuesta de la original:

```

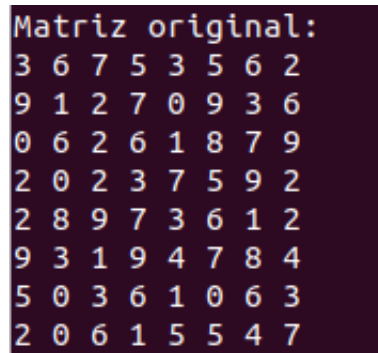
Matriz resultante:
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 0 5
6 3 7 9 1 8 6 4
2 6 9 2 2 4 3 7

```

Figure 1.13: Matriz traspuesta

### 1.6.2 Traspuesta Divide y Venceras

Partimos de la siguiente matriz



|                  |   |   |   |   |   |   |   |
|------------------|---|---|---|---|---|---|---|
| Matriz original: |   |   |   |   |   |   |   |
| 3                | 6 | 7 | 5 | 3 | 5 | 6 | 2 |
| 9                | 1 | 2 | 7 | 0 | 9 | 3 | 6 |
| 0                | 6 | 2 | 6 | 1 | 8 | 7 | 9 |
| 2                | 0 | 2 | 3 | 7 | 5 | 9 | 2 |
| 2                | 8 | 9 | 7 | 3 | 6 | 1 | 2 |
| 9                | 3 | 1 | 9 | 4 | 7 | 8 | 4 |
| 5                | 0 | 3 | 6 | 1 | 0 | 6 | 3 |
| 2                | 0 | 6 | 1 | 5 | 5 | 4 | 7 |

Figure 1.14: Matriz original

A continuación se llama a la función `traspuestaDYV` que es la que la subdivide en cuadrantes y va calculando la traspuesta de estos

Como se puede ver en la siguiente figura si nos fijamos, primero se centra en el primer cuadrante. Lo subdivide en cuatro y hace la traspuesta de cada uno.



```

Ejecutando Traspuesta para tam. caso: 8

3 9 7 5 3 5 6 2
6 1 2 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7

3 9 7 2 3 5 6 2
6 1 5 7 0 9 3 6
0 6 2 6 1 8 7 9
2 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7

3 9 7 2 3 5 6 2
6 1 5 7 0 9 3 6
0 2 2 6 1 8 7 9
6 0 2 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7

3 9 7 2 3 5 6 2
6 1 5 7 0 9 3 6
0 2 2 2 1 8 7 9
6 0 6 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7

```

Figure 1.15: Primeros intercambios

A continuación hace lo de cambiar el cuadrante 2 y 3 de diagonal (todo esto repito centrado en lo que seria el primer cuadrante, lo que intercambia son los subcuadrantes de este)

```

3 9 0 2 3 5 6 2
6 1 6 0 0 9 3 6
7 2 2 2 1 8 7 9
5 7 6 3 7 5 9 2
2 8 9 7 3 6 1 2
9 3 1 9 4 7 8 4
5 0 3 6 1 0 6 3
2 0 6 1 5 5 4 7

```

Figure 1.16: Intercambio diagonal primer cuadrante

Y repite este mismo proceso con los demás cuadrantes

Cuando ya tiene todos traspuestos, hace el cambio con los cuadrantes de la primera división en 4 de la matriz

```

3 9 0 2 3 0 1 7
6 1 6 0 5 9 8 5
7 2 2 2 6 3 7 9
5 7 6 3 2 6 9 2
2 9 5 2 3 4 1 5
8 3 0 0 6 7 0 5
9 1 3 6 1 8 6 4
7 9 6 1 2 4 3 7

3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 0 5
6 3 7 9 1 8 6 4
2 6 9 2 2 4 3 7

```

Figure 1.17: Intercambio diagonal cuadrantes grandes

Y finalmente aparece la traspuesta

```

Matriz resultante:
3 9 0 2 2 9 5 2
6 1 6 0 8 3 0 0
7 2 2 2 9 1 3 6
5 7 6 3 7 9 6 1
3 0 1 7 3 4 1 5
5 9 8 5 6 7 0 5
6 3 7 9 1 8 6 4
2 6 9 2 2 4 3 7

```

Figure 1.18: Matriz traspuesta

## 2 Problema asignado: Serie unimodal de números

### 2.1 Enunciado

Sea un vector  $v$  de números de tamaño  $n$ , todos distintos, de forma que existe un índice  $p$  (que no es el primero ni el último) tal que a la izquierda de  $p$  los números están ordenados de forma creciente y a la derecha de  $p$  están ordenados de forma decreciente. Es decir

$$\forall i, j \geq p, i < j \rightarrow v[i] > v[j]$$

De forma que el máximo se encuentra en la posición  $p$ . Tenemos que diseñar el algoritmo normal y otro que use "divide y vencerás" y ver la complejidad de ambos algoritmos y la diferencia de eficiencia. También tenemos que ver a partir de que umbral (tamaño de elementos) es recomendable usar uno u otro

### 2.2 Algoritmos

#### 2.2.1 Unimodal sencillo

Para la versión sencilla hemos implementado el siguiente algoritmo (*unimodal.cpp*):

```
double uniforme() {
    double u;
    u = (double) rand();
    u = u / (double)(RAND_MAX + 1.0);
    return u;
}

int unimodal(int *T, int n){
    for(int i= 0; i < n; i ++){
        if(T[i]> T[i+1] && T[i]> T[i-1]){
            return i;
        }
    }
}

int main(int argc, char * argv[]){
    int n, i, argumento;

    if (argc <= 2) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n";
        cerr<<argv[0]<<" NombreFicheroSalida tamCaso1 tamCaso2 ... tamCasoN\n";
        return 0;
    }
}
```

```

// Pasamos por cada tam de caso
for (argumento= 2; argumento<argc; argumento++) {

    // Cogemos el tamaño del caso
    n= atoi(argv[argumento]);

    int * T = new int[n];
    assert(T);
    srand(time(0));
    double u=uniforme();
    int p=1+(int)((n-2)*u);
    T[p]=n-1;
    for (int i=0; i<p; i++) T[i]=i;
    for (int i=p+1; i<n; i++) T[i]=n-1-i+p;
    //for (int j=0; j<n; j++) {cout << T[j] << " ";}

    cerr<<"Ejecutando Unimodal para tam. caso: "<<n<<endl;
    cout<<"\nPosicion "<<unimodal(T,n)<<endl;
}
}

```

Simplemente el algoritmo busca el mayor elemento, es decir, la posición del vector que contiene al número que es a su vez mayor que el número que le precede y que el número que va después. Por lo tanto este algoritmo lo que hace es ir recorriendo el vector elemento a elemento y va comprobando en cada posición si se cumple la condición que acabamos de explicar. Si es así, es el elemento que buscamos y devuelve la posición donde se encuentra.

### 2.2.2 Unimodal Divide y Vencerás

La versión del código completa esta en (*unimodaldyv.cpp*)

```

double uniforme(){
    double u;
    u = (double) rand();
    u = u/(double)(RAND_MAX+1.0);
    return u;
}

int unimodal(int *v, int ini, int fin){
    int med = (fin + ini)/2;
    if(v[med] > v[med-1] && v[med] > v[med + 1])
        return med;
}

```

```

    else if(v[med] > v[med-1])
        unimodal(v, med, fin);
    else
        unimodal(v, ini, med);
}

int main(int argc, char * argv[]) {
    int n, i, argumento, x;

    if (argc <= 2) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n";
        cerr<<argv[0]<<" NombreFicheroSalida tamCaso1 tamCaso2 ... tamCasoN\n";
        return 0;
    }

    // Pasamos por cada tam de caso
    for (argumento= 2; argumento<argc; argumento++) {
        // Cogemos el tamaño del caso
        n= atoi(argv[argumento]);

        int * T = new int[n];
        assert(T);
        srand(time(0));
        double u=uniforme();
        int p=1+(int)((n-2)*u);
        T[p]=n-1;
        for (int i=0; i<p; i++) T[i]=i;
        for (int i=p+1; i<n; i++) T[i]=n-1-i+p;

        cerr << "Ejecutando Unimodal para tam. caso: " << n << endl;
        cout << "\nPosicion " << unimodal(T,0,n-1) << endl;

    }
}

```

Aquí lo que hacemos es dividir el problema en dos recursivamente. Al llamar a la función 'unimodal' lo primero que hace es calcular el elemento que está en la mitad del vector para ver si el elemento que hace de pico está a la derecha o a la izquierda de este.

Si el número que está en la posición 'med' es menor que el de la posición siguiente, tenemos que buscar el elemento máximo a la derecha. Si el número que está en la posición 'med' es mayor que el de la posición anterior, tenemos que buscar por la izquierda. Solo habremos dado con el elemento buscado cuando el elemento de en

medio sea el mayor.

## 2.3 Complejidad

### 2.3.1 Unimodal sencillo

Una vez se ha rellenado el vector con ayuda del generador que nos proporcionaba ya la práctica, nos enfrentamos al bucle for que recorre el vector elemento a elemento en busca de ese elemento que marca la diferencia de orden ascendente a descendente.

$$\sum_0^n (max(a, 0)) \text{ siendo 'a' las operaciones elementales del 'if'}$$

El vector se recorre prácticamente n veces. El orden de eficiencia será lineal: **O(n)**

### 2.3.2 Unimodal Divide y vencerás

Más complicado de estudiar que el sencillo ya que aparecen llamadas recursivas. Como ya hemos explicado, el problema se va dividiendo en dos hasta que se encuentra el elemento, por lo tanto el tiempo de ejecución es  $T(n) = T(n/2) + a$ . Hacemos el cambio de variable  $t_k = T(2^k)$  Y nos queda  $t_k = t_{k-1} + a$

Calculamos la ecuación característica. Seguimos los pasos de los apuntes del capítulo 3 de la asignatura.

Partimos de  $t_k = t_{k-1} + a$

Reemplazamos k por k+1 en la ecuación de recurrencia original y nos queda

$$t_{k+1} = t_k + a$$

Realizamos la siguiente resta:  $(t_{k+1} - t_k = a) - (t_k - t_{k-1} = a) = t_{k+1} - 2t_k + t_{k-1} = 0$

Se nos queda la expresión  $x^2 - 2x + 1 = 0 \rightarrow (x - 1)^2 = 0$

Luego  $t_k = c_1 k + c_2$  y como  $n = 2^k$

$$T(n) = c_1 \log n + c_2$$

El orden de complejidad es **O(log n)**

### Umbral

Siempre el divide y vencerás va a ser más eficiente que el unimodal sencillo, el orden de eficiencia es bastante más rápido sea el vector del tamaño que sea. Incluso en el caso más pequeño (n=3), el unimodal divide y vencerás sigue siendo más eficiente

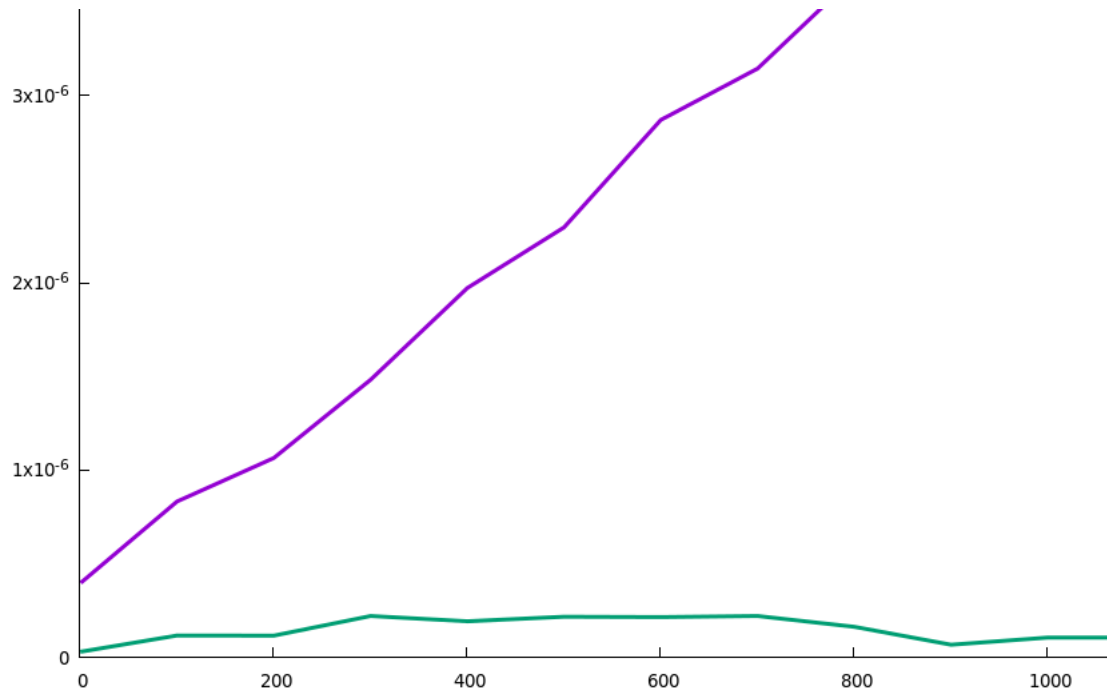


Figure 2.1: Umbral empírico unimodal

## 2.4 Análisis empírico

### 2.4.1 Comparación algoritmo sencillo con Divide y vencerás

Se ve la concordancia de la función unimodal con la eficiencia  $O(n)$  y la del algoritmo divide y vencerás con eficiencia logarítmica. Se confirma empíricamente que siempre va a ser mas eficiente la segunda que la primera.

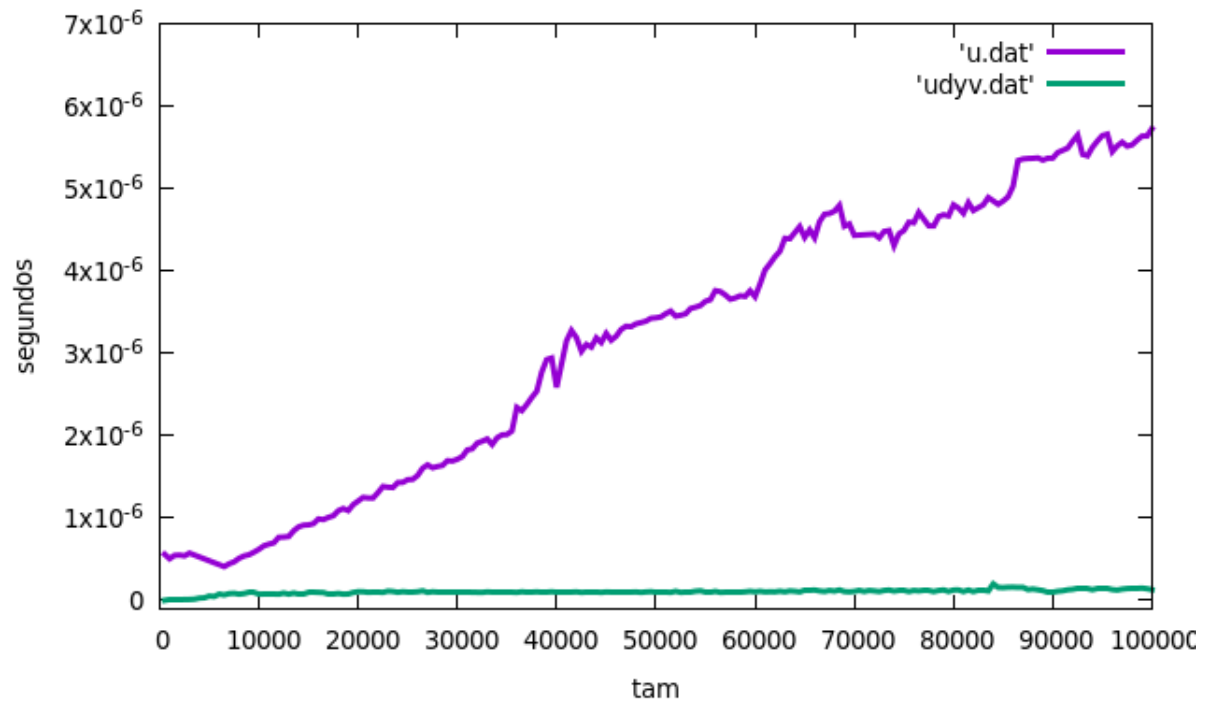


Figure 2.2: Unimodal sencilla y DyV de Irene

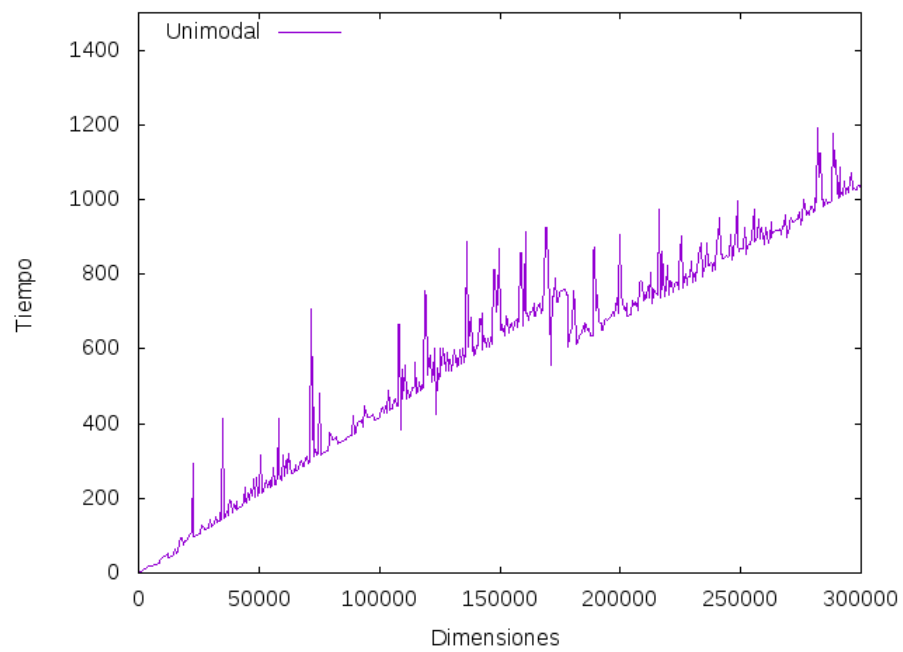


Figure 2.3: Unimodal sencilla de Javier



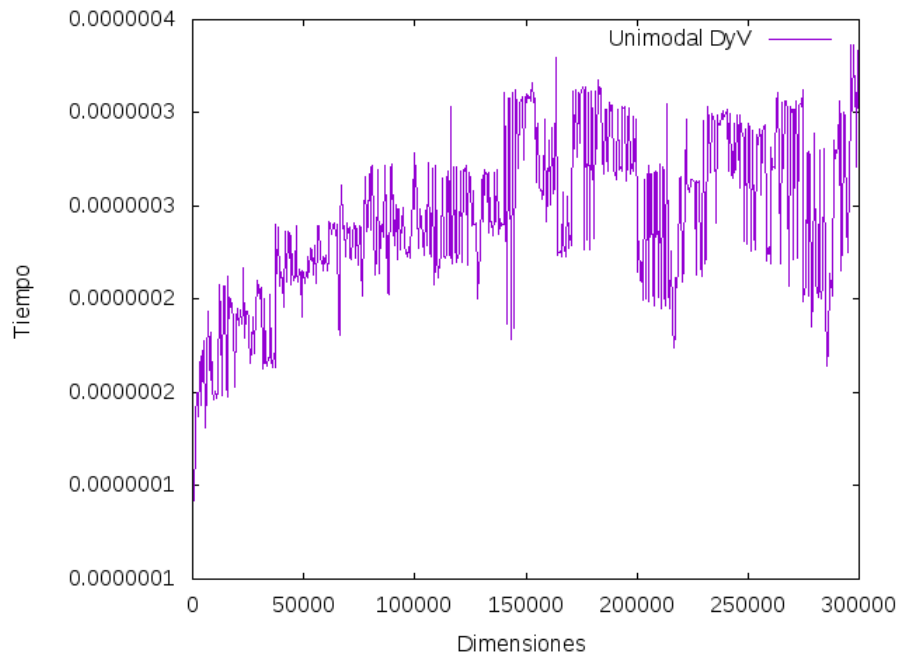


Figure 2.4: Unimodal DyV de Javier

Cabe destacar que Javier no ha podido incluir ambos resultados en una misma gráfica pues la diferencia es tal que no se ven las dos a la vez.

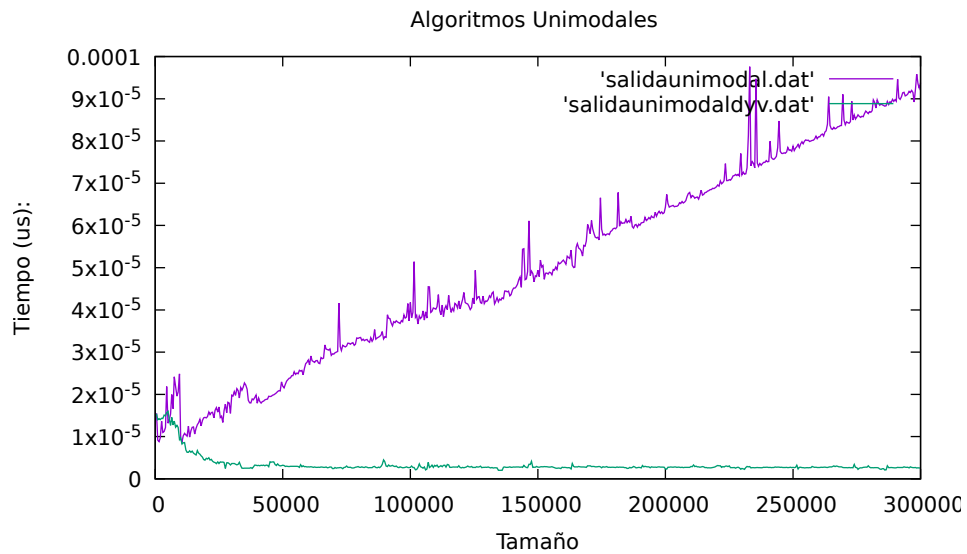


Figure 2.5: Unimodal sencilla y DyV de César

### 2.4.2 Comparación entre ordenadores de la empírica

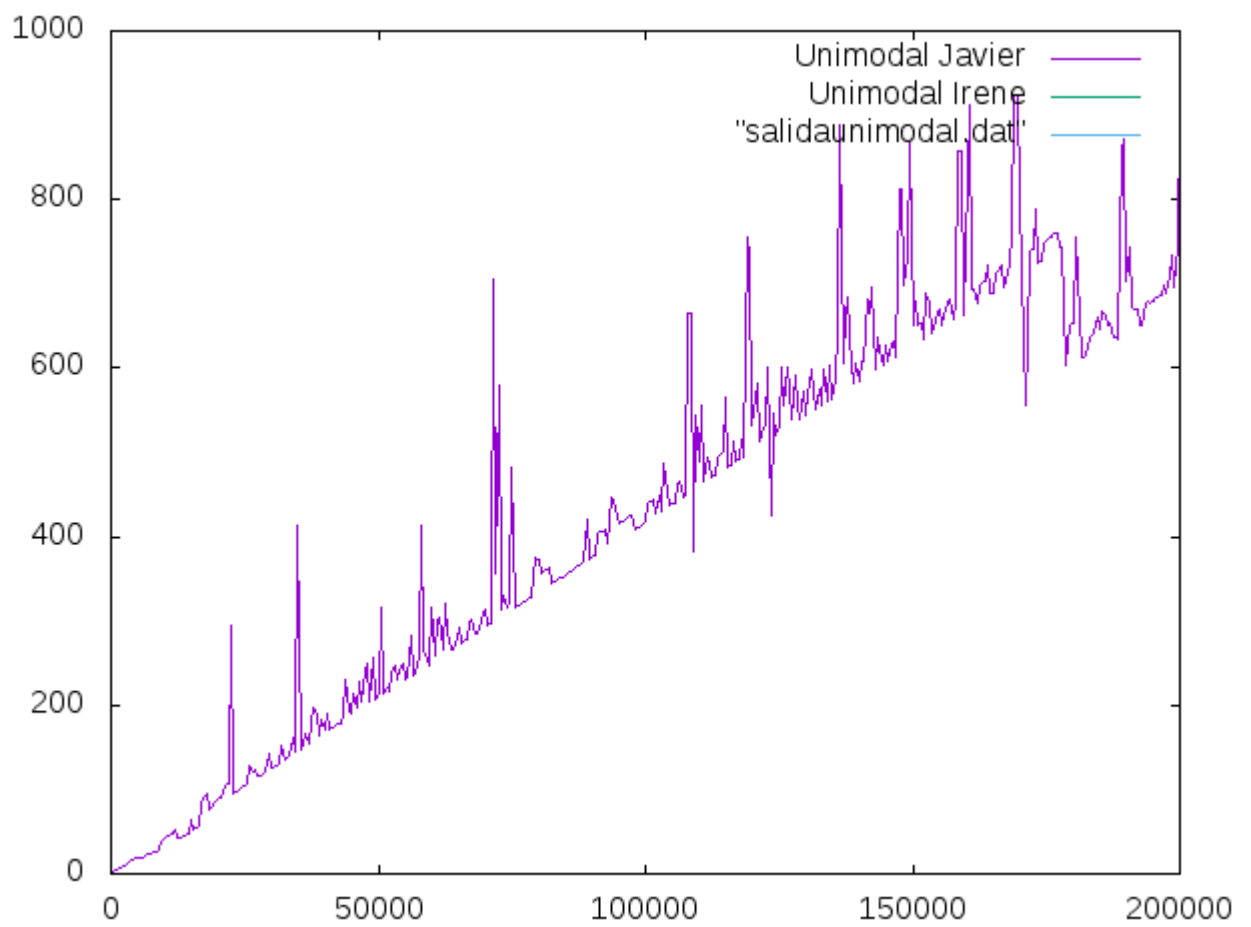


Figure 2.6: Unimodales

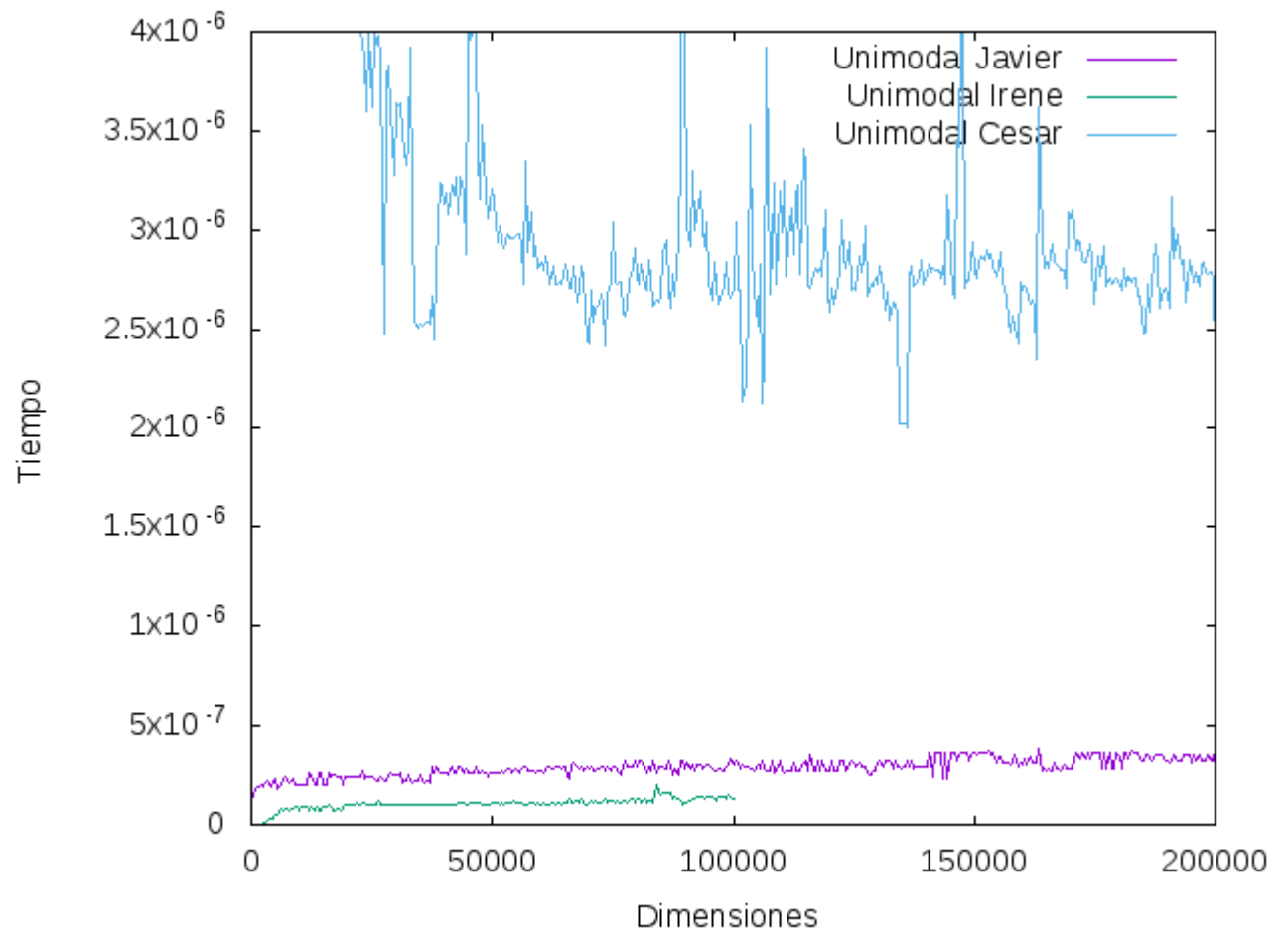


Figure 2.7: Unimodales DyV

## 2.5 Análisis híbrido

### 2.5.1 Análisis híbrido Unimodal

La función con la que la ajustamos es

$$f(x) = a_0x + a_1$$

Aquí los valores de las constantes tras el ajuste

```
gnuplot> f(x)=a0*x+a1
gnuplot> fit f(x) 'u.dat' via a0,a1
iter   chisq      delta/lim  lambda  a0          a1
0 6.2426232011e+11  0.00e+00  4.10e+04  1.000000e+00  1.000000e+00
1 4.4869758470e+06  -1.39e+10  4.10e+03  2.665932e-03  9.999849e-01
2 4.4153003419e+01  -1.02e+10  4.10e+02  -1.500185e-05  9.999823e-01
3 4.4126538135e+01  -6.00e+01  4.10e+01  -1.506995e-05  9.997192e-01
4 4.1893063491e+01  -5.33e+03  4.10e+00  -1.468361e-05  9.740902e-01
5 3.1774023378e+00  -1.22e+06  4.10e-01  -4.043835e-06  2.682653e-01
6 4.5552434108e-05  -6.98e+09  4.10e-02  -1.525319e-08  1.015963e-03
7 1.2481202913e-11  -3.65e+11  4.10e-03  5.778892e-11  2.585452e-07
8 1.2415404907e-11  -5.30e+02  4.10e-04  5.837085e-11  2.199410e-07
9 1.2415404907e-11  -1.30e-10  4.10e-05  5.837085e-11  2.199410e-07
iter   chisq      delta/lim  lambda  a0          a1

After 9 iterations the fit converged.
final sum of squares of residuals : 1.24154e-11
rel. change during last iteration : -1.30128e-15

degrees of freedom (FIT_NDF) : 184
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 2.5976e-07
variance of residuals (reduced chisquare) = WSSR/ndf : 6.7475e-14

Final set of parameters          Asymptotic Standard Error
=====
a0 = 5.83708e-11 +/- 6.748e-13 (1.156%)
a1 = 2.19941e-07 +/- 3.909e-08 (17.77%)

correlation matrix of the fit parameters:
      a0      a1
a0      1.000
a1     -0.873  1.000
```

Figure 2.8: Fit unimodal sencillo

Y la representación gráfica

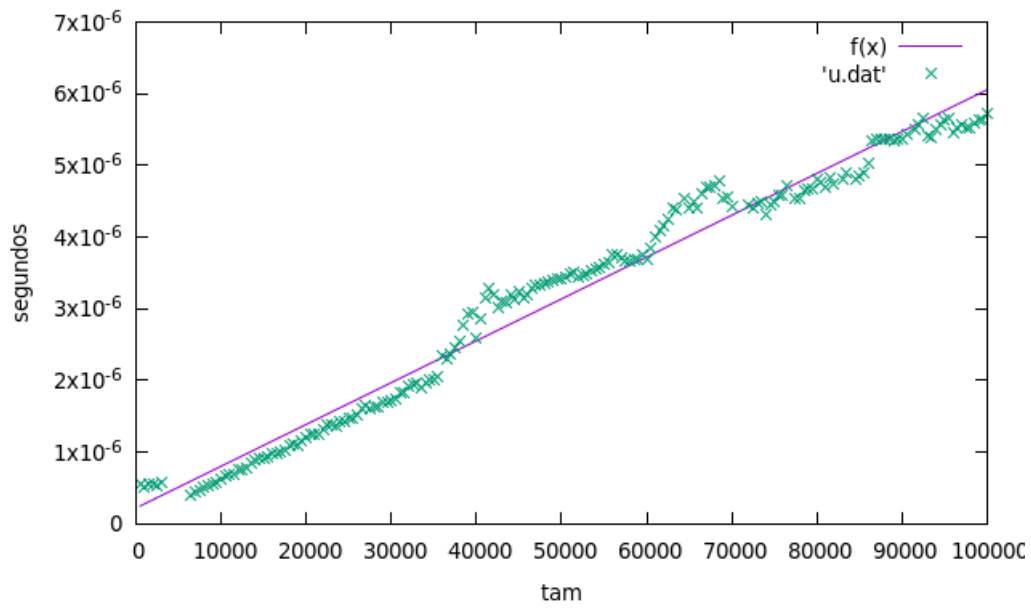


Figure 2.9: Híbrida unimodal sencillo

### 2.5.2 Análisis híbrido Unimodal Divide y Vencerás

Usamos función logarítmica

$$f(x) = a0\log(x) + a1$$

Valores de ajuste:

```

gnuplot> f(x)=a0*log(x)+a1
gnuplot> fit f(x) 'udyv.dat' via a0,a1
iter   chisq      delta/lim  lambda   a0          a1
  0  2.8720576067e-12   0.00e+00  1.56e-07  5.837085e-11  2.199410e-07
  1  1.5536496631e-13  -1.75e+06  1.56e-08  5.893526e-11  1.036773e-07
  2  1.5474930689e-13  -3.98e+02  1.56e-09  1.238340e-10  1.027033e-07
  3  1.1195934205e-13  -3.82e+04  1.56e-10  5.293502e-09  4.826267e-08
  4  3.7773535682e-14  -1.96e+05  1.56e-11  2.477385e-08  -1.568804e-07
  5  3.7668046970e-14  -2.80e+02  1.56e-12  2.553636e-08  -1.649101e-07
  6  3.7668046954e-14  -4.29e-05  1.56e-13  2.553665e-08  -1.649133e-07
iter   chisq      delta/lim  lambda   a0          a1

After 6 iterations the fit converged.
final sum of squares of residuals : 3.7668e-14
rel. change during last iteration : -4.29066e-10

degrees of freedom (FIT_NDF) : 198
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 1.37928e-08
variance of residuals (reduced chisquare) = WSSR/ndf : 1.90243e-16

Final set of parameters      Asymptotic Standard Error
=====
a0 = 2.55367e-08             +/- 1.024e-09 (4.011%)
a1 = -1.64913e-07           +/- 1.083e-08 (6.568%)

correlation matrix of the fit parameters:
      a0      a1
a0    1.000
a1   -0.996  1.000

```

Figure 2.10: Fit unimodal Divide y Vencerás

Y la representación gráfica

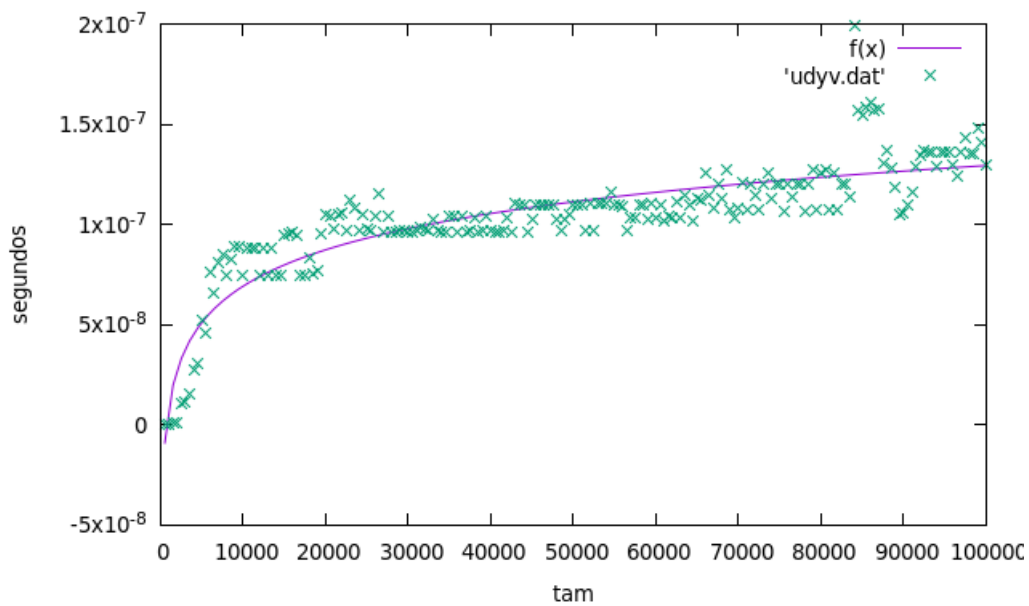


Figure 2.11: Híbrida Unimodal Divide y Vencerás

## 2.6 Ejemplo de ejecución

### 2.6.1 Unimodal sencilla

Inicializamos el vector con un tamaño de  $n = 1000$ . El algoritmo se basa en recorrer secuencialmente en  $i = 0, 1, \dots, n$  posiciones del vector hasta encontrar un elemento que cumpla los requisitos de  $v[i] > v[i-1]$  y  $v[i] > v[i+1]$ .

```
Ejecutando Unimodal para tam. caso: 1000  
Posicion 846 : v[845] < v[846]y v[847] < v[846]  
Tiempo de ejec.: 9.448e-06para tam. caso: 1000
```

Figure 2.12: Ejemplo de ejecución

### 2.6.2 Unimodal Divide y Vencerás

Inicializamos también el vector con un tamaño de  $n = 1000$ . En este caso se llama recursivamente en 2 funciones a la función unimodal que divide el vector en 2 subvectores y elige uno en el que  $v[i+1] > v[i]$  y se sigue buscando hasta que se cumpla que  $v[i] > v[i-1]$  y  $v[i] > v[i+1]$ .

```
Ejecutando Unimodal para tam. caso: 1000  
Posicion 930 : v[929] < v[930]y v[931] < v[930]  
Tiempo de ejec.: 5.8728e-05para tam. caso: 1000
```

Figure 2.13: Ejemplo de ejecución