Práctica 2

Algoritmos divide y vencerás

Manuel Jesús Núñez Ruiz
Miguel Pedregosa Pérez

Índice

1. Algoritmo de trasposición de matriz

1.1. Traspuesta de una matriz

Sea A una matriz con *m* filas y *n* columnas. La matriz traspuesta, denotada con A^t, está dada por:

$$\left(A^t\right)_{ij} = A_{ji}, 1 \leq i \leq n, 1 \leq j \leq m$$

En donde cada elemento a_{ij} de la matriz original *A* se convertirá en el elemento a_{ij} de la matriz traspuesta A^t. En otras palabras, consiste en cambiar las filas de la matriz por columnas. Un ejemplo de matriz traspuesta:

$$\begin{bmatrix} 0 & 0 & 4 \\ 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 3 & 2 \\ 0 & 2 & 3 \\ 0 & 3 & 4 \\ 3 & 3 & 1 \end{bmatrix}^t = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 3 & 2 & 3 & 3 \\ 4 & 4 & 0 & 2 & 3 & 4 & 1 \end{bmatrix}$$

1.2. Versión sencilla del algoritmo

Este algoritmo es bastante sencillo, lo único que hace es recorrer toda la matriz convirtiendo cada elemento a_{ij} de la matriz *A* en a_{ij} en la matriz traspuesta A^t.

```
11 void traspuesta(int** &matriz, int &n, int &m){
12     int** traspuesta;
13     int aux;
14
15     traspuesta=new int*[m];
16     for (int i=0; i<m; ++i){
17         traspuesta[i]=new int[n];
18         for (int j=0; j<n; ++j)
19             traspuesta[i][j]=matriz[j][i];
20     }
21     for (int i=0; i<n; ++i)
22         delete[] matriz[i];
23     delete[] matriz;
24
25     matriz=traspuesta;
26     aux=n;
27     n=m;
28     m=aux;
29 }
```

1.2.1. Eficiencia teórica

Como podemos observar en la imagen de arriba las líneas 12, 13, 15, 17, 19, 22, 23, 25, 26, 27 y 28 son operaciones elementales por lo que tienen eficiencia **O(1)**. El bucle *for* de la línea 16 tiene eficiencia **O(m)** y además tiene anidado otro bucle *for* con eficiencia **O(n)**, por lo que la eficiencia de este bloque es **O(n·m)**. Por último hay otro bucle *for* en la línea 21 con eficiencia **O(n)**.

Para calcular la eficiencia en el peor de los casos del algoritmo aplicamos la regla del máximo:

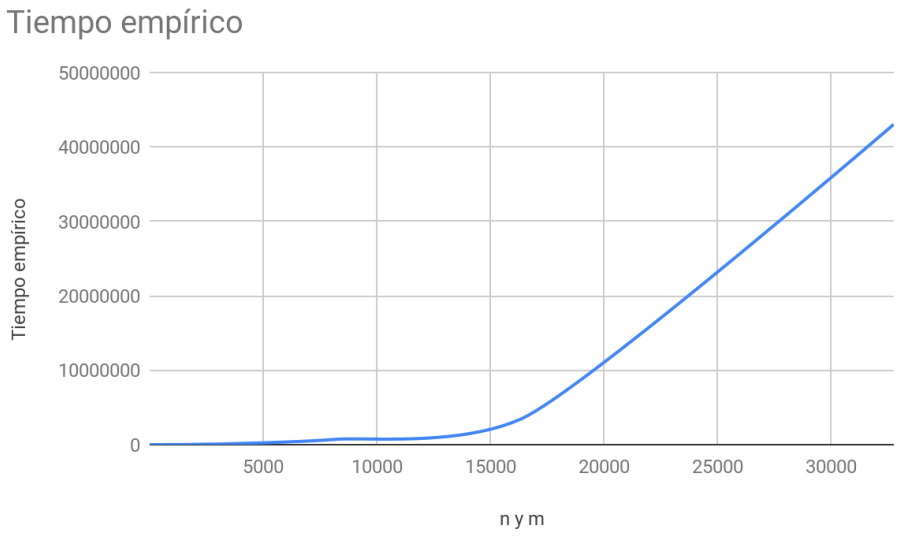
$$Max\{O(1), O(n{\cdot}m), O(n)\} = O(n{\cdot}m)$$

La eficiencia para el mejor de los casos de este algoritmo es **Ω(n·m)**, porque los bucles se recorrerán hasta el final pase lo que pase.

1.2.2. Eficiencia empírica

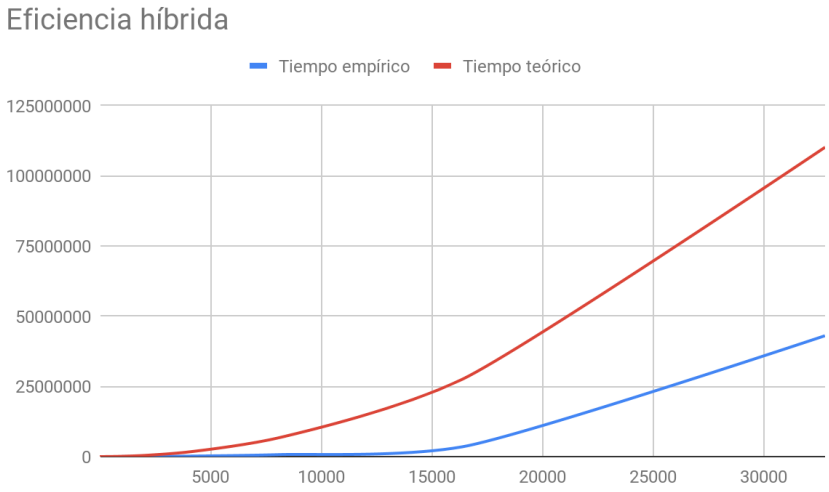
Aquí podemos ver los datos empíricos obtenidos, y su gráfica:

n y m	Tiempo empírico
2	4
4	6
8	2
16	3
32	8
64	23
128	86
256	383
512	1686
1024	7221
2048	36515
4096	166543
8192	713044
16384	3485055
32768	43023303



1.2.3. Eficiencia híbrida

Tras el ajuste de la función teórica, obtenemos el siguiente tiempo híbrido:



1.3. Versión Divide y Vencerás del algoritmo

Para hacer la versión Divide y Vencerás del algoritmo hemos separado la matriz en tres: diagonal principal, triángulo superior y triángulo inferior. Es decir, así:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

Podemos ver el algoritmo implementado en C++ en el Anexo I.

1.3.1. Eficiencia teórica

Para este algoritmo el peor de los casos es que la matriz sea cuadrada, ya que así todas las funciones hacen el mismo número de iteraciones del bucle (las que trasponen los triángulos), ya que si no fuesen cuadradas, las iteraciones que hace una función sería menor que la otra. Por lo que para este análisis suponemos que $n=m$.

Vayamos por partes:

Empecemos por la función que “TrasponDiagonal” la cual recorre la diagonal entera, por lo que es **O(n)**.

La función “TrasponTrianguloSuperior” recorre $\frac{(n^2-n)}{2}$ elementos de la matriz, por lo que su eficiencia es **O(n^2)**. Lo mismo pasa con “TrasponTrianguloInferior”.

Ahora vayamos a la función “Traspuesta”. Empieza con un bucle de eficiencia **O(n)**, luego se llama a las funciones y por último hay otro bucle de eficiencia **O(n)**.
Aplicamos la regla del máximo:

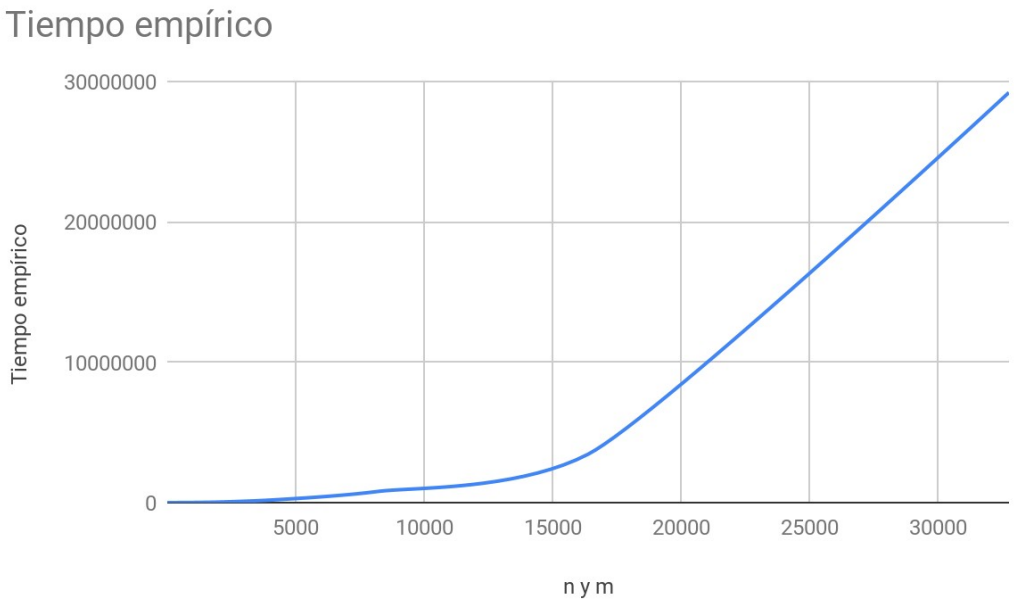
$$Max\{O(1),O(n),O(n),O(n^2),O(n^2),O(n)\}=O(n^2)$$

La eficiencia en el mejor de los casos de este algoritmo es **Ω(n^2)**, ya que la matriz se debe de recorrer entera sí o sí.

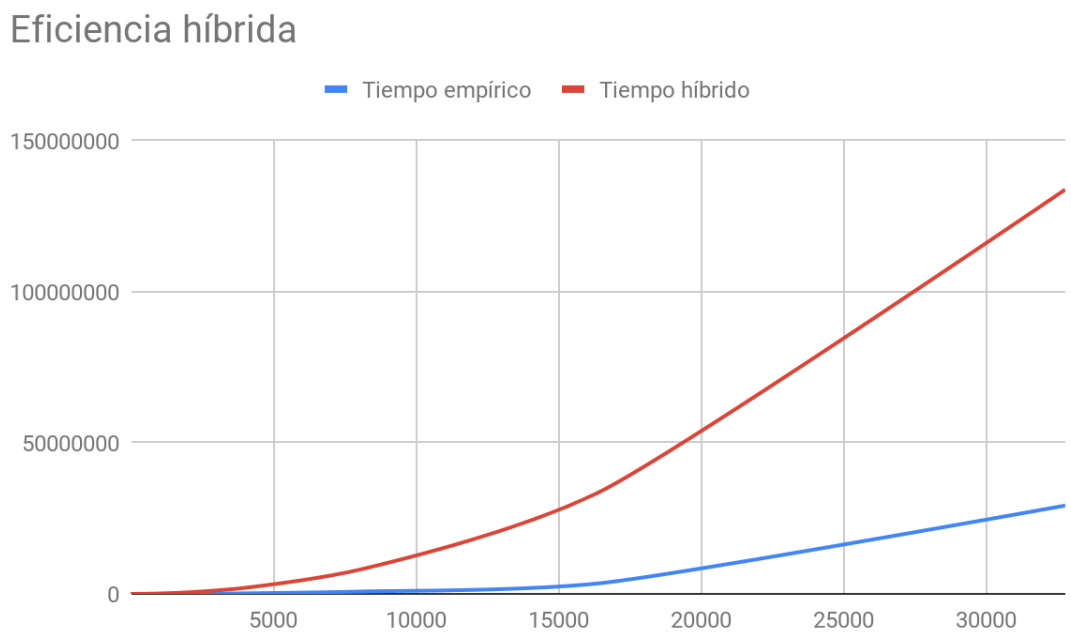
1.3.2. Eficiencia empírica

Tras el ajuste de la función teórica, obtenemos:

n y m	Tiempo empírico
2	5
4	5
8	6
16	10
32	25
64	72
128	423
256	1003
512	2407
1024	8847
2048	39901
4096	188107
8192	791992
16384	3444054
32768	29206591



1.3.3. Eficiencia híbrida



Como podemos observar en la gráfica, el tiempo empírico se mantiene por debajo del tiempo híbrido.

1.3.4. Caso de ejecución

```
migue@migue:~/Escritorio/ALG/practica2$ ./bin/traspuesta2 4 4
MATRIZ ORIGINAL
52526   56524   1185   83958
45044   23651   83644   37343
54522   28131   47929   14791
50944   53488   54249   87134

1. Se copia la diagonal
52526   0   0   0
0   23651   0   0
0   0   47929   0
0   0   0   87134

2. Se traspone el triangulo superior
52526   0   0   0
56524   23651   0   0
1185   83644   47929   0
83958   37343   14791   87134

3. Se traspone el triangulo inferior
52526   45044   54522   50944
56524   23651   28131   53488
1185   83644   47929   54249
83958   37343   14791   87134

TRASPUESTA
52526   45044   54522   50944
56524   23651   28131   53488
1185   83644   47929   54249
83958   37343   14791   87134
```

1.4. Comparación de las versiones sencilla y DyV del algoritmo

1.4.1. Comparación de la eficiencia teórica

El algoritmo de la versión sencilla tiene eficiencia **O(n·m)** y el algoritmo de la versión Divide y Vencerás tiene eficiencia **O(n^2)**. Si suponemos que *n*=*m* (matriz cuadrada), tenemos que la eficiencia del algoritmo sencillo es **O(n^2)**. Por lo que teóricamente no hemos obtenido ninguna mejora en la eficiencia al pasarlo a Divide y Vencerás. Debido a que tienen la misma eficiencia teórica no podemos hallar ningún umbral.

1.4.2. Comparación de la eficiencia empírica

Podemos ver que el tiempo empírico de la versión divide y vencerás es mejor, aunque se aprecia claramente que solo difieren por una constante multiplicativa, lo que nos hace suponer que las iteraciones de la versión Divide y Vencerás tardan una fracción del tiempo que tardan las de la versión sencilla. Esto sirve como una demostración más de que las eficiencias de las dos versiones del algoritmo son **Θ** la una de la otra.

2. Algoritmo de búsqueda sobre serie unimodal

2.1 Enunciado del problema:

Sea un vector v de números de tamaño n, todos distintos, de forma que existe un índice p (que no es ni 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 $\forall i, j \leq p, i < j \Rightarrow v[i] < v[j]$ y $\forall i, j \geq p, i < j \Rightarrow v[i] > v[j]$ (de forma que el máximo se encuentra en la posición p). Diseñe un algoritmo que permita determinar p.

2.2 Algoritmo propuesto

Dado nuestro vector v de longitud n y un número $k \in (0, n-1)$ podemos determinar si p es mayor, menor o igual a k de la siguiente manera:

i. Sean $mi, md : (0, n-1) \rightarrow \boldsymbol{B}$ definidas
   - $mi(k) = v[k] > v[k-1]$
   - $md(k) = v[k+1] > [k]$

ii. Sabemos por la geometría del problema que:
   - $mi(k) \wedge md(k) \Leftrightarrow p \in (k, n-1)$ ya que *k* está en la parte de *v* ordenada crecientemente.
   - $\neg mi(k) \wedge \neg md(k) \Leftrightarrow p \in (0, k)$ ya que *k* está en la parte de *v* ordenada decrecientemente.
   - $mi(k) \wedge \neg md(k) \Leftrightarrow p = k$ ya que *k* es el máximo del vector.

Al poder localizar p en función de k, estamos en condiciones para realizar una búsqueda equivalente a la búsqueda binaria sobre el vector, de la siguiente manera:

1) Definimos $k = \lfloor \frac{n}{2} \rfloor - 1$, es decir, como índice central del vector
2) Si p=k solución encontrada; si p<k volvemos al paso uno con la mitad izquierda del vector, es decir, con el subvector v con índices $i \in (0, k-1)$ y $tamaño = k$; si p>k volvemos al paso uno con la mitad derecha del vector, es decir, con el subvector v con índices $i \in (k+1, n-1)$ y $tamaño = n-k-1$. Tamaño pasará a ser n al volver al paso uno.

Podemos ver como este algoritmo sigue las directrices básicas de un divide y vencerás, de la siguiente manera:

- El vector se divide en tres subvectores: uno con los índices $i \in (0, k-1)$, otro con los índices $i \in (k+1, n-1)$ y otro con un único índice que es k.
- Para tratar los subvectores, hacemos el análisis sobre k para saber en cuál de los tres se encuentra p, obteniendo así un único subvector que contiene la solución de la búsqueda.
- La única operación que realizamos sobre los subvectores sin solución es el descarte, mientras que para encontrar la solución en el subvector que la contiene le aplicaremos de nuevo el algoritmo (a no ser que sea el subvector de una única posición). Así podremos realizar una implementación recursiva del algoritmo.

2.2.1 Análisis de eficiencia:

- **Caso mejor:**

La eficiencia en el caso mejor del algoritmo será **Ω(1)** y se corresponderá con el caso en el que *k* sea igual a *p* en la primera iteración.

- **Caso peor:**

La eficiencia en el caso peor será **O(log₂(n))** y se corresponderá con el caso en el que se recorran el máximo número posible de iteraciones del algoritmo. Este cálculo es el mismo que para la búsqueda binaria de la práctica uno, a diferencia de que los dos subvectores de mayor tamaño no tienen un tamaño de $\frac{n}{2}$, sino de $\frac{n}{2}-0.5$ de promedio. No obstante, despreciamos esta diferencia por lo que las cuentas serán las mismas.

- **Caso medio:**

Al igual que la búsqueda binaria, será **Θ(log₂(n)-2)**. El cálculo de esto se puede ver en la entrega de la práctica uno.

2.3 Versión de fuerza bruta y cálculo de umbral:

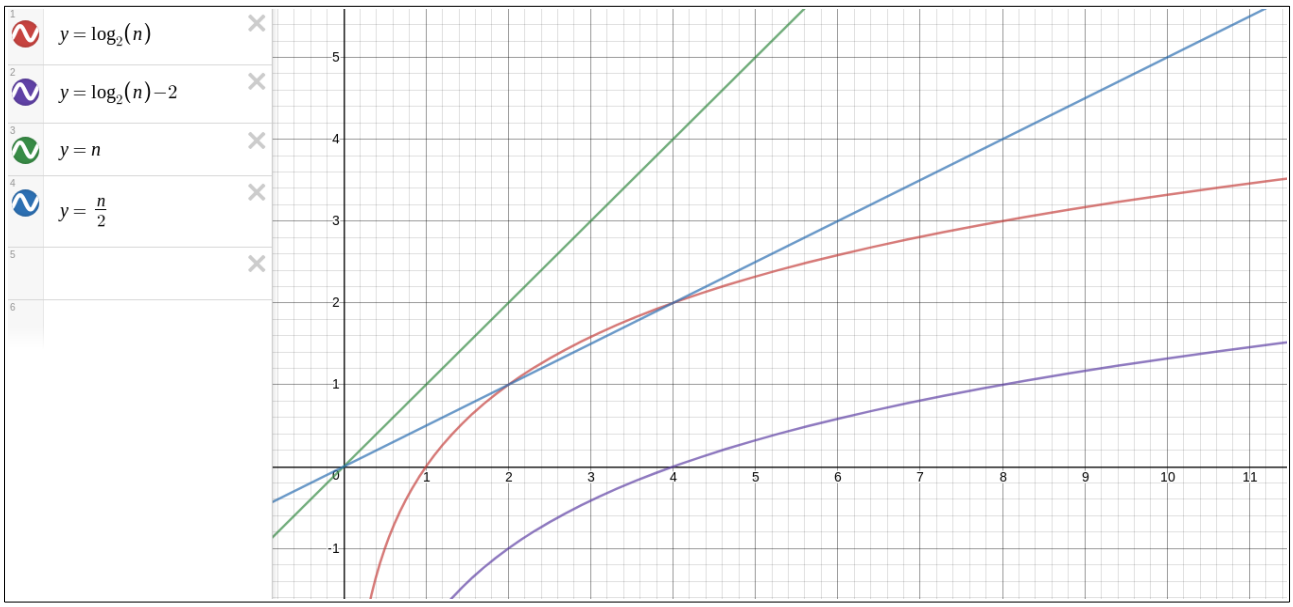
El algoritmo de fuerza bruta para este problema es trivial:

1) Situarse en la primera posición del vector
2) Si el valor de la posición siguiente a la actual es menor que el de la actual, solución encontrada: p es la posición actual. En caso contrario, avanzar una posición.
3) Volver al paso dos.

Su eficiencia es trivial también:

- En el mejor caso, **Ω(1)**, que se corresponde con la instancia del problema en la que la solución está en la primera posición del vector.
- En el peor caso, **O(n)**, que se corresponde con la instancia del problema en la que la solución está en la última posición del vector.
- En el caso medio, **Θ(n/2)**, ya que asumimos que la distribución de probabilidades de encontrar la solución en una posición es uniforme a lo largo de todo el vector.

Así, procedemos a calcular el umbral mediante el estudio de las gráficas:



Observamos que ni en los casos peores ni en los casos medios hay intersección: siempre es mejor la versión divide y vencerás del algoritmo. No obstante, deberíamos tener en cuenta las características de la implementación, ya que la función de la versión divide y vencerás tendrá bastantes más comparaciones y declaraciones de variables que la versión de fuerza bruta. Por esto, declaramos el umbral como un número pequeño relativamente arbitrario, ya que esas últimas llamadas recursivas de la función principal serán mucho mas costosas que una única llamada a la función de fuerza bruta.

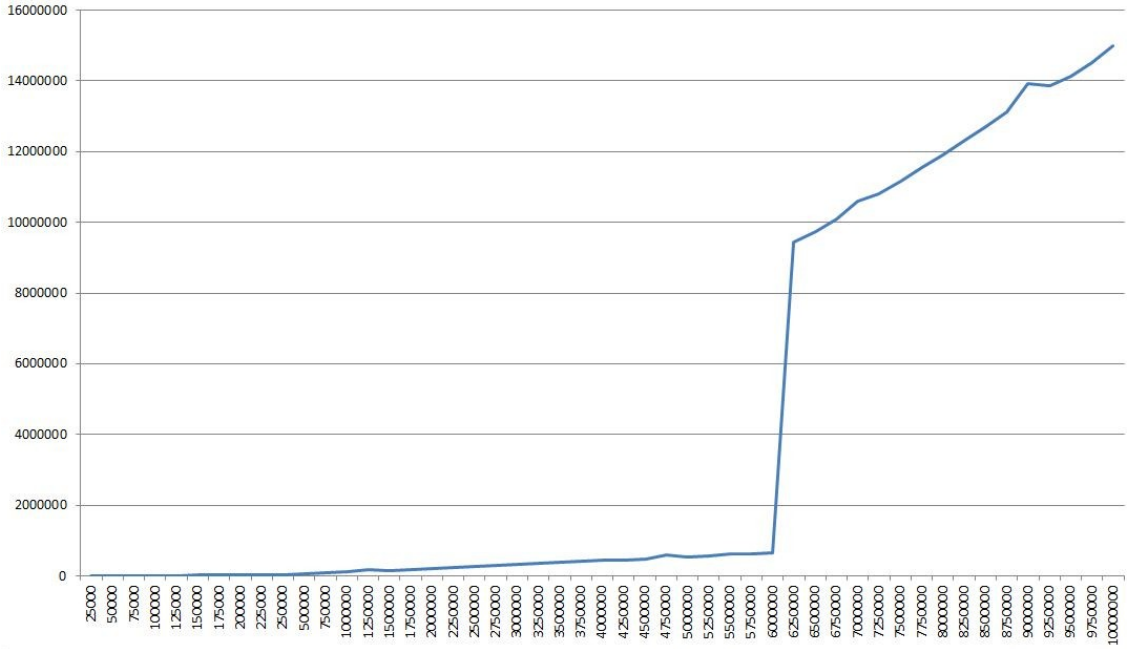
2.4 Implementación

Podemos ver la implementación en C++ de las dos versiones del algoritmo en el Anexo II. Al necesitar la versión de fuerza bruta para vectores de tamaño menor al umbral, se implementan ambas versiones del algoritmo en el mismo fichero. Se han comentado las partes de código destinadas a mostrar el caso de ejecución que se adjunta más adelante.

2.5 Datos experimentales por fuerza bruta

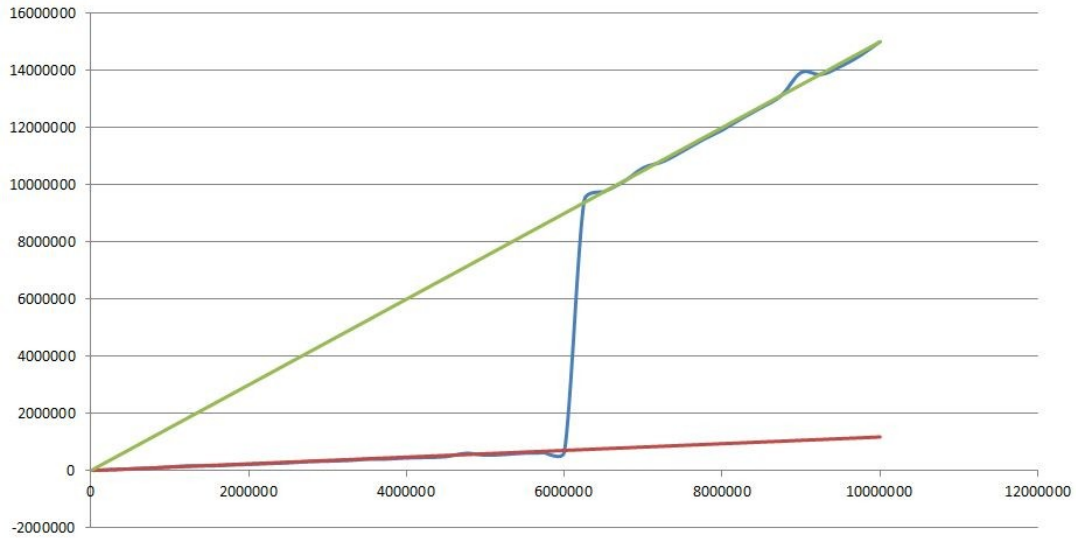
2.5.1 Eficiencia Empírica

Datos recogidos utilizando el algoritmo de fuerza bruta:



Identificamos un gran cambio en los tiempo de ejecución (medidos en nanosegundos) al rededor del tamaño de problema 600000 que se explicará a continuación. A parte de esto, se ve claramente como tenemos dos mitades claramente lineares, diferenciadas únicamente en la pendiente. Analizando la situación, descubrimos que esta discontinuidad se produce cuando el vector dinámico supera los 2GB de memoria, lo que nos hace suponer que la reserva de memoria se realiza en la parte de la memoria virtual no correspondiente con la memoria principal por el estado de esta en el momento de la ejecución. Al suponer un coste de tiempo mayor las lecturas de memoria, el tiempo de cada ejecución queda multiplicado por el ratio de velocidad de lectura entre memorias. Podemos suponer que en una máquina con más memoria principal no existiría esta discontinuidad.

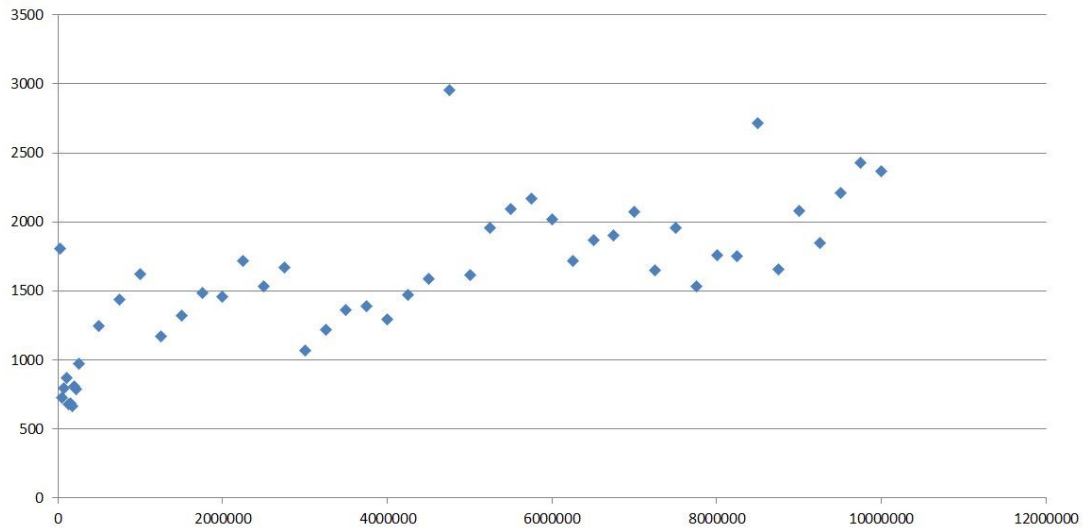
2.5.2 Eficiencia Híbrida



Se han calculado dos constantes multiplicativas para la función teórica, correspondientes a las dos partes de la gráfica, para demonstrar que estas dos partes se corresponden a la misma función de número de instrucciones con diferentes tiempos medios de operación elemental.

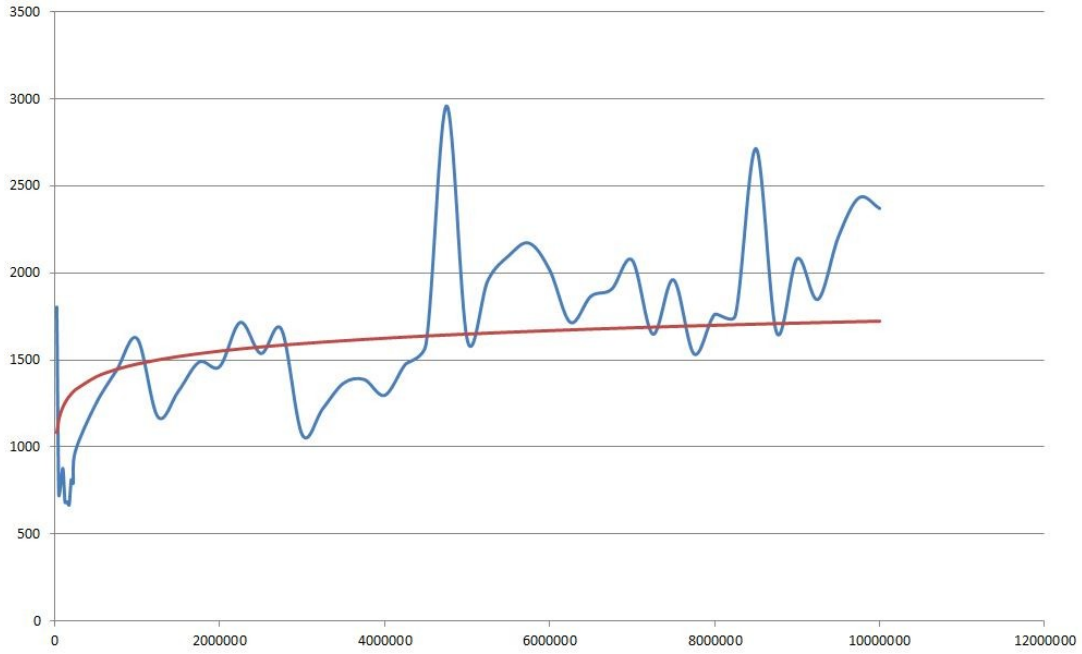
2.6 Datos experimentales por divide y vencerás

2.6.1 Eficiencia empírica



Al ser un algoritmo parecido a la búsqueda binaria, tenemos la misma gran desviación del caso medio que en la búsqueda binaria. Podemos observar bastantes grupos de tres datos que forman secuencias de tiempos linearmente ascendentes, esto lo podemos atribuir a que problablemente el generador de números pseudo-aleatorios genere números cuyas profundidades en el árbol de búsqueda no tengan una distribución uniforme.

2.6.2 Eficiencia híbrida



Si nos fijamos bien, podemos apreciar un aumento de los resultados de tiempo al rededor del tamaño de problema 600000. El cambio de almacenamiento no es significativo para la eficiencia de este algoritmo ya que las consultas de memoria no son su operación más frecuente, además de que realiza muchas menos iteraciones que la versión de fuerza bruta.

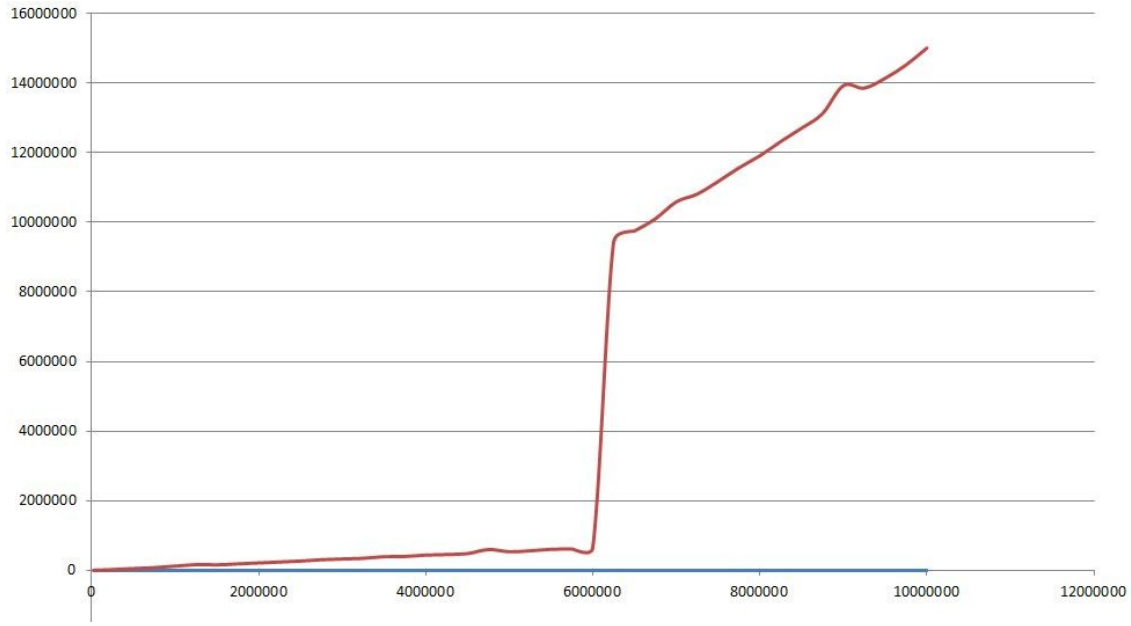
2.6.3 Caso de ejecución

```
migue@migue:~/Escritorio/ALG/practica2$ ./bin/serie-unimodal 30
GENERAMOS EL VECTOR Y P
P = 19
UMBRAL = 5

Vector actual: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 29 28 27 26 25 24 23 22 21 20 19
P EN MITAD DERECHA DEL VECTOR

Vector actual: 15 16 17 18 29 28 27 26 25 24 23 22 21 20 19
P EN MITAD IZQUIERDA DEL VECTOR

Vector actual: 15 16 17 18 29 28
P EN MITAD DERECHA DEL VECTOR

Vector actual: 18 29 28
N < UMBRAL -> FUERZA BRUTA
P ENCONTRADO, P = 19

TAM = 30        t = 98136
```

Se ha incluido un caso de ejecución con relativamente muchas iteraciones, para describir mejor el proceso. Nótese como se va partiendo el vector al realizar la búsqueda.

2.7 Comparación de las versiones del algoritmo

Aquí podemos ver los datos recogidos de ambas versiones:



Como esperábamos, el divide y vencerás es ampliamente más eficiente que el fuerza bruta.

Anexo I

```
//traspuestadyv.cpp
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <chrono>

using namespace std;
using namespace chrono;

const int MAXIMO=100000;

void imprimir(int** matriz, int n, int m){
	for (int i=0; i<n; ++i){
		for (int j=0; j<m; ++j)
			cout << matriz[i][j] << "   ";
		cout << endl;
	}
}

void TrasponDiagonal(int** &matriz, int** &traspuesta, int n, int m){
	int menor;
	if(n < m)
		menor = n;
	else if(m < n)
		menor = m;
	else
		menor = n;

	for(int i = 0; i < menor; ++i)
		traspuesta[i][i] = matriz[i][i];
}

void TrasponTrianguloInferior(int** &matriz, int** &traspuesta, int n, int m){
	int j = 1;
	int i = 0;
	bool terminado = false;

	if(n != 1)
		while(!terminado){
			traspuesta[i][j] = matriz[j][i];

			j++;

			if(j == n){
				i++;
				j = i+1;

				if(j == n || i == m)
```

```
                                        terminado = true;
                        }
                }
}

void TrasponTrianguloSuperior(int** &matriz, int** &traspuesta, int n, int m){
        int j = 0;
        int i = 1;
        bool terminado = false;

        if(m != 1)
                while(!terminado){
                        traspuesta[i][j] = matriz[j][i];
                        i++;
                        if(m == i){
                                j++;
                                i = j+1;
                                if(j == n || i == m)
                                        terminado = true;
                        }
                }
}

void traspuesta(int** &matriz, int &n, int &m){
        int** traspuesta;
        int aux;

        traspuesta=new int*[m];

        for (int i=0; i<m; ++i)
                traspuesta[i]=new int[n];

        //cout << endl << "1. Se copia la diagonal" << endl;
        TrasponDiagonal(matriz, traspuesta, n, m);
        //imprimir(traspuesta, m, n);
        //cout << endl << "2. Se traspone el triangulo superior" << endl;
        TrasponTrianguloSuperior(matriz, traspuesta, n, m);
        //imprimir(traspuesta, m, n);
        //cout << endl << "3. Se traspone el triangulo inferior" << endl;
        TrasponTrianguloInferior(matriz, traspuesta, n, m);
        //imprimir(traspuesta, m, n);

        for (int i=0; i<n; ++i)
                delete[] matriz[i];
        delete[] matriz;

        matriz=traspuesta;
        aux=n;
        n=m;
        m=aux;
```

```
}

int main(int argc, char** argv){
	if (argc<3){
		cerr << "Necesarios dos argumentos: dimensiones de la matriz" << endl;
		return 1;
	}

	int n, m;
	int** matriz;
	srand(time(NULL));
	chrono::_V2::system_clock::time_point t1, t2;
	unsigned long t;


	n=strtol(argv[1], NULL, 10);
	m=strtol(argv[2], NULL, 10);

	matriz=new int*[n];
	for (int i=0; i<n; ++i){
		matriz[i]=new int[m];
		for(int j=0; j<m; ++j)
			matriz[i][j]=rand()%MAXIMO;
	}

	cout << "MATRIZ ORIGINAL" << endl;
	imprimir(matriz, n, m);

	t1=high_resolution_clock::now();
	traspuesta(matriz, n, m);
	t2=high_resolution_clock::now();

	cout << endl << "TRASPUESTA" << endl;
	imprimir(matriz, n, m);

	t=duration_cast<microseconds>(t2-t1).count();
	//cout << n << "x" << m << "\t" << t << endl;

	return 0;
}
```

Anexo II

```
//serie-unimodal.cpp
#include <iostream>
#include <cstdlib>
#include <climits>
#include <cassert>
#include <chrono>

using namespace std;
using namespace chrono;

#define UMBRAL 5

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

int encuentra_p(int * v, int tam){
	int p=-1;

	for(int i=0; i<tam-1; ++i){
		if(v[i]>v[i+1]){
			p=i;
			break;
		}
	}
	if(p==-1)
		p=tam-1;

	return p;
}

int encuentra_p_divide(int * v, int tam){
	int p, m;
	bool md, mi;
	//cout << "\nVector actual: ";
	/*for (int i=0; i<tam; ++i)
		cout << v[i] << " ";
	cout << endl;*/

	if(tam<=UMBRAL){
		//cout << "N < UMBRAL -> FUERZA BRUTA" << endl;
		p=encuentra_p(v, tam);
	}else{
		m=(tam/2)-1;
		md=v[m]<v[m+1];
```

```
		mi=v[m-1]<v[m];

		if(md && mi){
			//cout << "P EN MITAD DERECHA DEL VECTOR" << endl;
			p=m+1+encuentra_p_divide(v+m+1, tam-m-1);
		}else if(!md && !mi){
			//cout << "P EN MITAD IZQUIERDA DEL VECTOR" << endl;
			p=encuentra_p_divide(v, m);
		}else{
			//cout << "P EN EL CENTRO DEL VECTOR" << endl;
			p=m;
		}
	}
	return p;
}

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

	int n = strtol(argv[1], NULL, 10);
	int P;
	int * T = new int[n];
	chrono::_V2::system_clock::time_point t1, t2;
	unsigned long t;

	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;

	//cout << "GENERAMOS EL VECTOR Y P\nP = " << p << endl;

	t1=high_resolution_clock::now();
	P=encuentra_p_divide(T, n);
	t2=high_resolution_clock::now();
	cout << "P ENCONTRADO, P = " << P << endl;

	cout << "P: " << P << endl;*/
	t=duration_cast<nanoseconds>(t2-t1).count();
	cout << n << "\t\t" << t << endl;
	delete[] T;
	return 0;
}
```