Algorítmica (2016-2017)

Grado en Ingeniería Informática

Universidad de Granada

Práctica 2

Algoritmos Divide y Vencerás: Eliminar elementos repetidos

Gregorio Carvajal Expósito
Gema Correa Fernández
Jonathan Fernández Mertanen
Eila Gómez Hidalgo
Elías Méndez García
Alex Enrique Tipán Párraga

Contenidos

Introducción

En esta práctica, lo que se nos pide es que dado un vector de *n* elementos, de los cuales algunos pueden estar duplicados, obtener otro vector donde todos los elementos duplicados hayan sido eliminados.

Para esta tarea diseñaremos y analizaremos la eficiencia implementando un algoritmo sencillo. Luego, haremos lo mismo con un algoritmo más eficiente, basado en “divide y vencerás”, de orden *O(n logn*). Además, realizaremos para ambos algoritmos un estudio empírico e híbrido.

Características del Ordenador

Para la realización del estudio de la eficiencia, ejecutaremos los datos en un ordenador con las siguientes características:

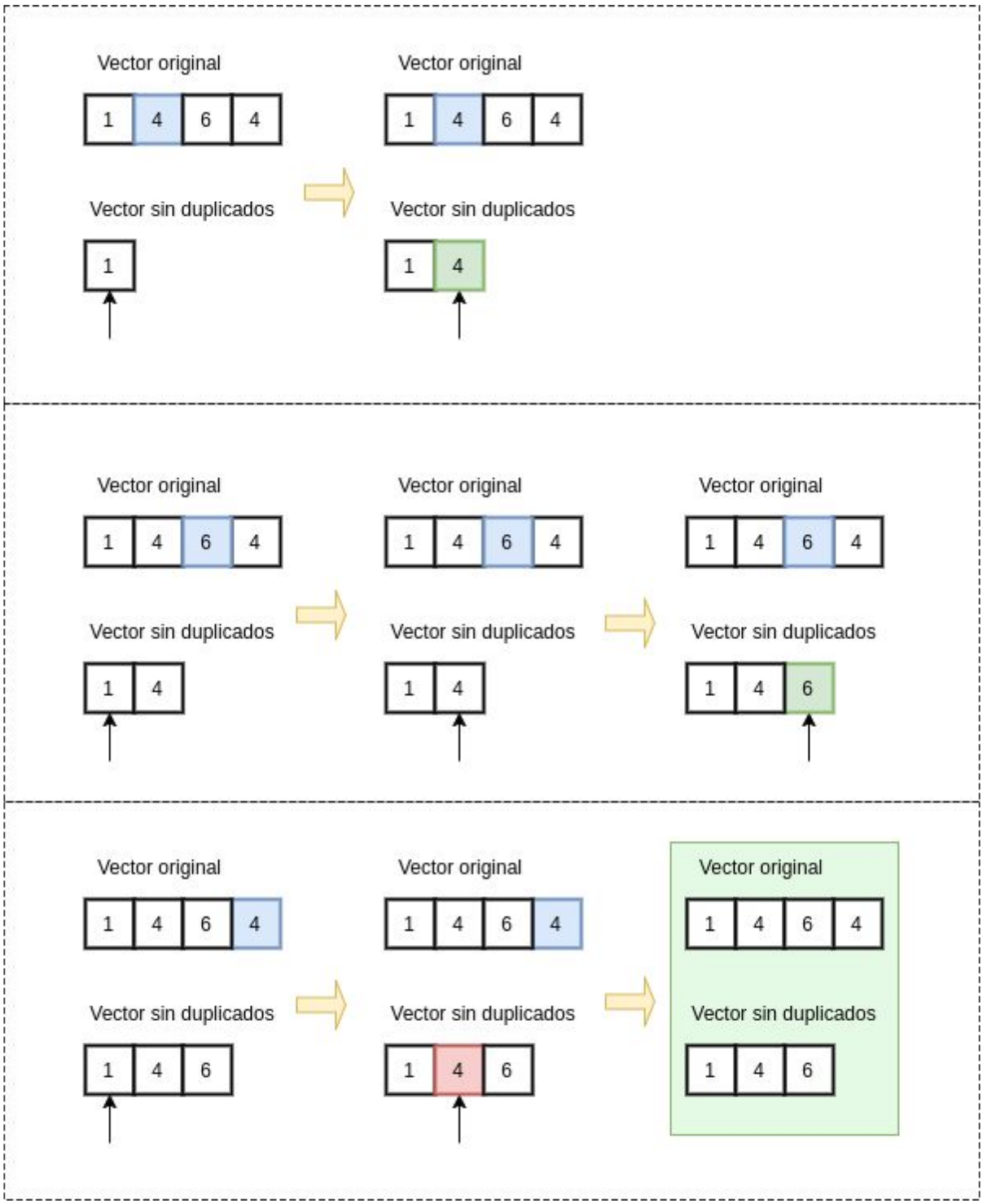
- **Procesador:** Intel(R) Core(TM) i7-4720HQ.
- **Núcleos y velocidad:** 8 Núcleos a 2.60GHz.
- **Cachés:** Caché L1 32K | Caché L2 256K | Caché L3 6144K.
- **Arquitectura:** 64 bits.
- **Memoria RAM:** 16GB a 1600 MHz.
- **Sistema Operativo:** Arch Linux (rolling release).
- **Versión de kernel:** 4.10.6-1-ARCH
- **Versión de compilador:** g++ (GCC) 6.3.1 201.70306.
- **Opciones de compilación:** -O3 -std=c++11

Algoritmo Sencillo (Sin usar DyV)

Implementaremos un algoritmo sencillo que nos devuelva un vector, donde todos sus elementos duplicados hayan sido eliminados.

Para ello iremos recorriendo el vector original y buscaremos cada elemento de éste en el vector sin duplicados. Si un elemento no se encuentra significa que es la primera vez que aparece en el vector original, entonces debemos insertarlo en vector sin duplicados. Si por el contrario ya lo hemos insertado anteriormente continuamos con el siguiente elemento del vector el original. Al final tendremos un nuevo vector con todos los elementos que aparecen el el original pero sin duplicar, ya que solamente insertamos en la primera aparición de cada elemento.

En la siguiente imagen podemos ver el funcionamiento del algoritmo a partir de un pequeño ejemplo:



La implementación en la que nos basaremos es la siguiente:

```
inline static Vector* elimina_repetidos(const Vector &vec){

	int n_elem_max = vec.n_elem;
	int elem;
	bool encontrado = false;

	Vector* sin_repetidos = new Vector;
	sin_repetidos->v = new int[n_elem_max];

	for (int i = 0; i < n_elem_max; ++i) {
		elem = vec.v[i];

		for (int j=0; j<sin_repetidos->n_elem && !encontrado; ++j) {
			if (elem == sin_repetidos->v[j])
				encontrado = true;
		}

		if (!encontrado) {
			sin_repetidos->v[sin_repetidos->n_elem] = elem;
			++sin_repetidos->n_elem;
		}
		else
			encontrado = false;
	}

	return sin_repetidos;
}
```

Como vemos en la implementación anterior mediante un primer for vamos recorriendo cada elemento del vector y un segundo for anidado se encarga de comprobar si ese elemento ha sido insertado con anterioridad en el nuevo vector antes de insertarlo.

Algoritmo Divide y Vencerás

Ahora, realizaremos la misma tarea que antes, pero usando un algoritmo “divide y vencerás”.

La idea de este algoritmo se basa en ordenar un vector con un algoritmo “divide y vencerás”, como por ejemplo, mergesort. El mergesort consiste en ir subdividiendo el vector a la mitad y llamando de forma recursiva a la función mergesort con cada mitad del vector hasta llegar a la condición de parada. La condición de parada será cuando el vector tenga uno o dos elementos, en cuyo caso se comprueba quien es el mayor y se ordenan. Una vez hemos terminado de ordenar cada mitad, se juntan comprobando los dos vectores a la vez

para ir comprobar cuál es el mayor de ambos. Luego, se introduce en el vector en la posición adecuada.

```
void mergesort(int T[], int num_elem) {
	mergesort_lims(T, 0, num_elem);
}

static void mergesort_lims(int T[], int inicial, int final) {
	int diferencia = final - inicial;

	if (diferencia == 1) {
		return; //No hacer nada
	} else if (diferencia == 2) {
		if (T[0] > T[1]) {
			int aux = T[0];
			T[0] = T[1];
			T[1] = aux;
		}
	} else {
		int k = diferencia / 2;

		int *U = new int[k - inicial + 1];
		assert(U);
		int l, l2;

		for (l = 0, l2 = inicial; l < k; l++, l2++)
			U[l] = T[l2];

		U[l] = INT_MAX;

		int *V = new int[final - k + 1];
		assert(V);

		for (l = 0, l2 = k; l < final - k; l++, l2++)
			V[l] = T[l2];

		V[l] = INT_MAX;

		mergesort_lims(U, 0, k);
		mergesort_lims(V, 0, final - k);
		fusion(T, inicial, final, U, V);

		delete[] U;
		delete[] V;
	}
```

```
}

static void fusion(int T[], int inicial, int final, int U[], int V[]) {
	int j = 0;
	int k = 0;

	for (int i = inicial; i < final; i++) {
		if (U[j] < V[k]) {
			T[i] = U[j];
			j++;
		} else {
			T[i] = V[k];
			k++;
		}
	}
}
```

Una vez ordenado, se recorre el vector comprobando el elemento actual y el siguiente. En caso de que ambos sean distintos, añadimos el siguiente al vector sin repetidos. Para que el algoritmo funcione correctamente, se añade siempre el primer elemento antes de comenzar a recorrer el vector.

```
inline static Vector* elimina_repetidos(const Vector &vec)
{
	int elem, sig_elem, n_elem_max = vec.n_elem;

	int *copia = new int[n_elem_max];
	std::copy(vec.v, vec.v + n_elem_max, copia);

	mergesort(copia, n_elem_max);

	Vector* sin_repetidos = new Vector;
	sin_repetidos->v = new int[n_elem_max];

	sin_repetidos->v[0] = copia[0];
	++sin_repetidos->n_elem;

	for (int i = 0; i < n_elem_max - 1; ++i)
	{
		elem = copia[i];
		sig_elem = copia[i + 1];

		if (elem != sig_elem) {
			sin_repetidos->v[sin_repetidos->n_elem] = sig_elem;
			++sin_repetidos->n_elem;
```

```
            }
        }

        delete []copia;

        return sin_repetidos;
}
```

Eficiencia Empírica

Procedimiento

Para el cálculo de la eficiencia empírica de los distintos algoritmos, hemos realizado el siguiente procedimiento:

1. Para medir los tiempos en cada programa se ha utilizado la biblioteca `chrono` del estándar de c++11. Para ello, añadimos a la cabecera de cada programa, lo siguiente:

```
#include <chrono>
using namespace std::chrono;
```

2. Para mejorar la automatización de la obtención de tiempos al programa, se ha editado el código para que reciba dos argumentos. El primero es el número de datos con los que se debe ejecutar el algoritmo y el segundo es el número con el que se debe repetir el mismo procedimiento para sacar una media de tiempos y así intentar evitar las variaciones de tiempos.

```
if (argc != 3) {
    cerr << "El programa necesita dos argumentos." << endl
           << "Uso: " << argv[0]
           << "<número de datos> <número de repeticiones>"
           << endl;

    return (-1);
}

int n = std::stoi(argv[1]);            // número de datos
int iteraciones = std::stoi(argv[2]);  // número de repeticiones
```

3. Para garantizar que todas las repeticiones se hacen con los mismos datos antes de ser ordenados, lo primero que hacemos, al empezar otra iteración, es crear una copia de los datos originales y trabajar sobre esa copia.

4. Para ayudarnos a calcular la media del tiempo de las distintas iteraciones se ha creado una clase llamada `TimeCollector` que almacena los tiempos y es capaz de mostrarlos por pantalla en un formato compatible con *gnuplot*. También es capaz de exportar la medida de tiempo a un archivo *.csv* para que sea más sencillo su utilización con programas de hojas de cálculo.

5. Para la medición de tiempos usamos `high_resolution_clock::now()`, que nos da la ventaja de obtener medidas de tiempo muy pequeñas, que no se van a redondear a cero. Para la medición de tiempo, se consulta el reloj antes y después de lanzar el algoritmo y calculamos la diferencia entre ambos momentos de tiempo para obtener el tiempo de ejecución final. Por último, haremos la media con la acumulación de tiempos obtenidos.

```
TimeCollector collector(n, iteraciones);

for (int i = 0; i < iteraciones; ++i) {

      auto t1 = high_resolution_clock::now();
      sin_repetidos = elimina_repetidos(vec);
      auto t2 = high_resolution_clock::now();

      collector.addTime(duration_cast<duration<double>>
                                    (t2 - t1).count());

      delete[] sin_repetidos->v;
      delete sin_repetidos;
}
```

6. Una vez terminada la ejecución, se obtiene la media de los tiempos y se muestra por pantalla en un formato compatible con *gnuplot* y se escribe en un archivo *.csv*.

```
collector.print();
collector.writeToCsv("repetidos_divide.csv");
```

Para facilitar la obtención de los tiempos se ha usado el siguiente *Makefile* y *script*. Para generar las curvas de regresión y obtener los valores del ajuste con la curva de tendencia nos hemos ayudado de *gnuplot* y del *script* `generateRegresion.sh`.

Makefile

```
##############################################################################
# Makefile
##############################################################################

SHELL = /bin/bash # for ubuntu

##############################################################################

SRC = $(wildcard *.cpp)
EXE = $(basename $(SRC))
DAT = $(EXE:=.dat)
CSV = $(EXE:=.csv)
SVG = $(EXE:=.svg)

##############################################################################

CFLAGS = -O3
CXXFLAGS =  $(CFLAGS) -std=c++11 # -std=c++1y

##############################################################################

default: $(EXE)

all: default data

clean:
	$(RM) -rfv $(EXE)

purge: clean
	$(RM) -rfv $(SVG) $(DAT) $(CSV)

data: default
	chmod +x generateData.sh
	./generateData.sh
```

generateData.sh

```
#!/bin/bash

# Argumentos
# $1 nombre del programa a ejecutar
# $2 tamaño de datos inicial
# $3 tamaño de datos de la última iteración
# $4 incremento entre cada iteración
# $5 número de veces que el programa repetirá el cálculo para
sacar la media

function generateData() {
  echo "" > $1.dat

  for ((c = $2; c <= $3; c += $4)); do
    echo "Calculo con $c datos y $5 iteraciones"
    ./$1 $c $5 >> $1.dat
  done
}

echo "Calculando los tiempos del algoritmo: eliminar alternativo"
generateData repetidos_sencillo 10000 250000 10000 15

echo "Calculando los tiempos del algoritmo: eliminar divide y
venceras"
generateData repetidos_divide 10000 250000 10000 30
```

generateRegresion.sh

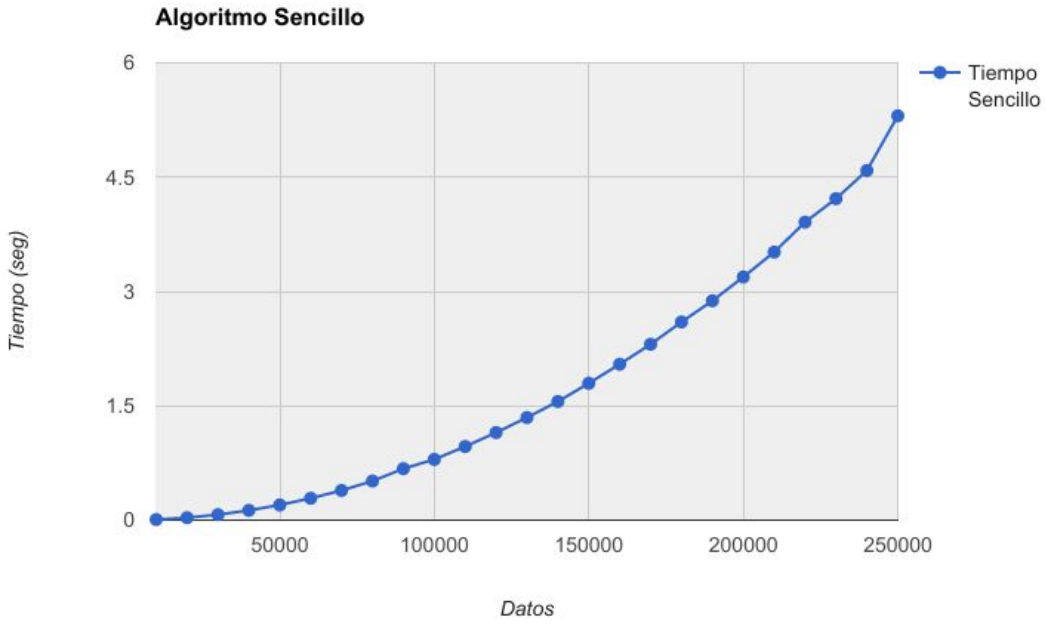
```
#!/bin/bash
#
http://computingnote.blogspot.com.es/2013/04/calculating-r2-with-gnuplot
.html

#Argumentos
# $1 nombre del fichero
# $2 formula con la que ejecutas la regresion
# $3 variables de la regresion
function createGraphAndData() {
  echo "Generando gráfico de: $1"
  gnuplot <<< "\
            set terminal png; \
            set output 'grafico_$1.png'; \
            set xlabel 'Tamaño'; \
            set ylabel 'Tiempo (seg)'; \
            mean(x)= m; \
            fit mean(x) '$1.dat' using 1:2 via m; \
            SST = FIT_WSSR/(FIT_NDF+1); \
            f(x) = $2; \
            fit f(x) '$1.dat' using 1:2 via $3; \
            SSE=FIT_WSSR/(FIT_NDF); \
            SSR=SST-SSE; \
            R2=SSR/SST; \
            set title '$1'; \
            set key left Left; \
            set title sprintf('R^2=%f', R2); \
            plot '$1.dat' title 'Eficiencia de $1', f(x) title 'Curva
      ajustada'" \
      2> regresion_$1.txt
}

# variables de las formulas.
cuadrada="a0*x*x+a1*x+a2";
logaritmica="a0*x*log10(x)";

# variables de la regresion por formula
v_cuadrada="a0,a1,a2";
v_logaritmica="a0";

createGraphAndData "repetidos_sencillo" $cuadrada $v_cuadrada
createGraphAndData "repetidos_divide" $logaritmica $v_logaritmica
```

A continuación, mostramos las tablas de tiempos de ambos algoritmos, representando cada uno con una gráfica.

Tiempos del algoritmo sencillo

Algoritmo Sencillo	
Nº de datos	**Tiempo (seg)**
10000	0.00813441
20000	0.032267
30000	0.0722235
40000	0.128482
50000	0.199211
60000	0.287266
70000	0.389983
80000	0.513263
90000	0.675671
100000	0.797275
110000	0.965846
120000	1.14919
130000	1.34652
140000	1.55632
150000	1.79604
160000	2.04587
170000	2.30748
180000	2.59981
190000	2.87792
200000	3.18999
210000	3.51723
220000	3.90999
230000	4.21788
240000	4.58698
250000	5.30422

Tabla 1. Tiempos obtenidos para el Algoritmo Sencillo



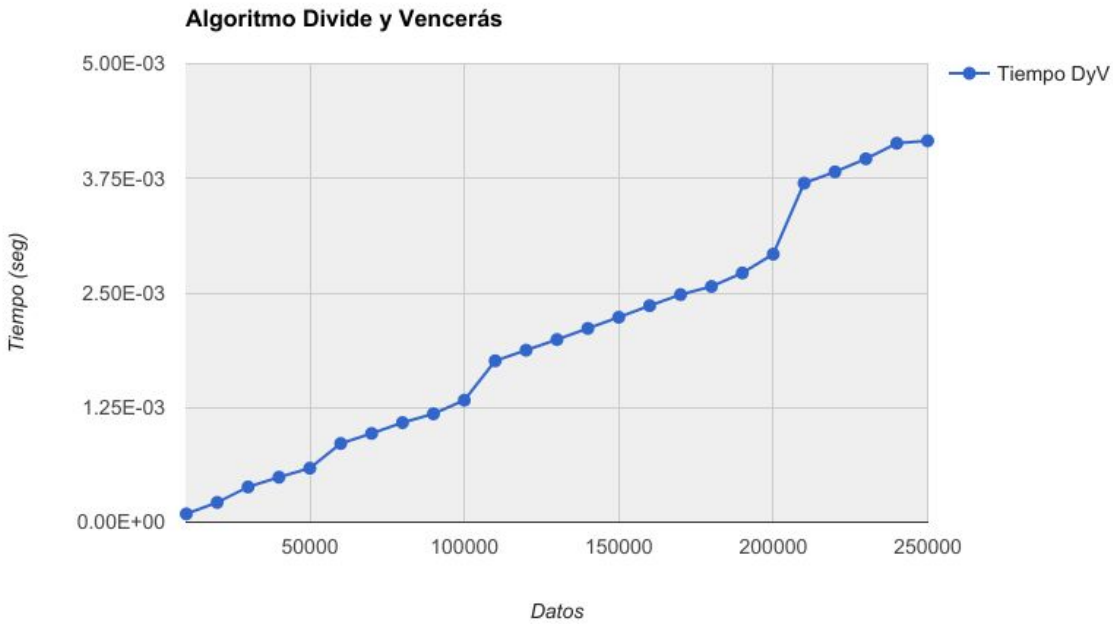
Gráfica 1. Gráfica del Algoritmo Sencillo

Tiempos del algoritmo divide y vencerás

Algoritmo Divide y Vencerás	
Nº de datos	**Tiempo (seg)**
10000	9.06E-05
20000	0.000215461
30000	0.000384402
40000	0.000490725
50000	0.000589327
60000	0.000858675
70000	0.000968072
80000	0.00108605
90000	0.0011821
100000	0.00133045
110000	0.00176017
120000	0.001876
130000	0.00199213
140000	0.00211383
150000	0.00223552

160000	0.00236148
170000	0.00248253
180000	0.002571
190000	0.00271817
200000	0.00292369
210000	0.00369807
220000	0.00382124
230000	0.00396376
240000	0.0041334
250000	0.00416028

Tabla 2. Tiempos obtenidos para el Algoritmo Divide y Vencerás



Gráfica 2. Gráfica del Algoritmo Divide y Vencerás

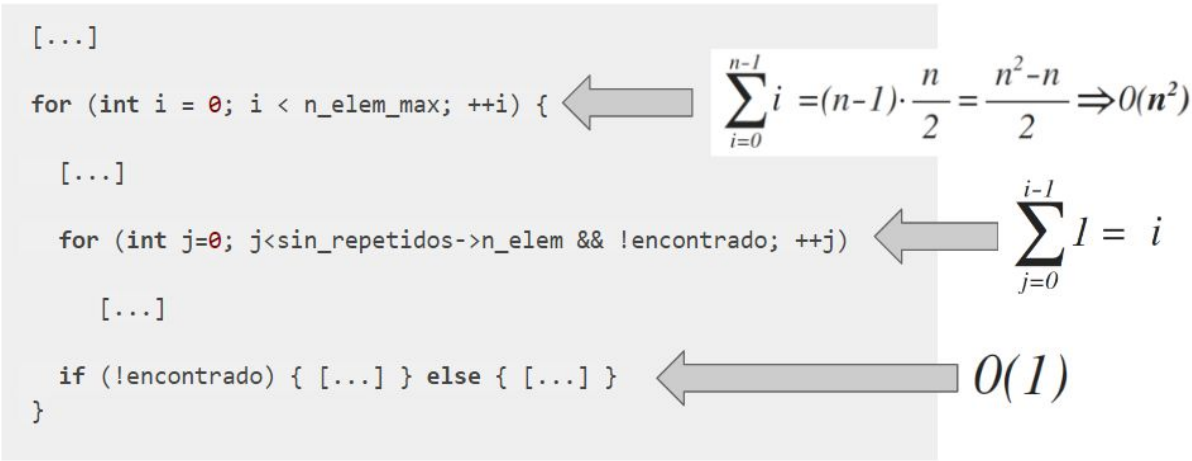
Como se puede comprobar, se reduce considerablemente el tiempo con la metodología de divide y vencerás.

Eficiencia Teórica

Vamos a analizar el código de ambos algoritmos con el fin de obtener su eficiencia teórica.

Algoritmo Sencillo (Sin usar DyV)

Se puede apreciar a simple vista que en el código hay 2 `for` anidados que van desde 0 hasta n, así que podemos decir que su eficiencia es **O(n^2)**, veámoslo gráficamente:



Algoritmo Divide y Vencerás

Este algoritmo está dividido en dos partes: la parte de ordenación y la parte de copia de elementos “correctos” a un nuevo vector.

Para la parte de ordenación, hemos usado el algoritmo mergesort puro, sin utilizar otro algoritmo más básico por debajo de ningún umbral. Ya vimos en la práctica anterior la eficiencia del algoritmo mergesort, y obtuvimos que era **O(n log(n)).**

Dicho esto, vamos a analizar la parte de elementos correctos al nuevo vector. Esta parte está formada por un `for` que recorre el vector entero y sus instrucciones anidadas son todas O(1), por tanto podemos afirmar que la parte de copia tiene un orden de eficiencia de **O(n)**.

Por tanto, primero vamos a calcular la eficiencia del `mergesort`. Analizando su código, obtenemos la siguiente recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right)+n \qquad n \geq 2$$

Ahora, expandimos la recurrencia:

$$
\begin{aligned}
T(n) &= 2T(n/2)+n && n \geq 2 \\
T(n) &= 2^2 T(n/4)+2n && n \geq 4 \\
&\vdots \\
T(n) &= 2^i T(n/2^i)+i \cdot n && n \geq 2^i
\end{aligned}
$$

entonces si $i = log_2(n)$ tenemos que $T(n) = n \cdot log_2(n)$

y por tanto el orden de eficiencia es: $O(n \cdot log_2(n))$

A continuación, se ve el orden de eficiencia de cada trozo del algoritmo “divide y vencerás”.

```
mergesort(copia, n_elem_max);

[...]

for (int i = 0; i < n_elem_max - 1; ++i)
{
     [...]

     if (elem != sig_elem)
     {
        [...]
     }
}
```

$O(n \cdot log(n))$

$\sum_{i=0}^{n-2} 1 = n-1 \Rightarrow O(n)$

$O(1)$

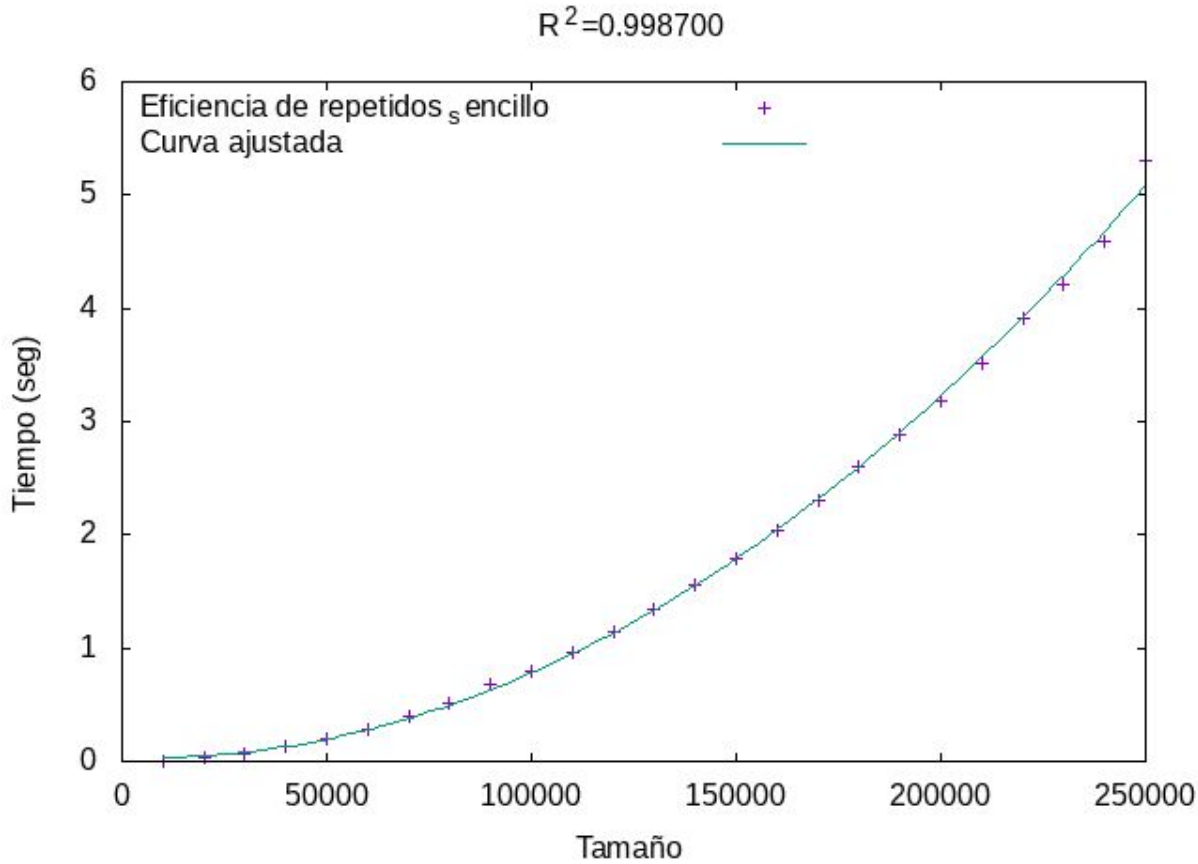
Por la regla de la suma, nos quedaremos con la mayor de las eficiencias, por eso el `for` que es O(n) no afecta a la eficiencia teórica, porque está la llamada a la función `mergesort` con un orden mayor. En conclusión, nuestro algoritmo para eliminar elementos repetidos tiene una eficiencia de orden **O(n log(n))**.

Eficiencia Híbrida

Usando los datos obtenidos en los apartados anteriores, tanto de la eficiencia teórica como de la empírica, hemos realizado un ajuste de regresión para el algoritmo “sencillo” y para el algoritmo con “divide y vencerás”. A continuación, mostramos las gráficas empíricas ajustadas al orden de eficiencia teórico obtenido en el apartado anterior.

Algoritmo Sencillo (Sin usar DyV)

Ecuación: $f(x) = a0*x^2 + a1*x + a2$
Constantes: a0 = 8.51193e-11 a1 = -1.07647e-06 a2 = 0.0378072

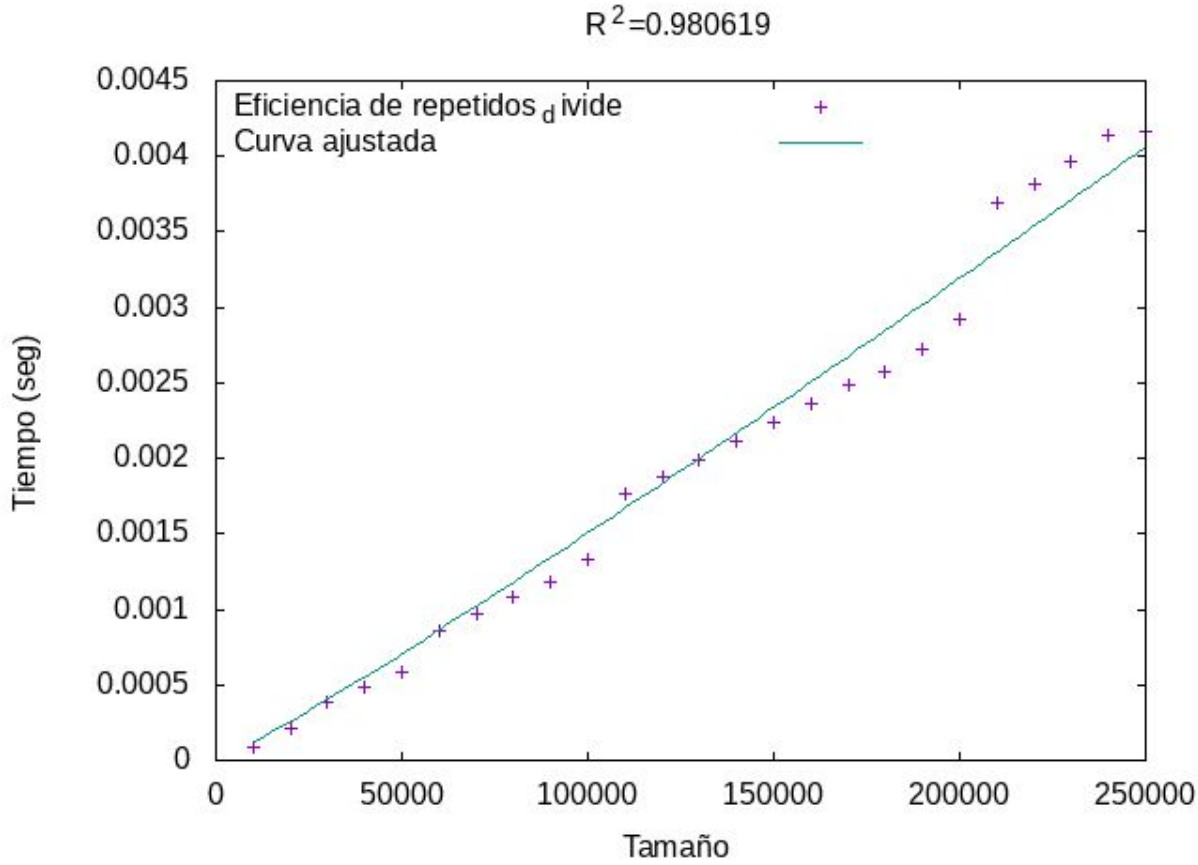


Gráfica 3. Ajuste de la curva del Algoritmo Sencillo

Como podemos observar en la gráfica el ajuste de la curva es casi perfecto, por lo tanto podemos confirmar que la eficiencia teórica se corresponde con la empírica. Obteniendo así, un ajuste de calidad. Esto lo podemos afirmar al ver que el coeficiente de determinación(R^2) se acerca mucho a 1.

Algoritmo Divide y Vencerás

Ecuación: f(x) = a0*x*log10(x)
Constantes: a0 = 3.01119e-09



Gráfica 4. Ajuste de la curva del Algoritmo Divide y Vencerás

Como se puede observar en la gráfica, el ajuste de la curva es considerablemente bueno, aunque se observa algún que otro pico. Además el valor de R^2 es cercano a 1. El bajón puede ser debido, que al usar vectores aleatorios, haya habido un momento con menos colisiones, es decir, que ese vector esté medianamente ordenado. Por tanto, tiene que hacer menos cambios.