Algorítmica

Grado en Ingeniería Informática

Práctica 2

El elemento en su posición

Autores

María Jesús López Salmerón

Nazaret Román Guerrero

Laura Hernández Muñoz

José Baena Cobos

Carlos Sánchez Páez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

Índice de cuadros

Índice de figuras

1. Descripción de la práctica

El objetivo de esta práctica es diseñar un algoritmo para resolver el problema del *elemento en su posición*. Este algoritmo consiste en los siguiente:
Dado un vector v, determinar si $\exists i : v[i] = i$ Se implementarán dos versiones de este algoritmo: una siguiendo el algoritmo obvio y otra empleando la estrategia *Divide y Vencerás*.

Para generar los vectores, utilizaremos el generador disponible en *decsai.ugr.es*.

2. Algoritmos diseñados

```
int elementoEnSuPosicion(const vector<int> v) {
        for (int i = 0; i < v.size() ; i++)
                if (v[i] == i)
                        return i;
        return -1;
}
```

Figura 1: Algoritmo clásico

```
int elementoEnSuPosicion(const vector<int> v, const int ini, const int fin) {
        if (ini == fin) {        //Caso base, vector de un solo elemento
                if (v[ini] == ini)
                        return ini;
                else
                        return -1;
        }
        else {        //Partimos en dos el vector y llamamos recursivamente
                int mitad = (ini + fin) / 2;
                int pos_izq = elementoEnSuPosicion(v, ini, mitad);
                int pos_dcha = elementoEnSuPosicion(v, mitad + 1, fin);
                if (pos_izq != -1)
                        return pos_izq;
                else
                        return pos_dcha;
        }
}
```

Figura 2: Algoritmo Divide y Vencerás

3. Estudio de eficiencia

En esta sección procederemos a estudiar la eficiencia de los algoritmos en cuestión.

3.1. Eficiencia teórica

Como podemos ver en el algoritmo clásico, iteramos por el bucle hasta encontrar un elemento en el que se cumpla la condición ($v[i] = i$). Por tanto, en el peor de los casos recorreremos el bucle completo, siendo la eficiencia de **O(n)**.

En cuanto al algoritmo Divide y Vencerás, nuestro algoritmo es recursivo y la metodología es la siguiente:

- **Caso base**. El vector consta de un solo elemento.
- Calculamos la posición del elemento mitad del vector.
- Llamamos recursivamente a la función desde el tamaño inicial hasta la mitad y desde la mitad + 1 hasta el final.
- Comprobamos si el elemento se encuentra en la mitad izquierda. En caso afirmativo, devolvemos su posición.
- En caso contrario, devolvemos el resultado de aplicar el algoritmo en la mitad derecha.

Como podemos ver, aun así en el peor de los casos tendremos que recorrer el vector en su totalidad, por lo que la eficiencia del algoritmo empleando la estrategia *Divide y Vencerás* es **O(n)**.

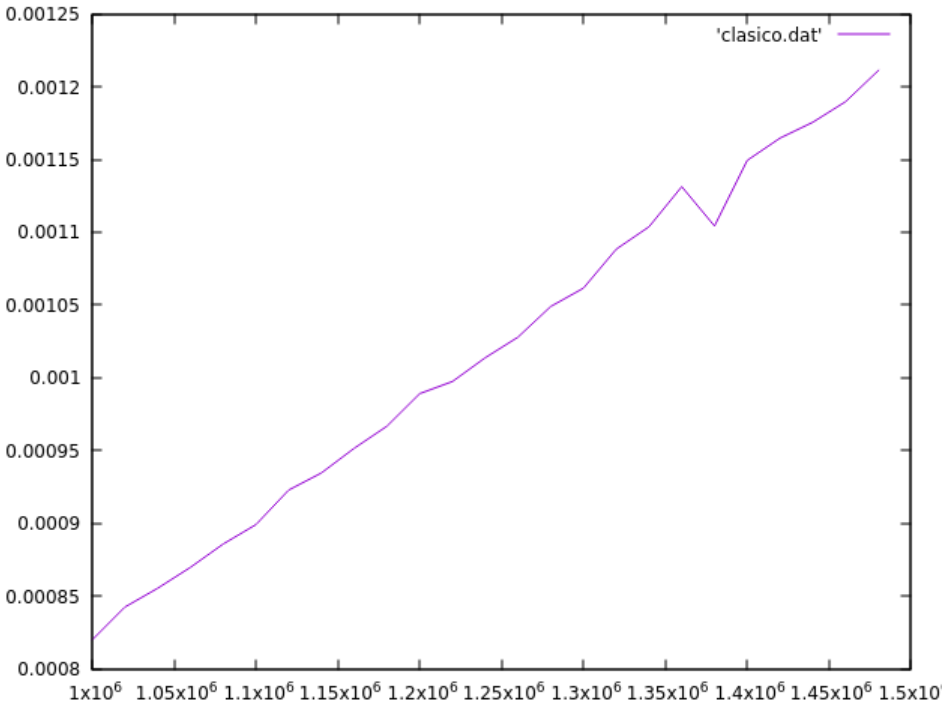
Por tanto, podemos adelantar que debido a los costes físicos de la recursión el algoritmo *clásico* será mejor que el *Divide y Vencerás*.

3.2. Eficiencia empírica

Haciendo uso de los scripts diseñados, ejecutamos los algoritmos con los siguientes tamaños:

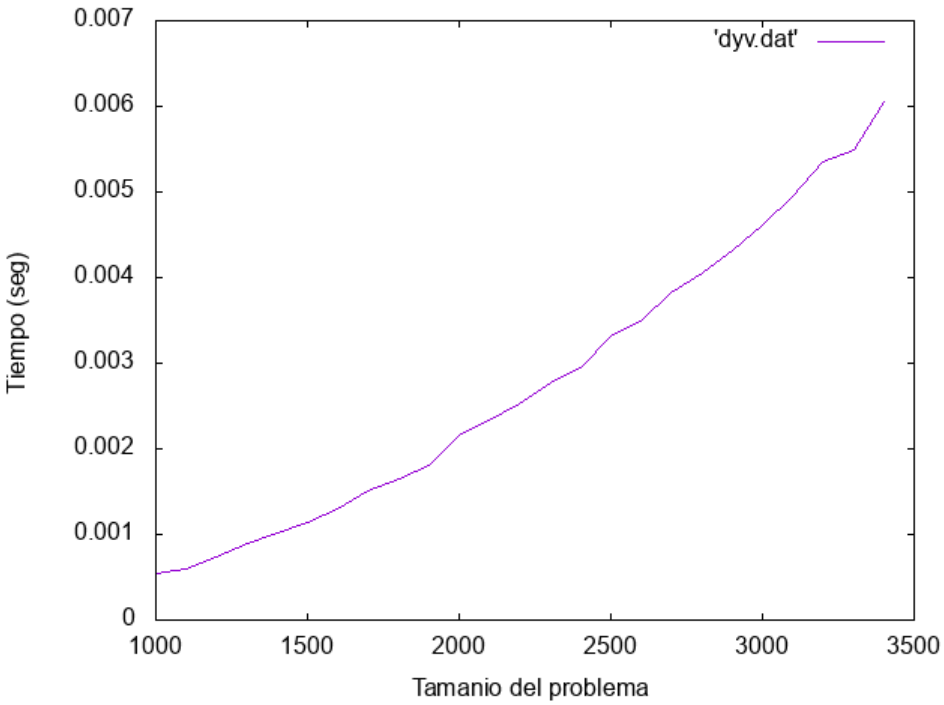
Algoritmo	Tamaño inicial	Tamaño final	Incremento
Clásico	1.000.000	1.480.000	20.000
Divide y Vencerás	1.000	3.400	100

Cuadro 1: Parámetros de ejecución de cada programa



Tamaño	**Tiempo (s)**
1000000	0.000819706
1020000	0.000842275
1040000	0.00085518
1060000	0.000869587
1080000	0.000885716
1100000	0.000899005
1120000	0.000922726
1140000	0.000934545
1160000	0.000951469
1180000	0.00096667
1200000	0.000988998
1220000	0.000997361
1240000	0.00101367
1260000	0.0010278
1280000	0.00104895
1300000	0.00106148
1320000	0.00108836
1340000	0.00110376
1360000	0.00113126
1380000	0.00110421
1400000	0.00114935
1420000	0.00116467
1440000	0.0011756
1460000	0.00118976
1480000	0.00121112

Figura 3: Algoritmo clásico



Tamaño	Tiempo (s)
1000	0.000551096
1100	0.000600369
1200	0.000736414
1300	0.000889537
1400	0.00100999
1500	0.00113876
1600	0.00129452
1700	0.00150525
1800	0.00165492
1900	0.00180961
2000	0.00215437
2100	0.00233609
2200	0.00252987
2300	0.00276466
2400	0.00294883
2500	0.00332321
2600	0.00349556
2700	0.00382512
2800	0.00405147
2900	0.00431329
3000	0.00460976
3100	0.00493946
3200	0.00534553
3300	0.00548893
3400	0.00605759

Figura 4: Algoritmo Divide y Vencerás

Algoritmo clásico	Algoritmo Divide y Vencerás
0,001009366976743	0,002223563169903

Figura 5: Tiempo medio (s)

3.3. Eficiencia híbrida

En esta sección ajustaremos los datos obtenidos a las expresiones teóricas obtenidas mediante una regresión con el objetivo de obtener su constante oculta.

Algoritmo	Eficiencia teórica	Valor de la constante oculta	Error
Clásico	O(n)	8.19304e-10	0.1441 %
Divide y Vencerás	O(n^2)	5.17311e-10	0.3221 %

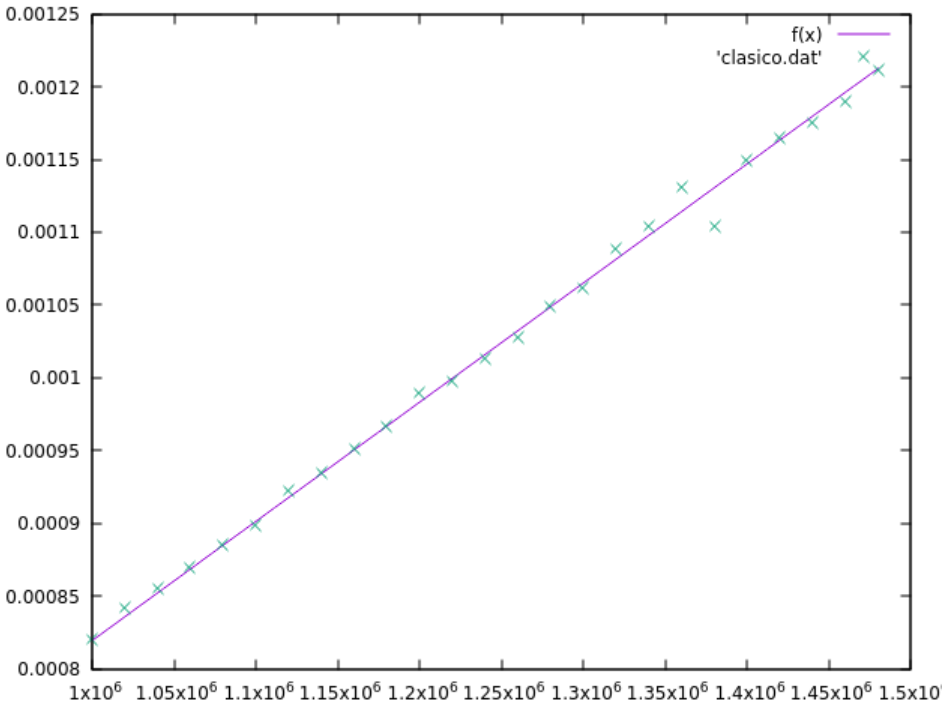


Figura 6: Eficiencia híbrida del algoritmo clásico

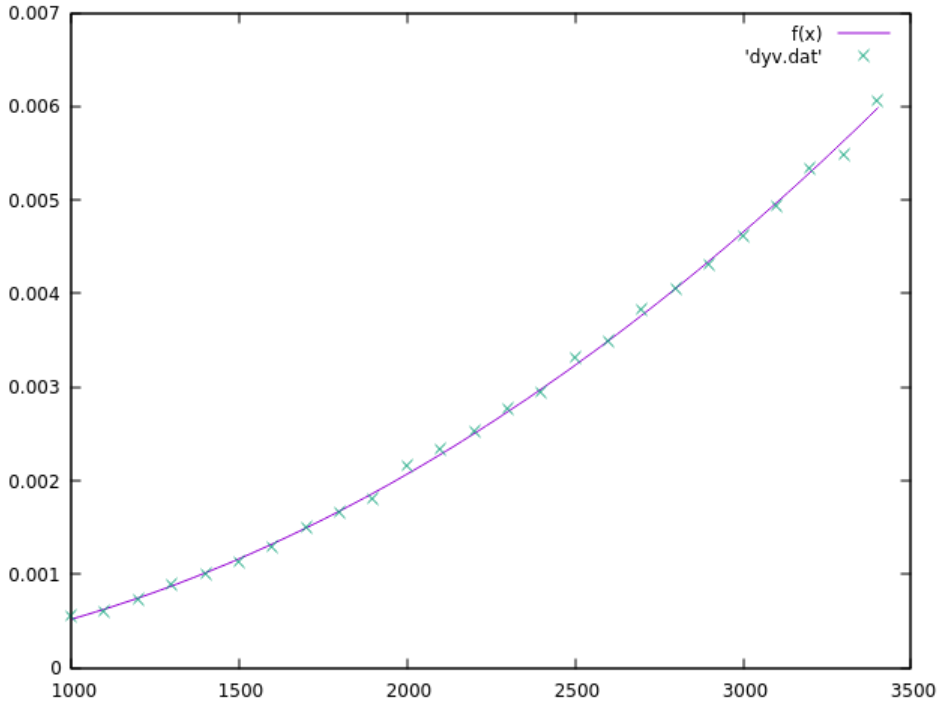


Figura 7: Eficiencia híbrida del algoritmo Divide y Vencerás

4. Conclusiones

5. Anexo: scripts implementados

```
#!/bin/bash
if [ $# -eq 3 ]
then
i="0"
tam=$2
#Primer argumento: programa a ejecutar
#Segundo argumento: tamaño inicial
#Tercer argumento : incremento
while [ $i -lt 25 ]
do
        ./$1 $tam >> ./$1.dat
        i=$[$i+1]
        tam=$[$tam+$3]
done
else
echo "Error de argumentos"
fi
```

Figura 8: Script individual

```
#!/usr/bin/gnuplot

set xlabel "Tamanio del problema"
set ylabel "Tiempo (seg)"
set terminal png size 640,480
set output 'clasico.png'
plot 'clasico.dat' with lines
set output 'dyv.png'
plot 'dyv.dat' with lines
```

Figura 9: Script de gnuplot

```
#!/bin/bash
        echo "Compilando..."
        g++ -o clasico clasico.cpp -O3 &&
        g++ -o dyv dyv.cpp -O3 &&
        rm -f ./clasico.dat ;
        rm -f ./dyv.dat ;
        echo "Ejecutando algoritmo clásico..." ;
        ./individual.sh clasico 1000000 20000;
        echo "Ejecutando algoritmo DyV..." ;
        ./individual.sh dyv 1000 100;
        echo "Generando gráficas..." ;
        ./gnuplot.sh ;
```

Figura 10: Script conjunto