



ugr

Universidad
de Granada

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



DECSAI

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2017-2018

Índice

1. Descripción de la práctica	1
2. Algoritmos diseñados	1
3. Estudio de eficiencia	3
3.1. Eficiencia teórica	3
3.2. Eficiencia empírica	3
3.3. Eficiencia híbrida	6
4. Conclusiones	7
5. Anexo: scripts auxiliares	8

Índice de figuras

1. Algoritmo clásico	1
2. Algoritmo Divide y Vencerás	1
3. Medición de tiempos	2
4. Algoritmo clásico	4
5. Algoritmo Divide y Vencerás	5
6. Tiempo medio (s)	6
7. Eficiencia híbrida del algoritmo clásico	7
8. Eficiencia híbrida del algoritmo Divide y Vencerás	7
9. Script individual	8
10. Script de gnuplot	8
11. Script conjunto	9

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 lo 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 calcular el tiempo de ejecución, cada algoritmo se ejecutará mil veces y nos quedaremos con el tiempo promedio para rebajar el impacto sobre el tiempo de factores externos como la carga del sistema. Para estudiar la eficiencia, ejecutaremos cada código 25 veces con tamaños variables dependiendo de su orden de eficiencia.

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

2. Algoritmos diseñados

```
1  int elementoEnSuPosicion(const vector<int> v) {
2      for (int i = 0; i < v.size() ; i++)
3          if (v[i] == i)
4              return i;
5      return -1;
6  }
```

Figura 1: Algoritmo clásico

```
1  int elementoEnSuPosicion(const vector<int> v, const int ini, const int fin) {
2      if (ini == fin) {          //Caso base, vector de un solo elemento
3          if (v[ini] == ini)
4              return ini;
5          else
6              return -1;
7      }
8      else {                    //Dividimos en dos el vector y llamamos recursivamente
9          int mitad = (ini + fin) / 2;
10         int pos_izq = elementoEnSuPosicion(v, ini, mitad);
11         int pos_dcha = elementoEnSuPosicion(v, mitad + 1, fin);
12         if (pos_izq != -1)
13             return pos_izq;
14         else
15             return pos_dcha;
16     }
17 }
```

Figura 2: Algoritmo Divide y Vencerás

```

1      int pos;
2      for (int i = 0; i < 1000; i++) {
3          tantes = high_resolution_clock::now();
4          pos = elementoEnSuPosicion(myvector);
5          tdespues = high_resolution_clock::now();
6          tiempo = duration_cast<duration<double>>
7                  (tdespues - tantes);
8          acumulado += tiempo.count();
9      }
10     acumulado /= 1000;

```

Figura 3: Medición de tiempos

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 que sigue es la siguiente:

- **Caso base.** El vector consta de un solo elemento. Si se cumple que $v[i] = i$, devolvemos [la posición](#). En caso contrario, devolvemos [-1](#).
- 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 $O(n)$.

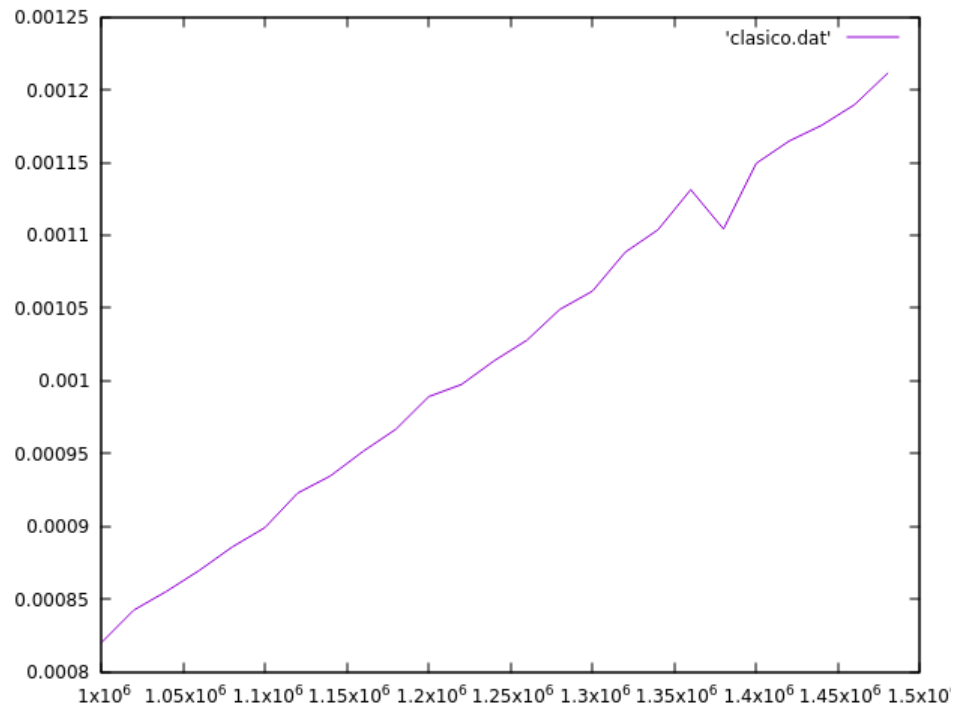
Por tanto, podemos adelantar que debido a los costes físicos de la recursión, el algoritmo *clásico* será mejor que el *Divide y Vencerás*.

3.2. Eficiencia empírica

Haciendo uso de [los scripts diseñados](#), ejecutamos los algoritmos con los siguientes tamaños:

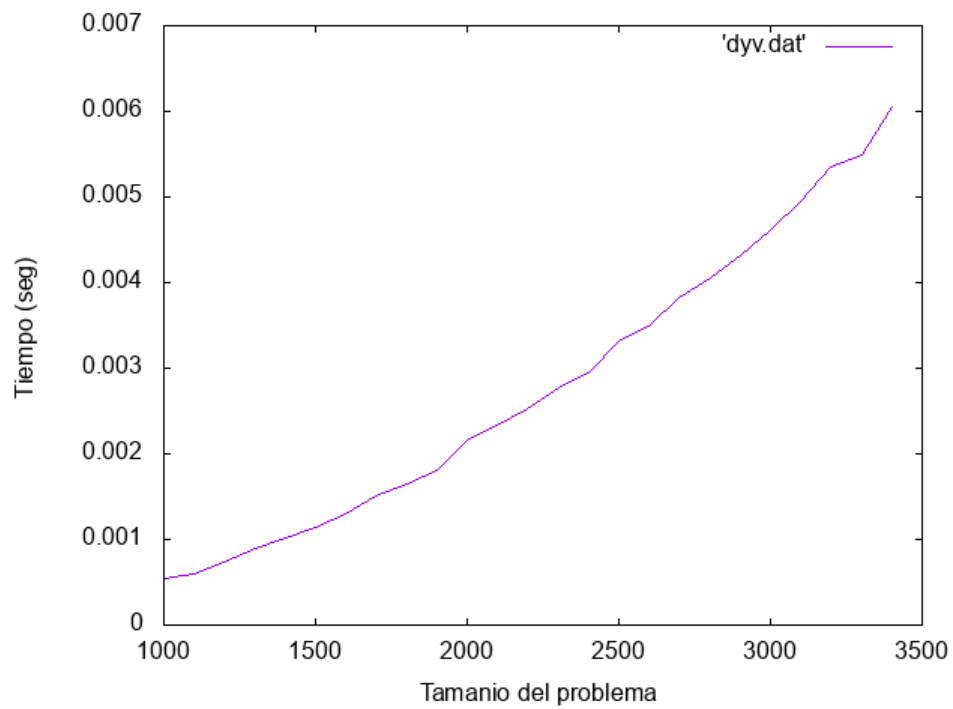
Algoritmo	Tamaño inicial	Tamaño final	Incremento
Clásico	1.000.000	1.480.000	20.000
Divide y Vencerás	1.000	3.400	100

Cuadro 1: Parámetros de ejecución de cada programa



Tamaño	Tiempo (s)
1.000.000	0,000819706
1.020.000	0,000842275
1.040.000	0,00085518
1.060.000	0,000869587
1.080.000	0,000885716
1.100.000	0,000899005
1.120.000	0,000922726
1.140.000	0,000934545
1.160.000	0,000951469
1.180.000	0,00096667
1.200.000	0,000988998
1.220.000	0,000997361
1.240.000	0,00101367
1.260.000	0,0010278
1.280.000	0,00104895
1.300.000	0,00106148
1.320.000	0,00108836
1.340.000	0,00110376
1.360.000	0,00113126
1.380.000	0,00110421
1.400.000	0,00114935
1.420.000	0,00116467
1.440.000	0,0011756
1.460.000	0,00118976
1.480.000	0,00121112

Figura 4: Algoritmo clásico



Tamaño	Tiempo (s)
1.000	0,000551096
1.100	0,000600369
1.200	0,000736414
1.300	0,000889537
1.400	0,00100999
1.500	0,00113876
1.600	0,00129452
1.700	0,00150525
1.800	0,00165492
1.900	0,00180961
2.000	0,00215437
2.100	0,00233609
2.200	0,00252987
2.300	0,00276466
2.400	0,00294883
2.500	0,00332321
2.600	0,00349556
2.700	0,00382512
2.800	0,00405147
2.900	0,00431329
3.000	0,00460976
3.100	0,00493946
3.200	0,00534553
3.300	0,00548893
3.400	0,00605759

Figura 5: Algoritmo Divide y Vencerás

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

Figura 6: 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 %

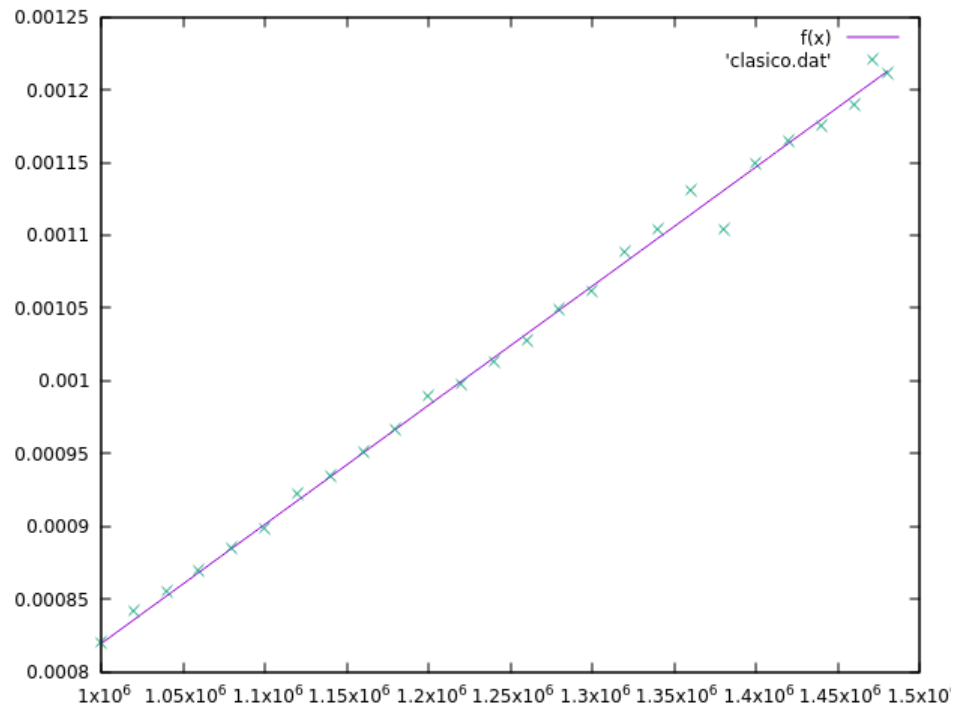


Figura 7: Eficiencia híbrida del algoritmo clásico

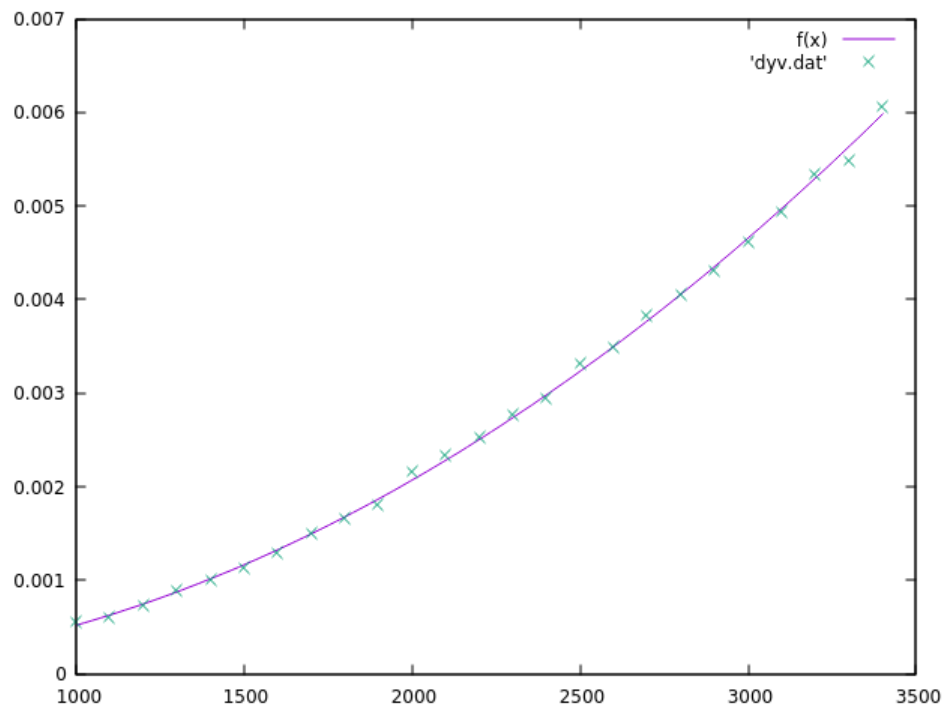


Figura 8: Eficiencia híbrida del algoritmo Divide y Vencerás

4. Conclusiones

SI HAY ELEMENTOS REPETIDOS AMBOS SIGUEN FUNCIONANDO

5. Anexo: scripts auxiliares

```
1  #!/bin/bash
2  if [ $# -eq 3 ]
3  then
4  i="0"
5  tam=$2
6  #Primer argumento: programa a ejecutar
7  #Segundo argumento: tamaño inicial
8  #Tercer argumento : incremento
9  while [ $i -lt 25 ]
10 do
11     ./$1 $tam >> ./$1.dat
12     i=$((i+1))
13     tam=$((tam+3))
14 done
15 else
16 echo "Error de argumentos"
17 fi
```

Figura 9: Script individual

```
1  #!/usr/bin/gnuplot
2
3  set xlabel "Tamano del problema"
4  set ylabel "Tiempo (seg)"
5  set terminal png size 640,480
6  set output 'clasico.png'
7  plot 'clasico.dat' with lines
8  set output 'dyv.png'
9  plot 'dyv.dat' with lines
```

Figura 10: Script de gnuplot

```
1  #!/bin/bash
2      echo "Compilando..."
3      g++ -o clasico clasico.cpp -O3 &&
4      g++ -o dyv dyv.cpp -O3 &&
5      rm -f ./clasico.dat ;
6      rm -f ./dyv.dat ;
7      echo "Ejecutando algoritmo clásico..." ;
8      ./individual.sh clasico 1000000 20000;
9      echo "Ejecutando algoritmo DyV..." ;
10     ./individual.sh dyv 1000 100;
11     echo "Generando gráficas..." ;
12     ./gnuplot.sh ;
```

Figura 11: Script conjunto