Algoritmos divide y vencerás

ALGORÍTMICA

UNIVERSIDAD DE GRANADA

JOAQUÍN ARCILA PÉREZ
LAURA LÁZARO SORALUCE
CRISTÓBAL MERINO SÁEZ
ÁLVARO MOLINA ÁLVAREZ

ÍNDICE

INTRODUCCIÓN

Esta práctica consiste en la resolución de dos problemas, uno de búsqueda y otro de inserción, tanto con un algoritmo clásico, como con la técnica divide y vencerás. Asimismo, se compara la eficiencia híbrida y empírica de ambos algoritmos.

Ejercicio 1: En este ejercicio se busca un número que coincida con el índice de la casilla en la que se encuentra en un vector ordenado de forma no decreciente y sin repeticiones. Tras construir los dos algoritmos mencionados, se comprueba si se pueden aplicar los mismos a un vector ordenado de la misma manera, esta vez con valores repetidos.

Ejercicio 2: Este ejercicio se basa en mezclar k vectores de n valores ordenados de manera no decreciente, para obtener un único vector ordenado. De nuevo hemos utilizado un algoritmo clásico y uno que utiliza la técnica de divide y vencerás para su resolución.

En ambos ejercicios, para la eficiencia teórica, hemos utilizado las técnicas de resolución de recurrencias vistas en clase. Para la empírica modificamos los códigos generadores de vectores y medimos el tiempo que tarda en ejecutarse cada algoritmo. Para evitar errores, hemos añadido un bucle que ejecuta el algoritmo 15 veces, para obtener una media del tiempo en lugar de un sólo valor. Por último, para la híbrida utilizamos gnuplot para ajustar las ecuaciones teóricas a los datos ordenidos, y obtener las constantes ocultas.

DESARROLLO

1.Ejercicio 1

1.1 Algoritmo obvio de búsqueda

```
void obvio1 (vector<int> v, int nelementos) {
   bool sigue = true;

   for (int i = 0; (i < nelementos) && (sigue); i++) {
      if (v[i] == i) {
         sigue = false;
      }
   }
}
```

Este bucle for recorre todos los elementos del vector empezando por la casilla 0. Va comparando cada elemento con su índice. Es por tanto un sumatorio desde i=0 hasta n-1 de a (constante).

En el momento en el que alguno coincida, la variable "sigue" se pone a false y por tanto se sale del bucle.

Resolviendo el sumatorio anterior, obtenemos la función polinómica an. Por lo tanto tenemos que $T(n) \in O(n)$.

1.2 Algoritmo divide y vencerás de búsqueda

```
void DyV1 (vector<int> v, int ini, int fin) {

 bool encontrado =false;

 while(ini<=fin && !encontrado) {
  int m=(fin+ini)/2;
  if(v[m]==m) {
   encontrado=true;
  }
  else if (v[m]<m) {
   ini=m+1;
  }
  else
   fin=m-1;
 }
}
```

Este bucle while comienza a trabajar con el vector original completo. Encuentra el valor del medio y lo compara con su índice. En cada vuelta, el vector se reduce a la mitad, es decir, se actúa como si se hubiese recorrido la mitad del vector restante, dando el bucle log(fin-ini) vueltas. Tomando fin=n e ini=0, se dan log(n) vueltas. Así, $T(n) \in O(\log(n))$.

Si el índice es mayor, se ejecuta de nuevo el bucle, pero esta vez trabajando sólo con la mitad derecha del vector original.

Si el índice es menor, se trabaja con la mitad izquierda. Y así sucesivamente, parando cuando se encuentre una coincidencia.

Como se puede observar, en el caso en que el índice es menor que el valor que contiene, nos quedamos con la mitad izquierda del vector, pues, al estar ordenado de forma creciente por enteros que no se repiten, los valores irán creciendo de la misma forma o más rápidamente que el índice, luego el este nunca alcanzará el valor que contiene. Ocurre lo mismo para el caso contrario.

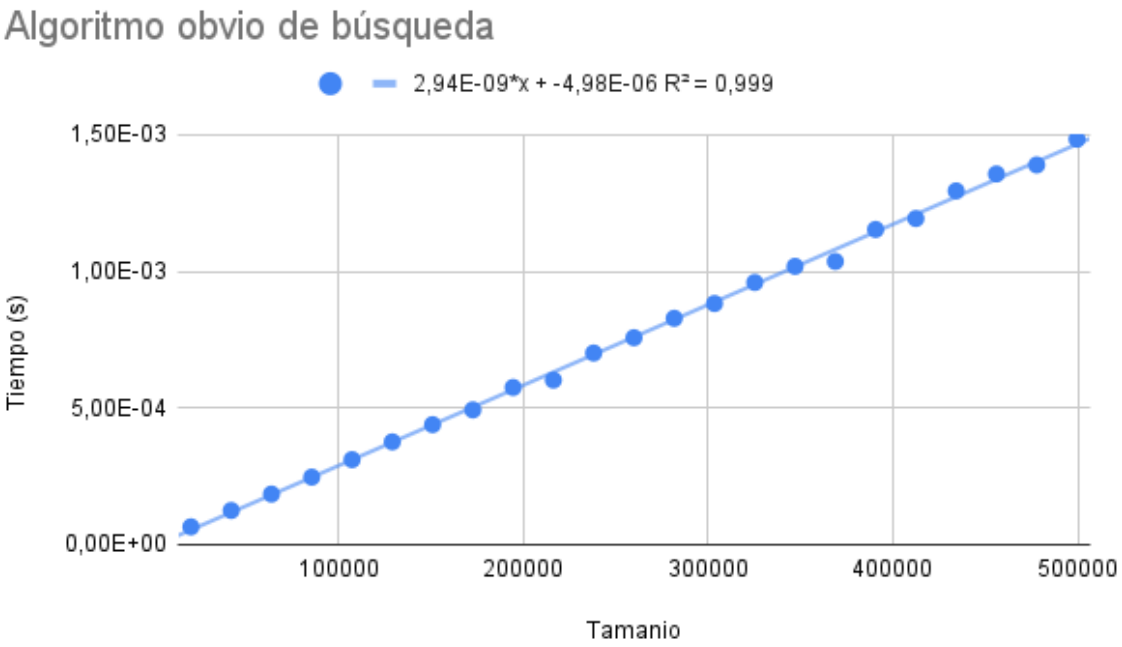
DESARROLLO

Para la eficiencia empírica de este ejercicio, hemos utilizado valores de 20000 a 500000 y hemos obtenido tiempos entre 6,51E-05 y 0,001483 para el algoritmo obvio; y entre 2,12E-05 y 0,000441 para el algoritmo de divide y vencerás. Escogimos ese rango de valores ya que comenzamos probando con valores hasta 200000 pero vimos que la ejecución era muy rápida y podíamos probar para valores más altos. Por otro lado, probamos también con valores más pequeños, y obtuvimos datos inestables, que fluctuaban mucho.

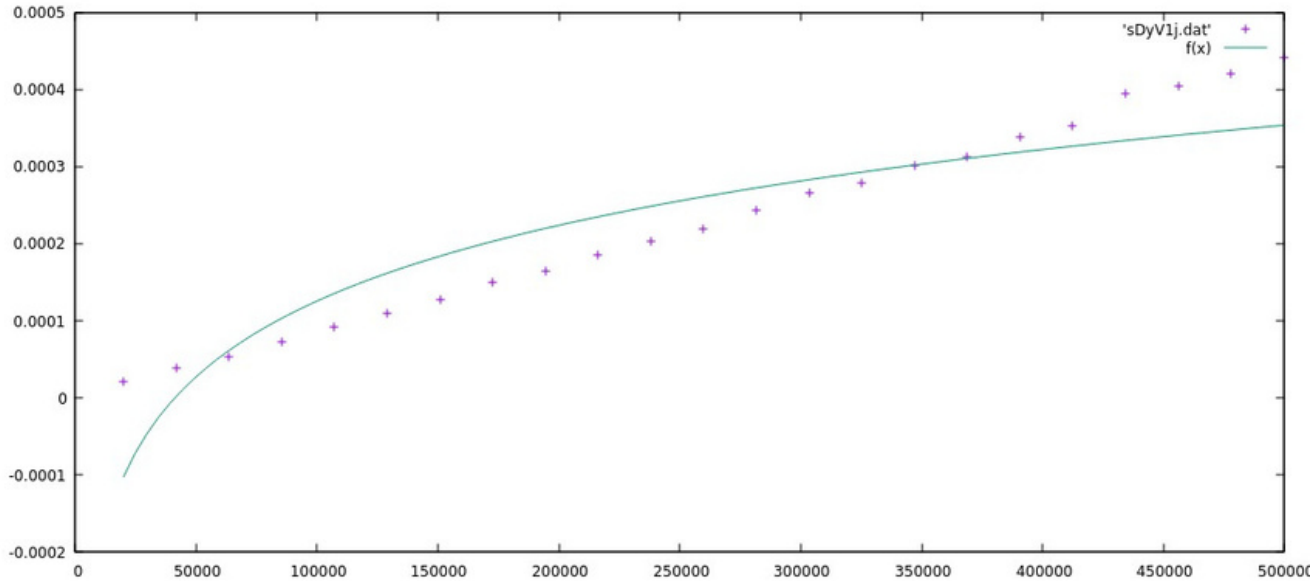
	Laura Obvio	Laura DyV	Joaquin Obvio	Joaquin DyV
20000	5,63E-05	8,87E-06	6,51E-05	2,12E-05
41800	0,000103	9,73E-06	0,000125	3,88E-05
63600	0,000157	1,14E-05	0,000185	5,31E-05
85400	0,000207	1,65E-05	0,000247	7,28E-05
107200	0,000277	2,07E-05	0,000311	9,16E-05
129000	0,000315	2,68E-05	0,000376	0,000109
150800	0,000369	4,92E-05	0,000439	0,000127
172600	0,000434	4,97E-05	0,000493	0,000149
194400	0,000479	5,58E-05	0,000575	0,000165
216200	0,000529	6,11E-05	0,000602	0,000185
238000	0,000592	5,67E-05	0,000701	0,000203
259800	0,000641	7,96E-05	0,000757	0,000219
281600	0,000699	7,12E-05	0,000828	0,000244
303400	0,000764	7,91E-05	0,000882	0,000266
325200	0,000837	8,83E-05	0,000959	0,000279
347000	0,000869	9,21E-05	0,001018	0,000302
368800	0,000933	0,000101	0,001036	0,000313
390600	0,001031	0,000131	0,001153	0,000338
412400	0,001057	0,000118	0,001193	0,000353
434200	0,001117	0,000129	0,001294	0,000395
456000	0,001157	0,000159	0,001356	0,000405
477800	0,001272	0,000147	0,001389	0,000419
499600	0,001332	0,000166	0,001483	0,000441

DESARROLLO

Para la eficiencia híbrida, hemos comparado los datos obtenidos en las ejecuciones con las respectivas funciones teóricas, para obtener las constantes ocultas, y ver con el coeficiente de regresión, cuanto de bien se ajustan los datos a lo esperado.



En el caso del algoritmo obvio de búsqueda, hemos comparando los datos con una función lineal, y vemos que se ajustan casi perfectamente, como era de esperar, pues su eficiencia teórica es O(n). La ecuacion con las constantes obtenidas es: 2,94E-09*x -4,98E-06.



En el caso del algoritmo divide y vencerás de búsqueda, hemos comparando los datos con una función logarítmica, y vemos que se ajustan un poco peor que en el caso anterior, lo que nos lleva a pensar que, en este caso, el algoritmo en ejecución no es el caso ideal, pues tiende a un orden lineal. La ecuación con las constantes ocultas obtenidas es: 2.98E-08*log2(x)+1,35E-07.

DESARROLLO

1.3 Algoritmo con repetición

Se nos pide obtener una solución al mismo problema de búsqueda, en un vector ordenado de forma no decreciente, pero esta vez con valores repetidos.

Como acabamos de ver en el apartado anterior, podemos descartar una de las mitades del vector, gracias a que sabemos que los índices crecen o decrecen más despacio de lo que hacen los números del vector. Ahora bien, si se pueden repetir los elementos, ya no podemos asegurarlo, por lo que no podríamos usar el mismo método para la resolución del nuevo problema. Por ejemplo:

- Supongamos que tenemos el vector V=[-5, -4, -2, 0, 5, 5, 7, 8, 9]. Entonces el algoritmo propuesto en el apartado anterior cogería V[4] y, al ser V[4]=5 > 4, obviaría la mitad derecha del vector y nunca llegaría al caso V[5]=5. Luego el algoritmo no es válido.

Para este caso, hemos decidido modificar el algoritmo usado anteriormente para la primera parte del ejercicio de la siguiente manera:

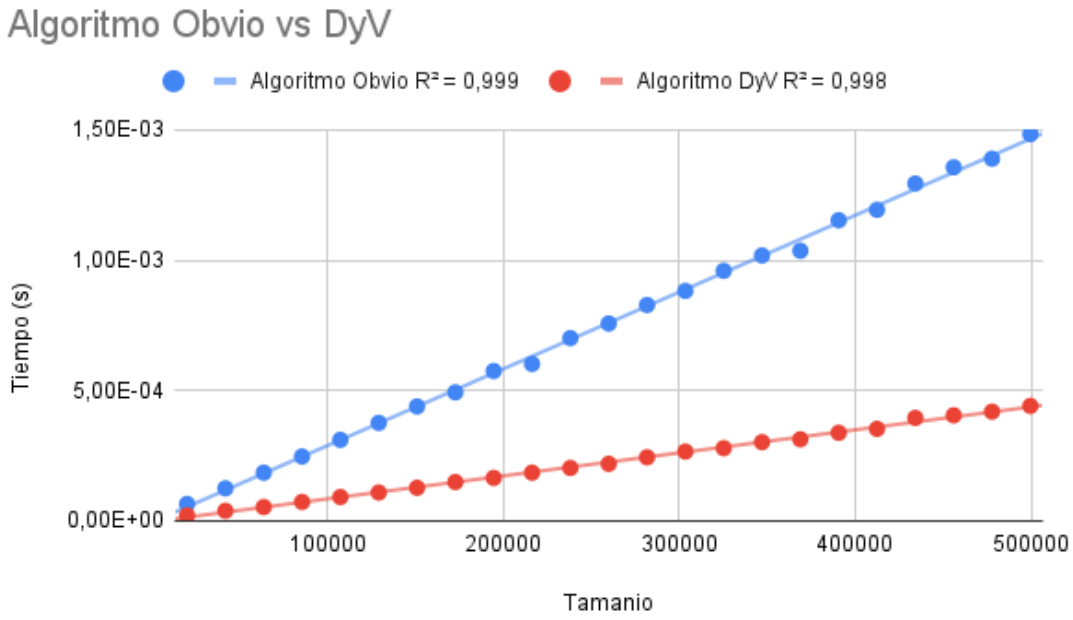
```
void DyV1 (vector<int> v, int ini, int fin) {
    bool encontrado =false;
    while(ini<=fin && !encontrado) {
        int m=(fin+ini)/2;
        if(v[m]==m) {
            encontrado=true;
        }
        else if (v[m]<m) {
            int i = m;
            while ((v[i-1]==v[i]) && !encontrado
                                    && i>=(ini+1)) {
                if (v[i-1] == i-1) {
                    encontrado = true;
                }
                else {
                    i--;
                }
            }
            ini=m+1;
        }
        else {
            int i = m;
            while ((v[i+1]==v[i]) && !encontrado
                                    && i<=(fin-1)) {
                if (v[i+1] == i+1) {
                    encontrado = true;
                }
                else {
                    i++;
                }
            }
            fin=m-1;
        }
    }
}
```

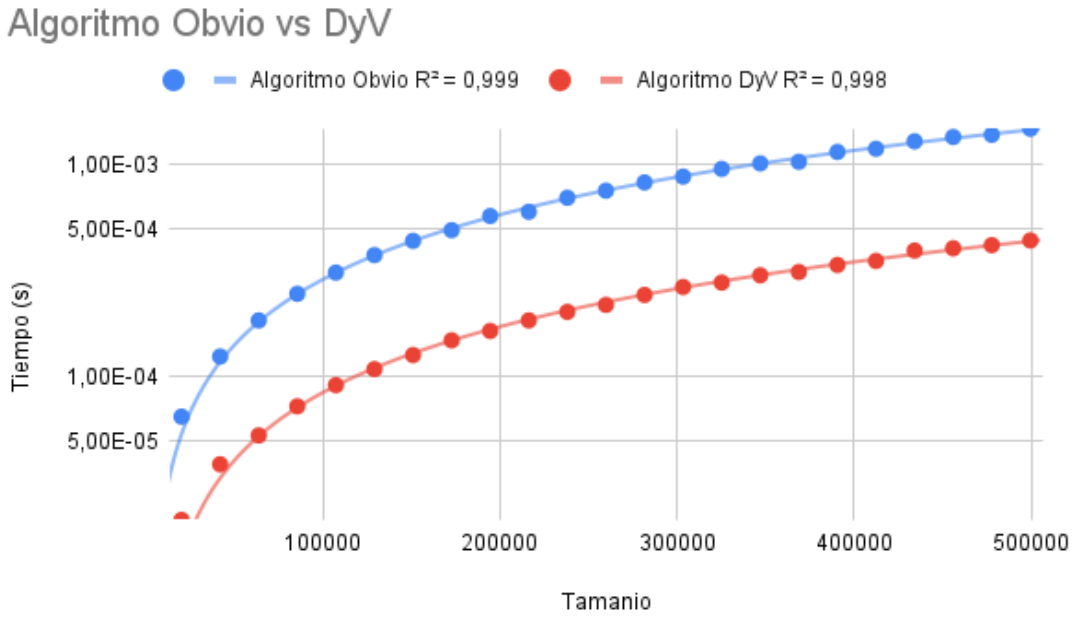
Esta es una de las muchas soluciones que tiene este problema, en la que se comprueba, dependiendo del caso, si el valor del vecino es igual al de la casilla actual. En el caso de que el valor del índice actual sea menor que el propio índice, se comprobará únicamente el valor del vecino de la izquierda, pues el valor actual seguirá siendo menor que el índice siguiente. Se trata de igual forma el caso opuesto.

En cuanto a la eficiencia, es claro que el algoritmo obvio es preferible, pues, en el peor caso, el algoritmo recién mostrado recorrerá el vector un mayor número de veces.

DESARROLLO

1.4 Comparación





En este gráfico podemos observar la diferencia entre ambos algoritmos. Como era de esperar, para el algoritmo obvio se han obtenido tiempos mayores que para el de divide y vencerás, sobretodo para tamaños más grandes de vectores. Lo hemos representado también con escala logarítmica para que se aprecie la diferencia para los vectores más pequeños.

DESARROLLO

2.Ejercicio 2

2.1 Algoritmo obvio

```
vector<int> mergevectors (vector<int> v1, vector<int> v2){
  auto it1=v1.begin(), it2=v2.begin();
  vector<int> result;
  while(it1!=v1.end()||it2!=v2.end()){
    if(it1==v1.end()){
      while(it2!=v2.end()){
        result.push_back(*it2);
        ++it2;
      }
    }

    else if(it2==v2.end()){
      while(it1!=v1.end()){
        result.push_back(*it1);
        ++it1;
      }
    }

    else if (*it1>=*it2){
      result.push_back(*it2);
      ++it2;
    }

    else {
      result.push_back(*it1);
      ++it1;
    }
  }

  return result;
}
```

El algoritmo recorre una sola vez ambos vectores, ya que funciona comparando cada vez las posiciones en las que se hallan ambos punteros, y avanzando el que tenga menor tamaño, así que la eficiencia del algoritmo es O(n), donde n es el tamaño de los vectores que estamos usando.

Primero cabe mencionar que hemos escrito una función mergevectors que usaremos para unir dos vectores y que servira para tanto el algoritmo obvio como para el nuestro. Como se observa, este algoritmo tiene eficiencia O(n).

DESARROLLO

2.1 Algoritmo obvio

```
vector<int> funcion_obvia(vector<vector<int>> &m){
   while(m.size()>1){
      m[0] = mergevectors(m[0],m[1]);
      m.erase(m.begin()+1);
   }
   return m[0];
}
```

Lo que rige la eficiencia del algoritmo obvio es el while, que comienza leyendo un vector de tamaño n, que luego es 2n, 3n... por lo que su eficiencia es $\Sigma i=1,k \ (k+1)=O(k^2)$, y como tenemos la función mergevectors que hemos dicho que tenía eficiencia O(n), la eficiencia del algoritmo completo es $O(k^2*n)$.

Como hemos mencionado, este algoritmo se basa en el algoritmo mergevectors para juntar dos vectores, y lo único que hace es ir juntando los dos primeros, esta combinación con el tercero y así sucesivamente. Esto hace que la eficiencia disminuya, ya que cada vez tiene que leer un vector más largo que el anterior.

DESARROLLO

2.2 Algoritmo divide y vencerás

```
vector<int> funcion(vector<vector<int>> &m){
   vector<vector<int>> aux;
   vector<int> solucion;
   for (int i=0;i<m.size()-1;i+=2){
      aux.push_back(mergevectors(m[i],m[i+1]));
   }
   if(m.size()%2 != 0){
      aux.push_back(m[m.size()-1]);
   }

   if(aux.size()>=2){
      solucion = funcion(aux);
   }
   else solucion=aux[0];
   return solucion;
}
```

Primero tenemos este bucle, que en la primera llamada a función tendrá eficiencia O(k+n). Si fuera como en el caso obvio, al duplicarse el tamaño de los vectores la eficiencia iría duplicándose, pero como a su vez el número de vectores se reduce a la mitad en cada llamada a funcion, la eficiencia se queda como O(k*n).

Dado que estamos ante una función recursiva que reduce cada vez a la mitad los vectores, tenemos que la función se repetirá un total de log(k) veces, por lo que obtenemos al final que la eficiencia es de O(k*log(k)*n), siendo más eficiente que el algoritmo obvio.

Tenemos ahora este algoritmo que hemos diseñado: es parecido al anterior, pero presenta la siguiente variación: es una función recursiva que toma un vector de vectores, que al llamarla lo que hace es juntar por parejas los vectores en un contenedor auxiliar, y añadir el ultimo si el numero de vectores es impar. Después, si el vector de vectores resulta tener un solo vector, lo devolverá como solución, si no, volverá a llamar a la función usando como argumento el vector de vectores auxiliar sobre el que hemos ido copiando los vectores. Así, solucionamos el principal problema que tenia el otro, ya que tenemos que realizar menos lecturas sucesivas de vectores demasiado largos.

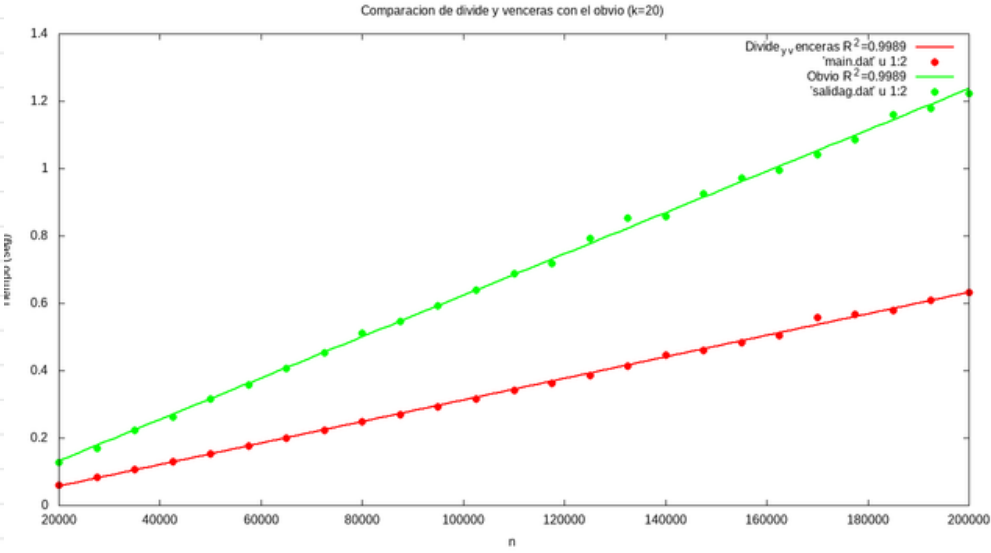
DESARROLLO

2.2 Algoritmo divide y vencerás

Para la eficiencia empírica de este ejercicio hemos tenido en cuenta dos situaciones; la primera con el numero de vectores fijo, y otro con el numero de elementos por vector fijo.

Primer caso. Con el número de vectores fijo, hemos utilizado valores de 20000 a 200000 para los elementos de los vectores, y hemos obtenido tiempos entre 0.0613041 y 0.632499 para el algoritmo obvio; y entre 0.613041 y 0.632499 para el algoritmo de divide y vencerás. Escogimos ese rango de valores ya que comenzamos probando con valores hasta 100000 pero vimos que los tiempos de ejecución eran muy rápidos y podíamos probar para valores más altos. Por otro lado, probamos también con valores más pequeños, y obtuvimos datos inestables, que fluctuaban mucho.

numero de elementos	tiempo Obvio	tiempo DyV
20000	0.127287	0.061304
27500	0.170293	0.084104
35000	0.223554	0.107288
42500	0.262377	0.130705
50000	0.316916	0.152339
57500	0.358815	0.176643
65000	0.406903	0.199607
72500	0.452975	0.224102
80000	0.510963	0.247771
87500	0.545793	0.270002
95000	0.592386	0.293336
102500	0.639692	0.315986
110000	0.687975	0.341033
117500	0.718726	0.363794
125000	0.792358	0.384994
132500	0.853879	0.413427
140000	0.859032	0.447382
147500	0.926081	0.460181
155000	0.971667	0.484515
162500	0.995849	0.505182
170000	1.040864	0.557585
177500	1.086037	0.566481
185000	1.159969	0.579076
192500	1.179592	0.608655
200000	1.223132	0.632499



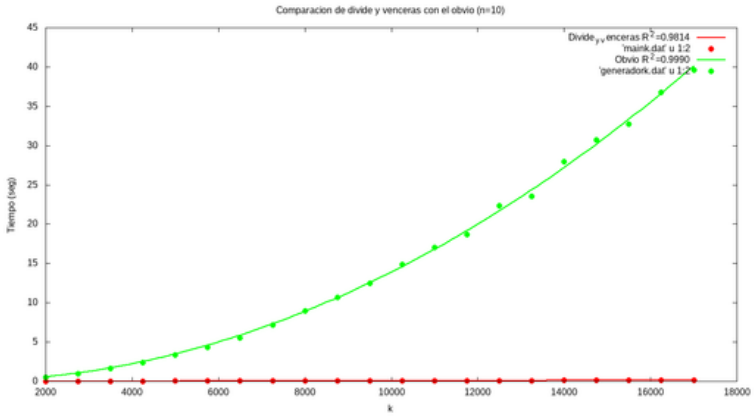
En este caso, como la eficiencia debía de ser lineal para ambos casos, vemos que la eficiencia empírica da el resultado que cabía esperar, aunque hay que mencionar que es notoria la diferencia de tiempo entre ambos algoritmos, aun con la misma teórica.

DESARROLLO

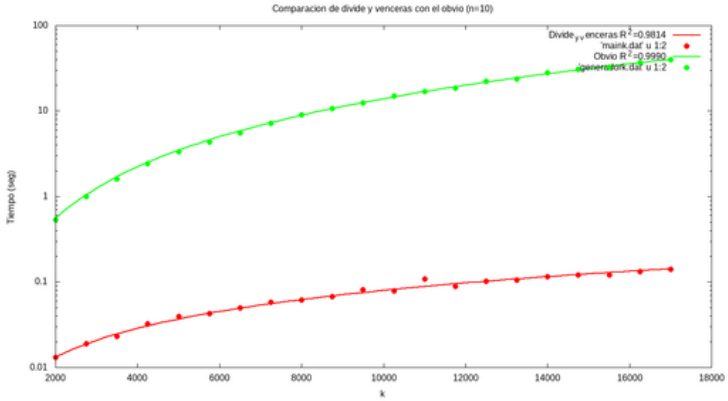
2.2 Algoritmo divide y vencerás

Segundo caso. Con el número de elementos de los vectores fijo, hemos utilizado valores de 2000 a 17000 para el número de vectores, y hemos obtenido tiempos entre 0.532055 y 39.6465 para el algoritmo obvio; y entre 0.0131647 y 0.141887 para el algoritmo de divide y vencerás.

numero de vectores	tiempo Obvio	tiempo DyV
2000	0.532055	0.013164
2750	0.996826	0.019154
3500	1.619431	0.023328
4250	2.416992	0.032576
5000	3.341922	0.039909
5750	4.350022	0.042761
6500	5.555929	0.049478
7250	7.164432	0.058376
8000	9.003652	0.061353
8750	10.725646	0.067239
9500	12.486865	0.081696
10250	14.890776	0.078781
11000	17.064574	0.108816
11750	18.671334	0.089727
12500	22.348323	0.102566
13250	23.576423	0.104751
14000	27.960975	0.116414
14750	30.733646	0.120707
15500	32.718686	0.121151
16250	36.756713	0.133052
17000	39.646523	0.141887



Como podemos ver, la gráfica en función del número de vectores se comporta como debería para el caso obvio, pero no se llega a apreciar para el nuestro por la diferencia en el tamaño de los datos



Para apreciar mejor las diferencias hemos introducido también una gráfica logarítmica, en la que se aprecia también que la del caso obvio tiene menos pendiente que la del caso nuestro.

CONCLUSIÓN

Para terminar con el trabajo, vamos a describir una breve conclusión.

Para el primer ejercicio hemos podido constatar la mejor eficiencia de la técnica divide y vencerás frente a un algoritmo clásico, gracias a la división del vector con el que trabajamos, que reduce el número de casillas por las que pasa el bucle. Esta diferencia se aprecia para todos los tamaños introducidos, aunque es más obvia para valores más grandes.

Hemos visto también que nuestro algoritmo de divide y vencerás no coincide exactamente con su eficiencia teórica esperada. Siendo en la práctica, tendiente a una función lineal.

Para el segundo ejercicio se aprecia lo mismo que en el ejercicio anterior, una clara mejoría del algoritmo de divide y vencerás frente al algoritmo obvio. Esta diferencia se nota severamente en todos los valores introducidos.

En este caso, los datos empíricos han coincidido con la eficiencia teórica hallada, a diferencia del ejercicio 1.

Como conclusión final podemos decir que el algoritmo divide y vencerás es siempre una mejor solución frente a los algoritmos basados en búsqueda lineal.

BIBLIOGRAFÍA

https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/
Información sobre la técnica divide y vencerás.