

PRACTICA 2

Algoritmos divide y vencerás



Manuel Contreras Orge
Raúl Durán Racero
Alberto Palomo Campos
Abel Rios Gonzalez
Salvador Romero Cortés

Índice:

Índice:	2
Problema	3
Algoritmo Básico $O(n)$	3
Algoritmo Divide y Vencerás:	5
Búsqueda Binaria $O(\log n)$	5
Comparación	8
Problema Supuesto	8
Porcentaje de participación:	9

Problema

El entero en su misma posición. Este es un algoritmo que, dado un vector de enteros de tamaño n , que está ordenado de manera estrictamente creciente (no es decreciente y sus valores son todos distintos entre sí), calcula si hay valores cuyo valor coincide con la posición del vector en la que se encuentra. ($v[i] = i$).

Vamos a resolver este problema usando dos tipos de algoritmos:

- *Algoritmo básico*
- *Algoritmo Divide y Vencerás*

Algoritmo Básico $O(n)$

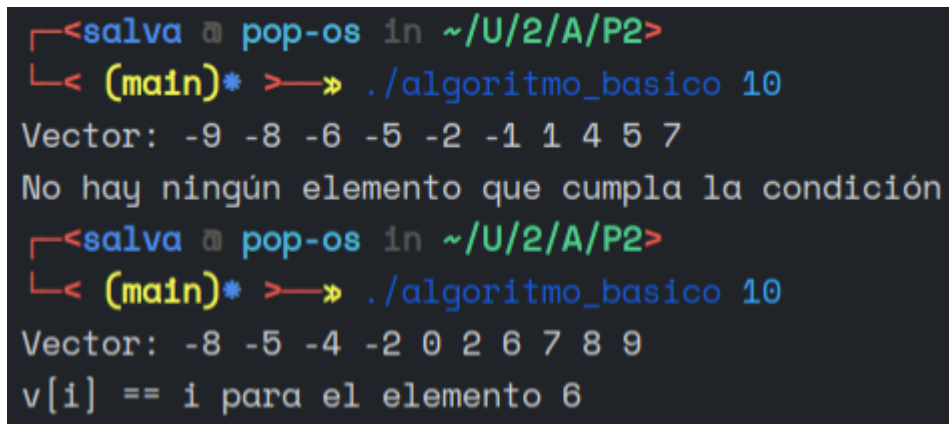
Este algoritmo, el más simple que existe para este problema, se compone de un bucle que se inicia en el primer elemento del vector ordenado (posición 0) y va hasta el primer elemento en el que coincida el índice con su valor o hasta el final, en el caso de que no coincida ningún elemento con su valor.

Al final, la función devuelve, en caso de que haya coincidencia, dicha posición (y por tanto su valor) y en el caso de que no la haya el valor -1.

Implementación del algoritmo:

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

Comprobamos su correcto funcionamiento:



```
└─<salva @ pop-os in ~/U/2/A/P2>  
└─< (main)* >─» ./algoritmo_basico 10  
Vector: -9 -8 -6 -5 -2 -1 1 4 5 7  
No hay ningún elemento que cumpla la condición  
└─<salva @ pop-os in ~/U/2/A/P2>  
└─< (main)* >─» ./algoritmo_basico 10  
Vector: -8 -5 -4 -2 0 2 6 7 8 9  
v[i] == i para el elemento 6
```

Cálculo de la eficiencia teórica:

El bucle realiza n iteraciones en el peor de los casos, que sería en el caso que el elemento coincidente con su índice no existe, es decir, no se cumple la condición $v[i] == i$ para ninguna posición. Al tiempo de ejecución constante del bucle lo llamamos “a” y al tiempo de ejecución constante externo al bucle lo llamamos “b”, obteniendo así la siguiente expresión:

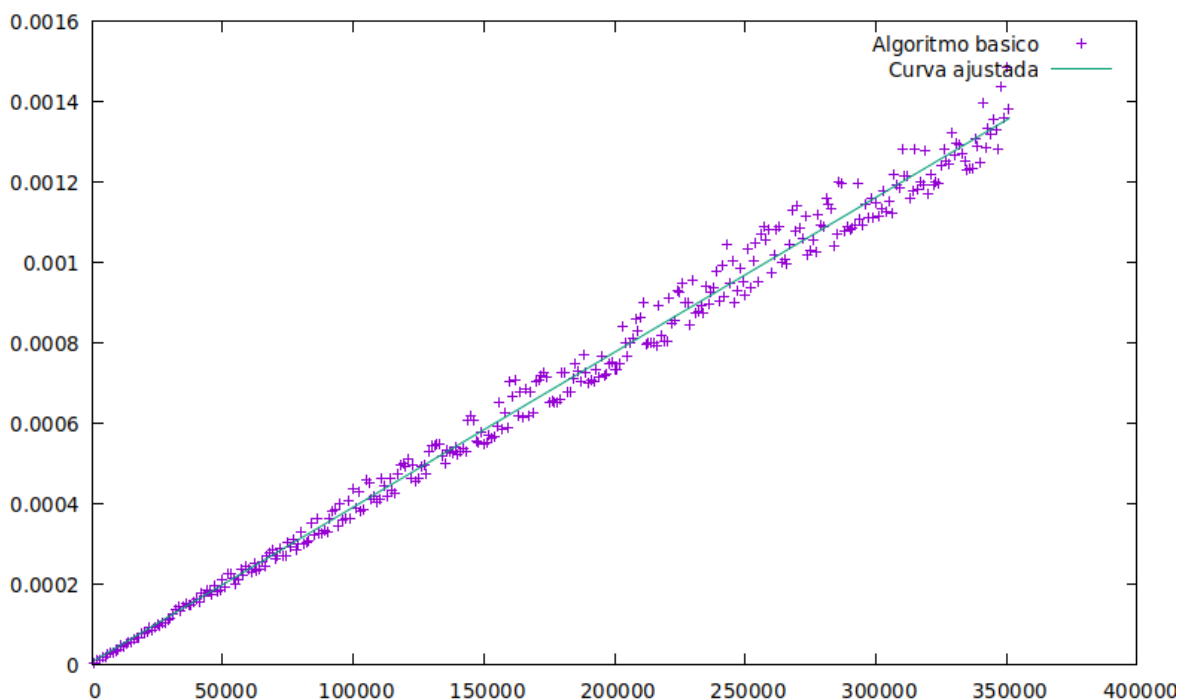
$$a \cdot n + b$$

Si eliminamos las constantes obtenemos, como era de esperar, que la eficiencia del algoritmo básico es del orden $O(n)$

```
int algoritmo_basico(vector<int> & v){
    int res = -1;  $O(1)$   $b$ 
    for (int i=0; i < v.size() && res == -1; ++i)  $O(1)$ 
        if (v[i] == i)  $O(1)$ 
            res = i;  $O(1)$ 
    return res;
}
```

$O(n)$ $O(n)$

Gráfica resultante de ejecutar la función:



Vemos que se ajusta con bastante precisión. También se aprecian muchos picos en la gráfica, pero pensamos que esto se trata de algo sin importancia puesto que sigue siendo dentro de un tiempo ínfimo, por lo que puede deberse a la gestión de recursos del Sistema Operativo.

Los errores y constantes ocultas se pueden ver en la siguiente imagen:

Constantes ocultas de la gráfica ajustada:

Final set of parameters		Asymptotic Standard Error	
=====		=====	
a	= 3.85039e-09	+/- 1.857e-11	(0.4823%)
b	= 6.74952e-06	+/- 3.771e-06	(55.87%)
correlation matrix of the fit parameters:			
	a	b	
a	1.000		
b	-0.867	1.000	

Algoritmo Divide y Vencerás:

Búsqueda Binaria $O(\log n)$

Es la aplicación más sencilla de Divide y Vencerás. Consiste en dividir repetidamente el vector ordenado en 2 partes y seleccionar la parte en la que se puede encontrar el resultado según un criterio.

Para obtener dicho criterio hay que plantearse si se puede determinar que en cierto intervalo se encuentra la solución sin explorar el intervalo y utilizando sus extremos en su lugar. Partiendo de la base de que el vector está ordenado y de que no tiene repeticiones, conforme recorremos el vector hacia la derecha siempre vemos un incremento de al menos 1 respecto al valor anterior (diferencia de 1 o más entre $v[i]$ y $v[i-1]$), y por otra parte el índice crece siempre de 1 en 1. Si analizamos cada posición del vector como una diferencia entre el valor y el índice que tenga en cuenta la dirección de la diferencia ($v[i] - i$), aplicando el razonamiento anterior llegamos a la conclusión de que dicha diferencia sólo puede ir en aumento o mantenerse. Si además tenemos en cuenta que un valor y su índice son iguales cuando su diferencia es 0, sabemos que si la diferencia es menor que 0 en un punto, el índice que cumple la condición del problema únicamente puede encontrarse más adelante, ya que más atrás la diferencia se alejaría de 0. Esto se aplica también en el caso contrario, si la diferencia en un punto es mayor que 0, ir más adelante en el vector incrementaría o se mantendría, en el mejor caso, la diferencia, por lo que tampoco se podría encontrar diferencia 0.

El algoritmo por lo tanto consiste en consultarla diferencia en el punto medio de un intervalo. Si es mayor que 0 es imposible que el índice buscado esté a la derecha así que eliminamos la mitad derecha y repetimos el proceso para la mitad restante. Si es menor que 0 ocurre lo mismo pero al contrario. Y si es 0 entonces el centro es el índice en el que se cumple la condición del problema. Si realizamos este proceso para el vector entero y no encontramos diferencia 0, es que no existe ningún valor en el vector que satisface dicha condición.

Implementación del algoritmo:

```
int algoritmo_dyv(vector<int> &v) {
    int i_centro = v.size() / 2;
    int dif_centro = v[i_centro] - i_centro;
    int i_izqda = 0, i_dcha = v.size() - 1;
    bool puede_estar = true;

    while (i_izqda <= i_dcha && puede_estar) {
        if (dif_centro > 0){
            i_dcha = i_centro - 1;
        }
        else if (dif_centro < 0){
            if(i_izqda < v[i_izqda])
                puede_estar = false;
            else
                i_izqda = i_centro + 1;
        }
        else
            return i_centro;

        i_centro = (i_izqda + i_dcha) / 2;
        dif_centro = v[i_centro] - i_centro;
    }
    return -1;
}
```

Comprobamos su correcto funcionamiento:

```
<salva @ pop-os in ~/U/2/A/P2>
└─< (main)* >─» ./algoritmo_dyv 10
Vector: -9 -6 -5 -3 2 4 5 7 8 9
v[i] == i para el elemento 7
└─<salva @ pop-os in ~/U/2/A/P2>
└─< (main)* >─» ./algoritmo_dyv 10
Vector: -9 -8 -7 -6 -2 1 2 4 6 7
No hay ningún elemento que cumpla la condición
```

Cálculo de la eficiencia teórica:

El bucle hace $\log(n)$ iteraciones en el peor de los casos ya que siempre va dividiendo sus iteraciones a la mitad hasta llegar a 1 ($N \rightarrow N/2 \rightarrow N/4 \rightarrow \dots \rightarrow 1$), cada iteración está acotada por un tiempo constante “a” y eso se suma al tiempo constante previo al bucle, tiempo al que llamamos “b”. Con esto obtenemos la expresión:

$$a \cdot \log(n) + b$$

Si eliminamos las constantes obtenemos el siguiente orden: $O(\log(n))$

```

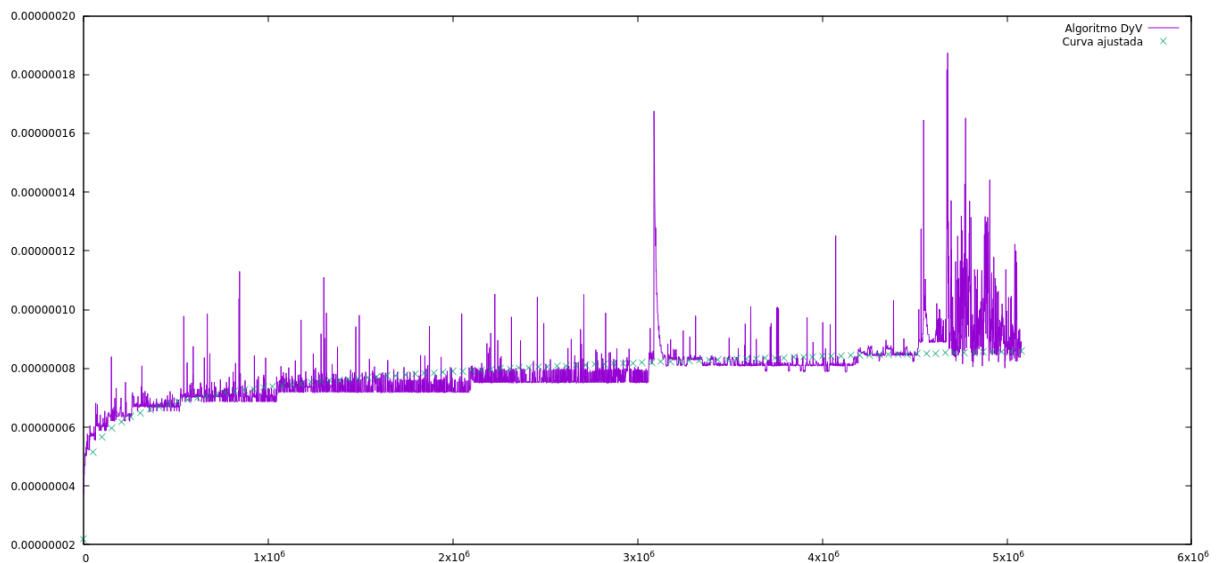
int algoritmo_dyv(vector<int> &v) {
    int i_centro = v.size() / 2;  $O(1)$ 
    int dif_centro = v[i_centro] - i_centro;  $O(1)$ 
    int i_izqda = 0, i_dcha = v.size() - 1;  $O(1)$ 
    bool puede_estar = true;  $O(1)$ 
    while (i_izqda <= i_dcha && puede_estar) {
        if (dif_centro > 0) {  $O(1)$ 
            i_dcha = i_centro - 1;  $O(1)$ 
        }
        else if (dif_centro < 0) {  $O(1)$ 
            if (i_izqda < v[i_izqda])  $O(1)$ 
                puede_estar = false;  $O(1)$ 
            else
                i_izqda = i_centro + 1;  $O(1)$ 
        }
        else
            return i_centro;  $O(1)$ 

        i_centro = (i_izqda + i_dcha) / 2;  $O(1)$ 
        dif_centro = v[i_centro] - i_centro;  $O(1)$ 
    }
    return -1;  $O(1)$ 
}

```

Handwritten annotations: A red bracket labeled 'b' groups the initialization lines. A red bracket labeled 'a' groups the while loop body. A green bracket on the right side of the while loop is labeled $O(\log(n))$.

Ajuste híbrido de la función:



En esta gráfica se pueden también observar picos, como en la anterior, siendo estos picos del tamaño no superior a 0.00000008 segundos, pero como los tiempos de ejecución del

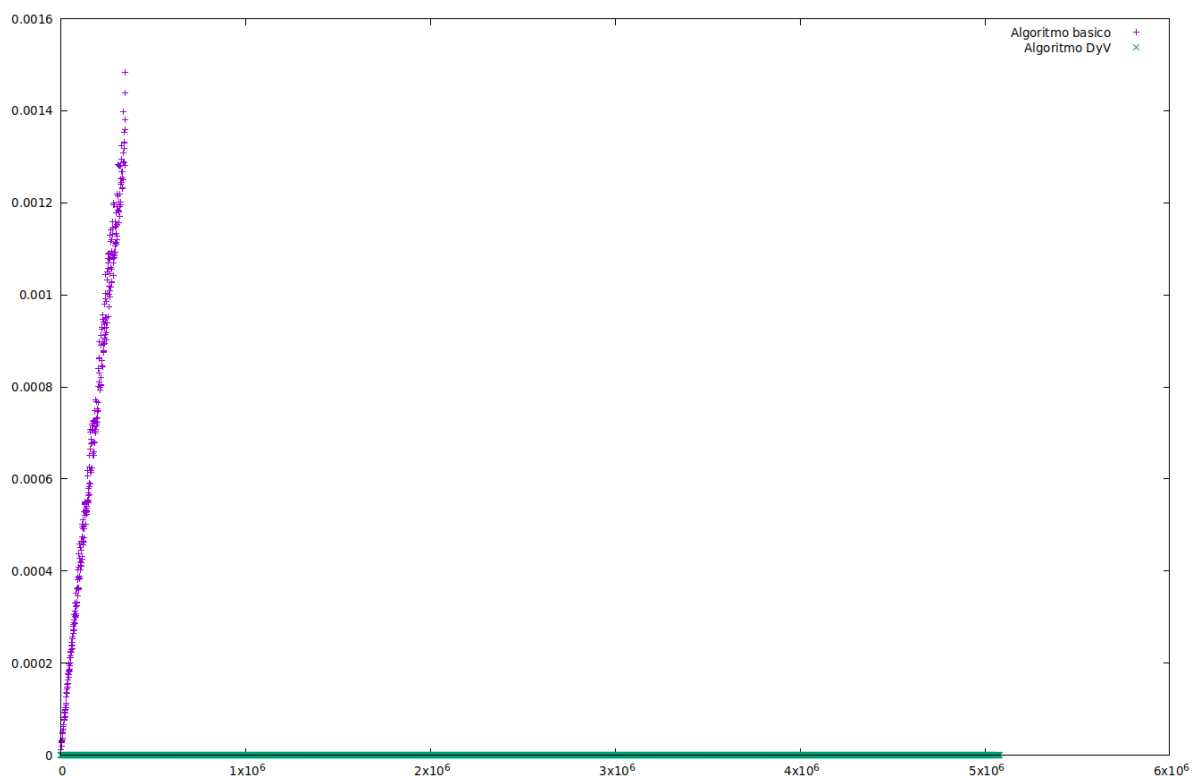
algoritmo son muy pequeños, aparentan ser mayor. Estos picos, como los de la anterior gráfica los atribuimos a la gestión de recursos del SO.

Vemos además que se ajusta con bastante acierto a la eficiencia teórica esperada ($O(n)$). Esto lo podemos ver en los errores del ajuste de la siguiente imagen.

Constantes ocultas de la gráfica ajustada:

Final set of parameters		Asymptotic Standard Error	
=====		=====	
a	= 5.18114e-09	+/- 7.454e-11	(1.439%)
b	= -2.95503e-08	+/- 1.557e-09	(5.268%)
correlation matrix of the fit parameters:			
	a	b	
a	1.000		
b	-0.998	1.000	

Comparación



Como se puede apreciar, la diferencia de ejecución entre las dos soluciones es muy grande, porque mientras la primera (la solución básica) crece de manera lineal a una velocidad normal, el segundo (el de divide y vencerás) crece de manera logarítmica con unos valores muy cercanos al cero, tanto que no se pueden apreciar bien en la gráfica. Como se puede apreciar, la diferencia entre ambos algoritmos es abismal.

Problema Supuesto

En el caso de que los números enteros puedan repetirse, este algoritmo no es válido, ya que se basa en la diferencia del incremento de los índices del vector con respecto al de los valores de dicho vector. Si un vector contiene números repetidos, esta diferencia puede no solo crecer o mantenerse, sino que también puede decrecer y por lo tanto no se puede determinar si en una mitad puede encontrarse el valor o no, exactamente tal y como ocurriría en caso de que los vectores no estuviesen ordenados.

Ejemplo:

$v = [-9, -6, -5, -3, 2, 4, 8, 8, 8, 9]$

La ejecución sería la siguiente:

1. Se fija en el valor central: $v[5]=4$
 $4-5<0$
Ahora el extremo izquierdo será el índice $5+1=6$ (el 5 queda considerado con una comprobación directa)
2. Consideramos sólo $v=[8, 8, 8, 9]$ -> Se fija en el centro: $i=(9+6)/2=7$
3. $v[7]=8$
 $8-7>0$
El algoritmo comprobará a la izquierda de $v[7]$, ya que como la diferencia entre el valor y el índice es positiva, supone que no puede haber ningún número coincidente a la derecha.
4. Ahora comprobará $v[6]=8$, siendo que $8-6>0$, así que devolverá que no hay número coincidente

Como podemos observar, no nos sería de utilidad el algoritmo de tipo Divide y Vencerás, *Búsqueda binaria*, al menos no con esta implementación.

Porcentaje de participación:

Manuel Contreras Orge: 20%

Raúl Durán Racero: 20%

Alberto Palomo Campos: 20%

Abel Rios González: 20%

Salvador Romero Cortés: 20%