Práctica 2 Algorítmica
Algoritmos Divide y vencerás

Realizado por:
Sergio Cervilla Ortega
Daniel Díaz Pareja
Marina Hurtado Rosales
Adrián Morente Gabaldón
Joaquín Rodríguez Claverías

Índice

1. Descripción del problema.

El problema propuesto en esta práctica es el siguiente:

Dado un vector ordenado (de forma no decreciente) de números enteros v, todos distintos, el objetivo es determinar si existe un índice i tal que v[i] = i y encontrarlo en ese caso. Diseñar e implementar un algoritmo “divide y vencerás" que permita resolver el problema. ¿Cuál es la complejidad de ese algoritmo y la del algoritmo “obvio" para realizar esta tarea? Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Supóngase ahora que los enteros no tienen por qué ser todos distintos (pueden repetirse). Determinar si el algoritmo anterior sigue siendo válido, y en caso negativo proponer uno que sí lo sea. ¿Sigue siendo preferible al algoritmo obvio?

2. Implementación del algoritmo obvio.

- Código:

```
// Algoritmo básico para encontrar un número i tal que v[i] = i
// Simplemente compara cada valor del vector con su indice.
int finder(int begin, int end, const vector<int> myvector){
        bool rompe = false; // Variable que nos permite salir del bucle
                            // si encontramos una solución
        int res = -1;
        for(int i=begin; i<=end && !rompe; i++)
                if(i == myvector[i]){
                        res = i;
                        rompe=true;
                }
        // Devolvemos el resultado, si es -1, no se ha encontrado ninguno.
        return res;
}
```

- Funcionamiento: El algoritmo consiste en simplemente aplicar lo que dice el enunciado del problema, un bucle for que recorre el vector y comprueba el valor de cada casilla con su indice (v[i] = i]). Si encuentra una coincidencia, la variable booleana “res” hace que el bucle pare de iterar. Finalmente se devuelve el resultado.

3. Implementación DyV del algoritmo.

- Código:

```
// Algoritmo DyV para encontrar un número i tal que v[i] = i
int finder(vector<int> arr, int low, int high)
{
    // Condición de parada del bucle, que el tope mayor sea menor o igua al tope menor del vector
    // (es decir, que el vector que se esta analizando sea de 1 casilla)
    while(low<=high)
    {
        int middle = (low+high)/2; // Variable que contiene la posición intermedia del vector

        // Dividimos el vector por la mitad y comprobamos primero que que se encuentre la solución
        // en dicha posición (v[mitad]=mitad)
        if(arr[middle] == middle)
            return middle;

        // Si no se ha encontrado la solución y el valor del vector es mayor que su indice, podemos
        // asegurar que el valor buscado solo puede encontrarse en la mitad inferior (ya que el
        // vector está ordenado), por lo que el tope superior será ahora el medio anterior - 1
        else if (arr[middle] > middle)
            high = middle - 1;

        // En el caso que queda, es decir que v[i] < i, sabemos que el valor buscado solo puede estar
        // en la mitad superior, por lo que ahora es el tope inferior el que se actualiza.
        else
            low = middle + 1;
    }

    // En caso de que no se haya encontrado solución, devolvemos un -1
    return -1;
}
```

- Funcionamiento: El funcionamiento del algoritmo está basado en la técnica de búsqueda binaria. Primero, se entra en un bucle que realizará lo siguiente hasta que tengamos un vector de tamaño 1:

  Se calcula el medio del vector. Si dicho medio no cumple que v[medio] = medio y v[medio] > medio, solo es posible que el valor buscado se encuentre en la mitad inferior (ya que el vector está ordenado). Entonces dividimos el vector y nos quedamos con la mitad izquierda, es decir, el nuevo “high” del vector será el medio - 1.

  En el caso que nos queda, que v[medio] < medio, solo es posible que el valor buscado se encuentre en la mitad superior. Esta vez entonces el nuevo “low” del vector será el medio +1.

  Una vez decidida la mitad con la que nos vamos a quedar, realizamos la misma operación sucesivamente hasta encontrar la solución o hasta que nos quedemos sin casillas, en cuyo caso se devuelve -1.

  Por contra, este algoritmo no funciona en el caso de que haya elementos repetidos en el vector. Supongamos el vector 1 2 3 5 5 5. Según el algoritmo anterior, la mitad sería la posición 3, por lo que v[3] = 5, es decir se cumple que v[i] > i, por lo que se

descartaría la mitad superior que contiene soluciones. Para ello, hemos implementado otro algoritmo divide y vencerás basado en recurrencia, que divide el vector por la mitad, y a su vez el vector resultante a la mitad, y así sucesivamente hasta encontrar el valor de la solución. Primero comprueba la primera mitad del vector y si no la encuentra, busca en la segunda.

- Código:

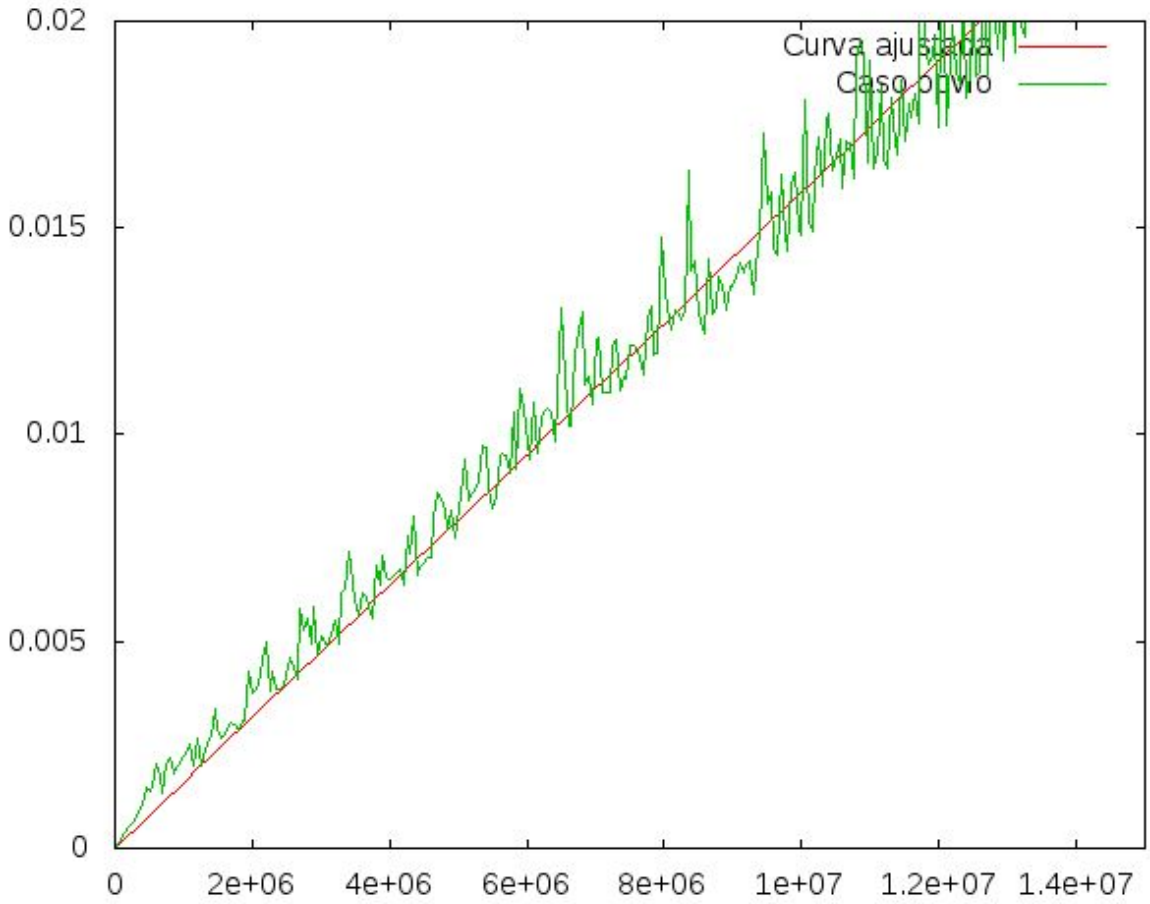
```
int umbral=750; // Umbral en el que se deja de aplicar recursividad
                // y se aplica un algoritmo de resolución básico.

// Función que encuentra un número i tal que v[i] = i
int finder(int begin, int end, const vector<int> myvector){
    int medio=(begin+end)/2; // Mitad del vector. Nos servirá para dividirlo en dos
                             // y así reducir el tamaño del problema

    int res=-1; // Variable donde se almacena el resultado, si es -1
                // no se ha encontrado coincidencia

    // Si el vector no es más pequeño que el umbral, seguimos aplicando recursividad.
    if((end-begin)>umbral){
        // Comprobamos que la casilla del medio sea solución.
        if(medio == myvector[medio])
            res = medio;
        // Si no lo es, investigamos la primera mitad del vector
        else
            res = finder(begin, medio-1, myvector);

        // Si en la primera mitad del vector no se ha encontrado solución,
        // investigamos en la segunda mitad.
        if (res == -1)
            res = finder(medio+1, end, myvector);
    }
    // Si el vector es más pequeño que el umbral aplicamos el algoritmo
    // básico de la solución, simplemente comparar cada indice del vector
    // con su valor (v[i] == i)
    else{
        bool rompe=false; // Variable que nos permite salir del bucle
                          // si encontramos una solución
        for(int i=begin; i<=end && !rompe; i++){
            if(i == myvector[i] || (i*(-1))==myvector[i]){
                res = i;
                //cout << endl << "Encontrado: v[" << i << "] = " << i << endl;
                rompe=true;
            }
        }
    }

    // Devolvemos el resultado, si es -1, no se ha encontrado ninguno.
    return res;
}
```

4. Análisis de eficiencia.

4.1 Algoritmo obvio.

- Eficiencia híbrida:

Este algoritmo es de orden de eficiencia O(n), ya que lo que realiza es recorrer un vector de n elementos, es decir, el tiempo crece en función de n, el tamaño del vector, un número constante.

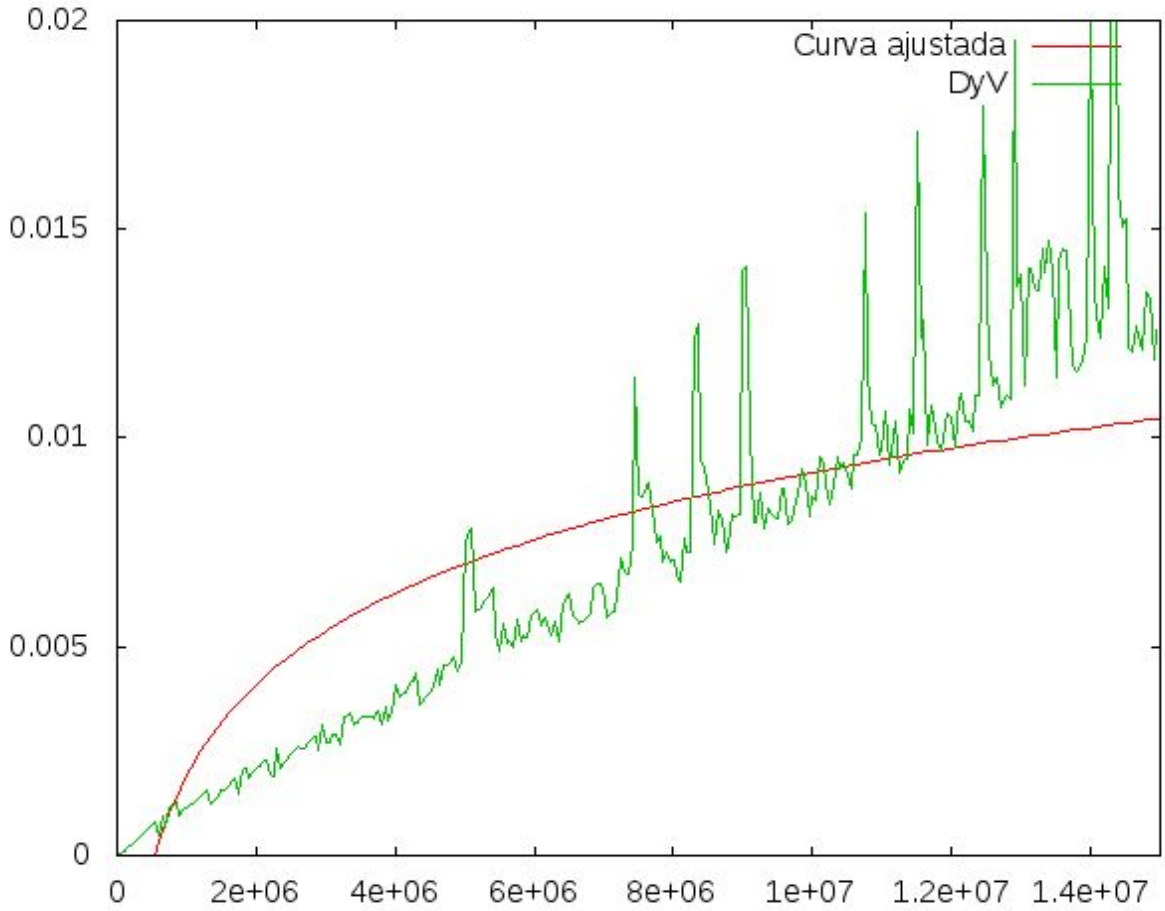


Constantes ocultas obtenidas:

Final set of parameters		Asymptotic Standard Error	
a	= 7.35205e-08	+/- 4.346e-09	(5.911%)
b	= 0.999998	+/- 0.05009	(5.009%)

4.2 Algoritmo DyV.

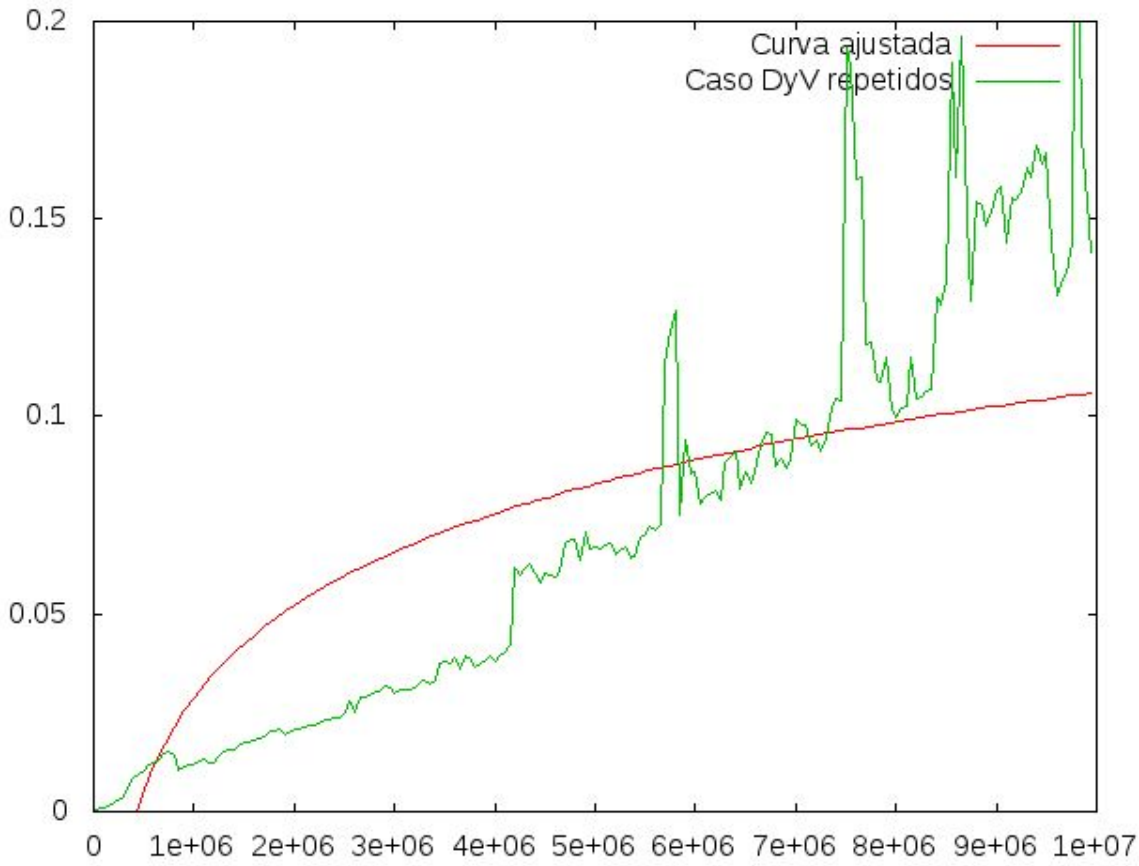
- Eficiencia híbrida para el caso de valores no repetidos:



Constantes ocultas obtenidas para la función ajustada: f(x) = a*log(b*x)

Final set of parameters		Asymptotic Standard Error
a	= 0.00316412	+/- 0.0001533 (4.844%)
b	= 1.8204e-06	+/- 2.233e-07 (12.27%)

- Eficiencia híbrida para el caso de valores repetidos:

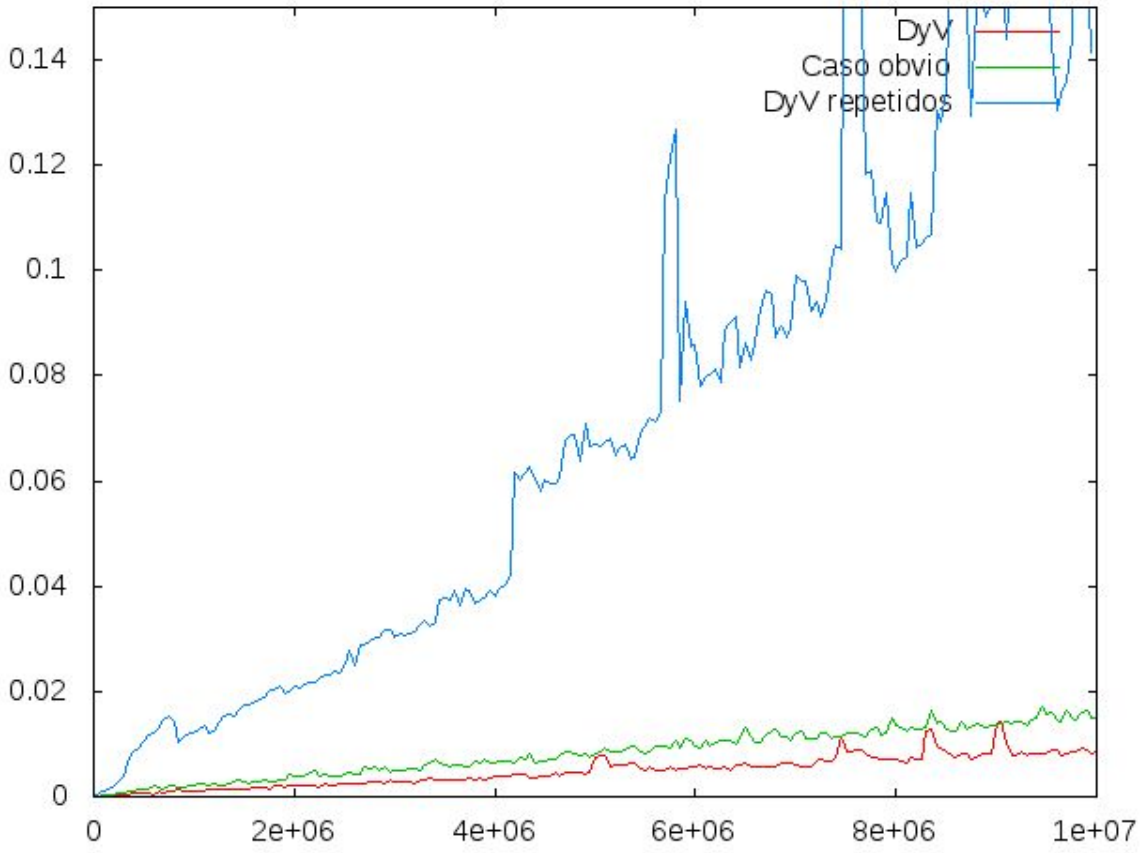


Constantes ocultas obtenidas para la función ajustada: f(x) = a*log(b*x)

Final set of parameters		Asymptotic Standard Error	
a	= 0.0335681	+/- 0.0023	(6.851%)
b	= 2.36393e-06	+/- 3.901e-07	(16.5%)

4.3 Comparación.

- Imagen de la comparación entre los tres algoritmos:



Como podemos apreciar, el mejor algoritmo es el divide y vencerás en el caso de vectores con valores no repetidos, cuyo orden de eficiencia es el mejor: O(log n).

En segunda posición se encuentra el algoritmo evidente, que es de orden lineal y además, su constante oculta dominante es muy pequeña (del orden de 10^{-8}), por lo que a efectos prácticos no es tan ineficiente como pudiera parecer.

El peor de los casos es el algoritmo divide y vencerás en el caso de que los vectores contengan valores repetidos, ya que la recursividad hace que el costo de descomposición en subproblemas y recomposición de las soluciones sea muy grande, por lo que en la práctica se obtienen tiempos peores que en algoritmo evidente.