# -ALGORITMOS DIVIDE Y VENCERÁS-

Álvaro Fernández García
Alexandra-Diana Dumitru
Ana María Romero Delgado
Germán Castro Montes
Marino Fajardo Torres

## ÍNDICE



## 1. DESCRIPCIÓN DEL PROBLEMA

El problema propuesto en la práctica es el siguiente:

**Señales regulares**

Dado un vector que contiene una secuencia de enteros, se pretende localizar subsecuencias consecutivas de valores xi , . . . , xj que cumplan la propiedad de regularidad siguiente:

$$|x_k - x_{k+1}| = M \ , \ \forall k, i \leq k < j,$$

para algún valor de M .

Por ejemplo, dada la secuencia:

[0, 1, 3, 5, 3, 5, 0, 10, 9, 10, 11, 12, 11, 0, 0]

podemos encontrar, entre otras, las subsecuencias:

- [0, 1] es una subsecuencia de tamaño 2 en la que M = 1.
- [1, 3, 5, 3, 5] es una subsecuencia de tamaño 5 en la que M = 2.
- [10, 9, 10, 11, 12, 11] es la subsecuencia más larga, de tamaño 6 y en la que M = 1.

Por tanto el problema es determinar la longitud de la subsecuencia consecutiva de longitud máxima que cumpla la propiedad anterior. Diseñar, analizar la eficiencia e implementar un algoritmo sencillo para esta tarea, y luego hacer lo mismo con un algoritmo más eficiente, basado en “divide y vencerás”, de orden O(n log n). Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

## 2. IMPLEMENTACIÓN DEL ALGORITMO SENCILLO

A continuación se muestra el fragmento de código asociado al algoritmo sencillo para resolver el problema:

```
solucion MetodoClasico_lims(int V[], int ini, int fin){
  solucion res;
  int com, M, i = ini, longMax = -1;

  M = abs(V[i]-V[i+1]);
  com = i;
  i++;

  while(i+1 <= fin){
    if(abs(V[i]-V[i+1]) != M){
      if((i-com)+1 > longMax){
        res.comienzo = com;
        res.fin = i;
        longMax = (i-com)+1;
        res.longitud = longMax;
        res.M = M;
      }
      com = i;
      M = abs(V[i] - V[i+1]);
    }
    else
      i++;
  }
  //Comprobar si la mayor secuencia es la última:
  if((i-com)+1 > longMax){
    res.comienzo = com;
    res.fin = i;
    longMax = (i-com)+1;
    res.longitud = longMax;
    res.M = M;
  }
  return res;
}
```

### Funcionamiento explicado:

El algoritmo consiste en aplicar de forma secuencial lo que pide el problema, comparando pares de números y obteniendo su M. El método recibe como entrada una cadena, su inicio y su fin, y la recorre entera buscando la secuencia de números más grande que compartan la misma M dos a dos.

Si la M del primer par de números procesados coincide con la M del segundo par de números, se pasa a comprobar los siguientes elementos. En caso de que no se cumpla la condición, se comprueba si la longitud de la cadena actual es mayor que la longitud máxima hasta el momento. Si esto es cierto se actualizan los valores almacenados de la máxima subsecuencia. Sea cierto o no, siempre se actualiza el inicio de la nueva subsecuencia a la posición actual y se calcula el nuevo M. Este proceso se repite hasta que se alcance el final del vector. Una vez fuera del bucle se comprueba únicamente si la última subsecuencia es mayor que la almacenada hasta el momento (ya que por como está colocada la condición de parada del while la ultima subsecuencia no se comprueba) si es así se actualiza el valor.

Por último se devuelve la cadena con más longitud, su inicio, su final y el valor M que comparten sus componentes.

## 3. IMPLEMENTACIÓN DEL ALGORITMO DIVIDE Y VENCERÁS

A continuación se muestra el fragmento de código asociado al Divide y Vencerás para resolver el mismo problema enunciado anteriormente:

```
// Método Divide y Vencerás para resolver el problema. O(nlog(n))
// Recibe el vector, y los límites donde aplicar el algoritmo.
solucion MetodoDivideVenceras_lims(int V[], int ini, int fin){
  solucion sal1, sal2, subcadena_inf, subcadena_sup, union_cadenas, res;
  int centro, max;
  union_cadenas.longitud = -1;
  union_cadenas.comienzo = -1;
  union_cadenas.fin = -1;
  union_cadenas.M = -1;

  if(fin - ini < UMBRAL)
    //Aplicar método clásico si esta por debajo del umbral:
    return MetodoClasico_lims(V, ini, fin);
  else{
    centro = (ini + fin)/2;
    sal1 = MetodoDivideVenceras_lims(V,ini, centro);
    //Calcula la ultima subsecuencia de la parte inferior:
    subcadena_sup = UltimaSubsecuencia(V, ini, centro);
    sal2 = MetodoDivideVenceras_lims(V, centro+1, fin);
    //Calcular la primera subsecuencia de la parte superior:
    subcadena_inf = PrimeraSubsecuencia(V, centro+1, fin);

    //Recombinar:
    //Calcular la M de donde hemos cortado las cadenas:
    union_cadenas.M = abs(V[subcadena_sup.fin] - V[subcadena_inf.comienzo]);

    //Comprobar si se ha cortado alguna serie por la mitad y si es así volver a unirlas:
    if(subcadena_inf.M == subcadena_sup.M && subcadena_sup.M == union_cadenas.M &&
subcadena_inf.M == union_cadenas.M){
      union_cadenas.longitud = subcadena_inf.longitud + subcadena_sup.longitud;
      union_cadenas.comienzo = subcadena_sup.comienzo;
      union_cadenas.fin = subcadena_inf.fin;
    }
    else if(subcadena_inf.M == union_cadenas.M){
      union_cadenas.longitud = subcadena_inf.longitud + 1;
      union_cadenas.comienzo = subcadena_sup.fin;
      union_cadenas.fin = subcadena_inf.fin;
    }
    else if(subcadena_sup.M == union_cadenas.M){
      union_cadenas.longitud = subcadena_sup.longitud + 1;
      union_cadenas.comienzo = subcadena_sup.comienzo;
      union_cadenas.fin = subcadena_inf.comienzo;
    }

    //Devolver la cadena con mayor longitud de las construidas:
    res = sal1;
    max = sal1.longitud;

    if(union_cadenas.longitud > max){
      res = union_cadenas;
      max = union_cadenas.longitud;
    }
    if(sal2.longitud > max)
      res = sal2;
  }
  return res;
}
```

## Funcionamiento explicado:

El algoritmo se complica, ya que debemos tener en cuenta los casos que pueden ocurrir cuando dividimos el problema en otro más sencillo (dividimos en dos subcadenas la cadena original).

Hemos delimitado el umbral a 100 elementos ya que era el mínimo número con el que se apreciaba una buena eficiencia. Si el número de elementos de la cadena a procesar es mayor que el umbral, se llamará recursivamente al mismo algoritmo divide y vencerás. Si el número de elementos es menor que el umbral, se usará el método sencillo antes descrito (caso base).

El algoritmo divide y vencerás se compone de los siguientes pasos:

- **Descomponer el problema en subcasos más pequeños:**
  Partimos la cadena inicial en dos partes iguales (dos mitades). A continuación se realiza una llamada recursiva, la cual calculará la mayor subsecuencia de la mitad izquierda (inferior) asignándola a `sal1` y la mayor subsecuencia de la mitad derecha, asignándola a `sal2` .
  También calculamos la última subsecuencia de la parte inferior y la primera subsecuencia de la parte superior como datos que servirán de ayuda en la parte de combinar las soluciones obtenidas, (de esta forma nos aseguramos que no hemos partido por la mitad la mayor subsecuencia). Para ello se utilizan dos algoritmos idénticos al método sencillo, con la única diferencia de que el bucle finaliza cuando se encuentra la primera (o última si comenzamos por el final) subsecuencia (no continua hasta el final). Las subsecuencias encontradas se asignan a las variables `subcadena_sup` y `subcadena_inf`. Para más información sobre los algoritmos `UltimaSubsecuencia` y `PrimeraSubsecuencia` consultar el código fuente.

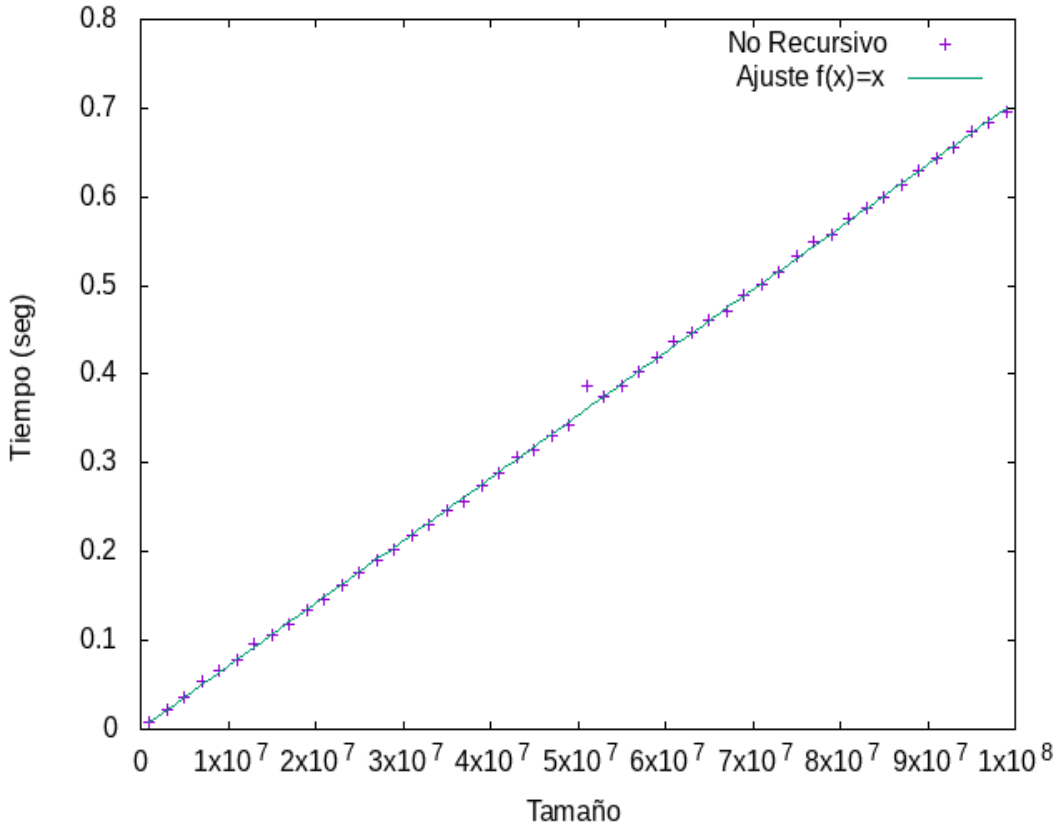
- **Resolver todos los casos:**
  Se aplicará el algoritmo divide y vencerás de forma recursiva hasta que el número de elementos de la subsecuencia coincida o sea menor que el umbral, entonces se aplicará el algoritmo sencillo.

- **Combinar las soluciones obtenidas:**
  Calculamos el valor de M de los dos números por los cuales hemos separado las cadenas, y lo comparamos con los valores de M almacenados en `subcadena_inf` y `subcadena_sup`. Pueden ocurrir cuatro casos: que hayamos cortado la subsecuencia en dos partes (los tres M coinciden) o que solo nos hayamos dejado un número aislado, (el M central coincide con el M superior o inferior). Sea cual sea el caso se construye la cadena correspondiente y se almacena en `union_cadenas`. También puede ocurrir que no se haya cortado ninguna cadena, en este caso union_cadenas no se construye y su longitud permanece a -1.
  Por último, devolvemos de las tres cadenas obtenidas (`sal1`, `sal2` y `union_cadenas`) aquella con la mayor longitud.

# 4. ANÁLISIS DE EFICIENCIA

## Eficiencia híbrida del algoritmo sencillo

Este algoritmo presenta un orden de eficiencia lineal (O(n)), ya que simplemente recorre un vector de n elementos de inicio a fin. El tiempo crece en función del tamaño de elementos de los que se compone el vector.




Hemos ajustado los tiempos obtenidos al usar el algoritmo sencillo para distintos tamaños de vectores a la función ***f(x)=a0*x+a1***. La constante *a0* es la que nos importa ya que es la que multiplica a *x*, y su valor es:

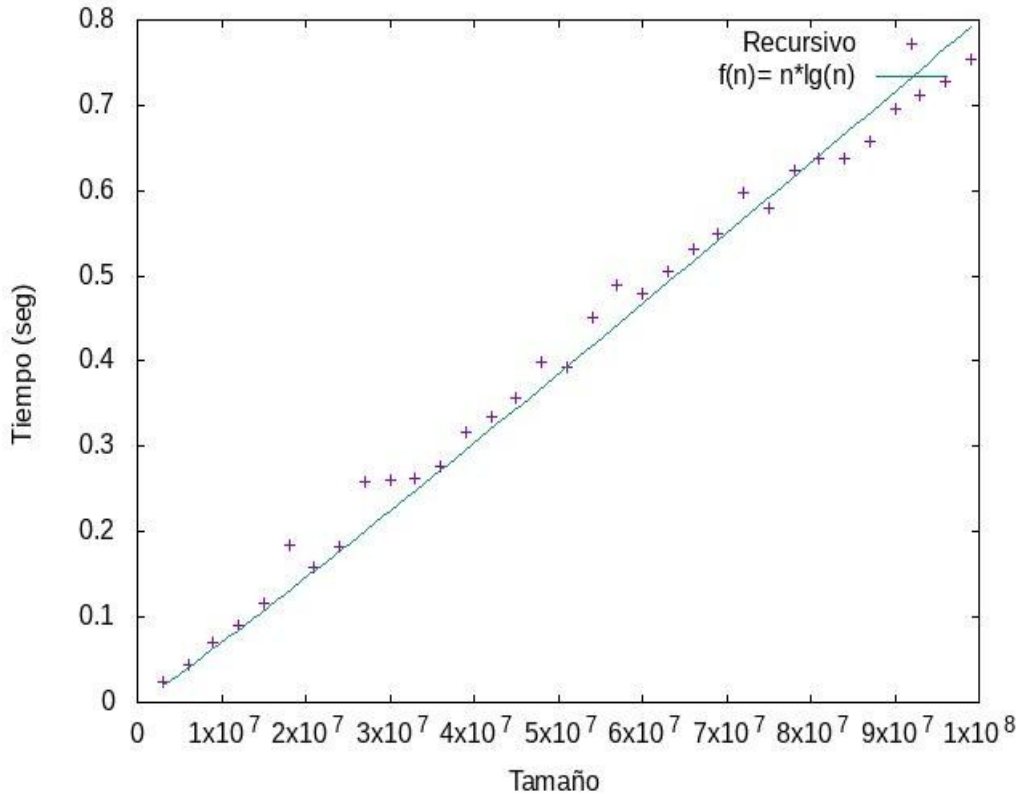
| Final set of parameters | Asymptotic Standard Error |
|---|---|
| ========== | ========== |
| **a0 = 7.0655e-09** | +/- 2.136e-11 (0.3023%) |

Podemos apreciar como los datos de tiempos obtenidos se ajustan aceptablemente a la función, demostrando que es de orden O(n).

## Eficiencia híbrida del algoritmo divide y vencerás

En el algoritmo divide y vencerás estamos dividiendo problema en dos subproblemas de tamaño la mitad, más un coste adicional de orden lineal (ya que bajo el umbral se está aplicando el algoritmo sencillo (de orden lineal), para calcular la subcadena inferior y superior aplica el mismo algoritmo (también orden n) y por último se realizan comparaciones y asignaciones entre enteros, lo que tiene un coste constante), en consecuencia la ecuación recurrente nos queda del tipo:

$$T(n) = 2*T(\frac{n}{2})+n \in O(nlog(n))$$




Hemos ajustado los tiempos obtenidos al usar el algoritmo divide y vencerás para distintos tamaños de vectores a la función ***g(x)=a0*log(x)*x+a1***. Necesitamos los valores de las constantes ocultas *a0* y *a1*:

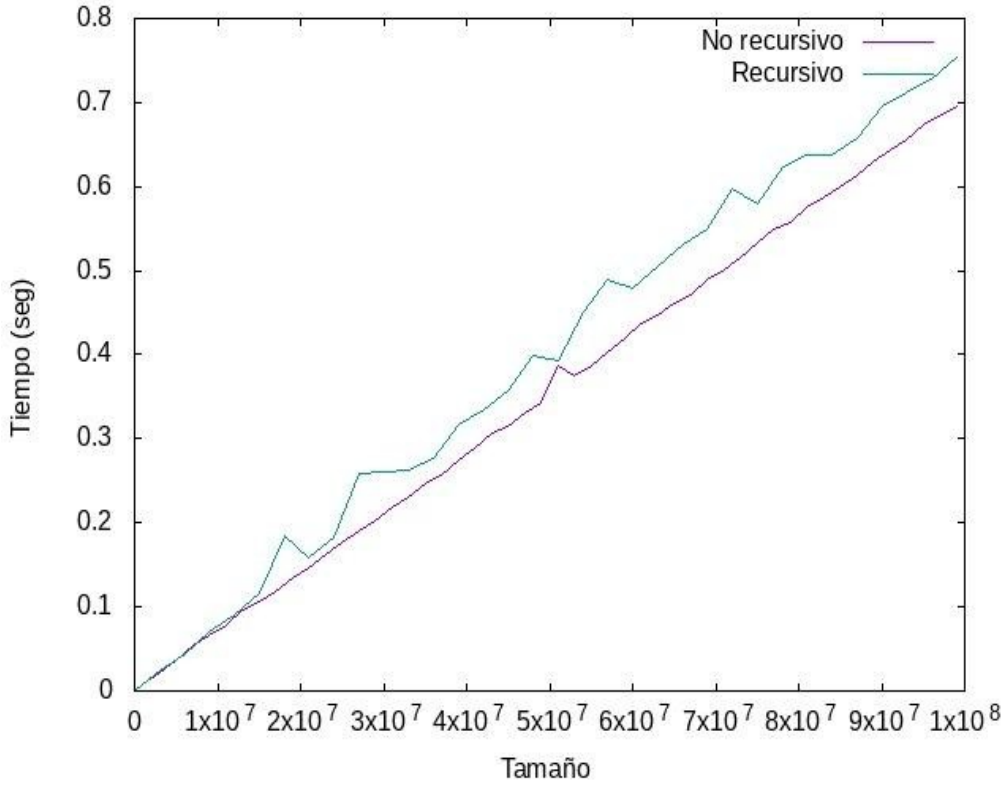
| Final set of parameters | Asymptotic Standard Error |
|---|---|
| ========== | ========== |
| **a0 = -3.82826e-10** | +/- 1.647e-10 (43.03%) |
| a1 = 1 | +/- 0.1742 (17.42%) |

Podemos apreciar como los datos de tiempo se ajustan a la función de orden O(n log n) pero no de un modo muy significativo. Esto es debido a que el generador de vectores genera un vector al azar,

que puede presentar mayor o menor dificultad según la composición de éste a la hora de que el algoritmo haga sus cálculos. Por ello algunas veces se obtienen mejores tiempos y otras peores.

## Comparación de ambos algoritmos

Gráfica que compara las funciones del algoritmo sencillo (no recursivo) y del algoritmo divide y vencerás (recursivo):



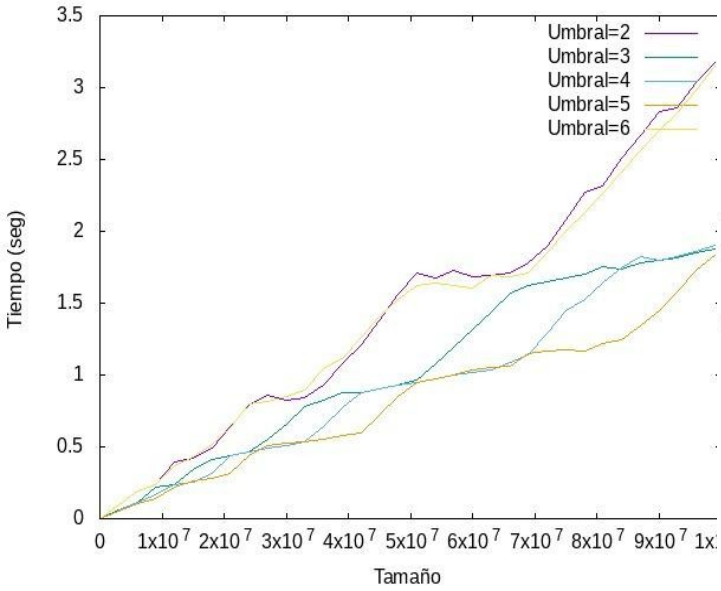

En esta gráfica podemos apreciar que el mejor algoritmo de los dos es el algoritmo sencillo, ya que presenta un tiempo de ejecución menor a la hora de encontrar la respuesta al problema. Esto se debe a que es de orden O(n), frente al orden O(nlog(n)) del algoritmo Divide y Vencerás. Además en este caso, el algoritmo sencillo es mucho más simple que el recursivo. En definitiva para este problema no proporciona mucha ventaja el hecho de aplicar un algoritmo Divide y vencerás, ya que pensándolo un poco se puede llegar a un algoritmo bastante bueno de orden lineal.

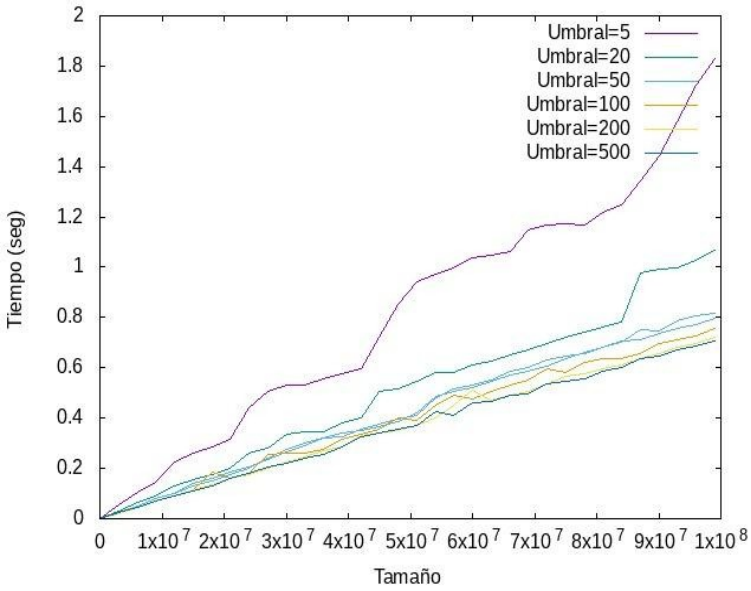
## 5. COMPARACIÓN ENTRE DISTINTOS UMBRALES

En el algoritmo Divide y Vencerás usamos el término *umbral de recursividad* para determinar si en un conjunto de elementos debemos volver a aplicar recursivamente el algoritmo DyV o podemos aplicar el caso base. En nuestro caso, el umbral determina el número de índices del que se compone una subcadena.

En la implementación hemos usado la variable *UMBRAL* con valor 100, ya que era el mínimo valor en el que la eficiencia del algoritmo no se veía muy afectada.

En este apartado hemos querido corroborar el hecho de que la determinación del umbral afecta a la eficiencia probando el algoritmo recursivo con distintos umbrales de recursividad, para apreciar como la eficiencia es peor en casos de que el umbral sea un número pequeño, y mejor si el umbral es un número mayor.




En el caso de usar umbrales pequeños (izquierda) la eficiencia empeora mucho si usamos el algoritmo no recursivo a partir de vectores de tamaño 2 o 6. Sin embargo, mejora si lo usamos en vectores de tamaño 3, 4 y 5. Entre estos 5 valores, la elección más acertada es escoger como umbral el número 5.

Pero decidimos probar a poner como umbral números mucho mayores (derecha), obteniendo una mejor eficiencia respecto al valor de umbral 5 que habíamos determinado anteriormente como la elección más acertada.
Como podemos ver en la gráfica, el valor de umbral 20 es el que más lejos se queda de una buena eficiencia, y el resto de valores mejoran conforme son números mayores.