
# Práctica 2
# Algorítmica

## Algoritmos Divide y Vencerás

Realizado por grupo Orquídeas (subgrupo A1):

- Azorín Martí, Carmen
- Cribillés Pérez, María
- Ortega Sevilla, Clara
- Torres Fernández, Elena

Profesora: Lamata Jiménez, María Teresa

Curso 2021/2022 2º cuatrimestre

# Índice

# 1. Introducción

## ¿Algoritmo fuerza bruta o Divide y Vencerás?

En esta práctica nos han proporcionado dos generadores de datos para poder probar los algoritmos que hemos creado. Hemos tenido que resolver dos problemas de diferentes alternativas y ver cuál es la más eficiente. Para ello, hemos resuelto ambos primero a **fuerza bruta**, es decir, lo más lógico y sencillo pero con una eficiencia peor. Posteriormente, hemos utilizado para resolverlo **Divide y Vencerás** que, como hemos comprobado, es mucho más eficiente que con fuerza bruta, pero hay que trabajar un poco más su implementación, ya sea por recurrencia u otros métodos más específicos.

## Funcionamiento Divide y Vencerás

¿Cómo hemos resuelto los problemas con Divide y Vencerás? Pues bien, tenemos un **problema inicial grande (P)**, el cuál lo hemos dividido en varios **subproblemas (Pi)** que hemos resuelto y al final, hemos combinado las **soluciones (Si)** de los subproblemas para obtener la **solución (S)** del problema inicial (P). Para poder resolver los subproblemas hemos tenido que aplicar nuestros conocimientos de recurrencia ya vistos en clase.

Antes de implementar Divide y Vencerás hemos tenido que hacer varias **comprobaciones previas**:

- Ver si el problema inicial se puede descomponer en subproblemas
- Los subproblemas tienen que tener la misma naturaleza que ellos mismos y el problema inicial.
- Los subproblemas no pueden ser muchos, sólo una cantidad pequeña. Además, cada subproblema debe de ser más sencillo que el problema inicial.
- Tiene que tener sentido combinar las soluciones de los subproblemas para obtener la solución final y que esa sea la solución del inicial.

**Esquema** que hemos seguido para aplicar este algoritmo:

Datos:

- DyV: función divide y vencerás
- P: problema inicial P
- tam: tamaño del problema inicial P
- S: solución del problema inicial P
- $n_0$: umbral caso base
- k : número de subproblemas
- P(i): subproblemas de P

- S(i): soluciones de los subproblemas
- Recombinar: función para combinar las soluciones
- fb: función del caso básico

Con estos datos ya podemos explicar el **pseudocódigo** de la técnica Divide y Vencerás:

```
Función DyV (P,tam)
        Si P es simple (tam<=n0) entonces se resuelve a fuerza bruta:
                S=fb(P, tam);
        En otro caso:
                Descomponemos P en subcasos P(1), P(2), … , P(k) más simples
                Para cada P(i), desde i hasta k:
                        S(i)=DyV(P(i), tam(i))
                S=Recombinar las S(i) en S
        Devolver S
Fin DyV
```

## Determinación del umbral

Es importante destacar que el algoritmo de Divide y Vencerás es más eficiente que el de Fuerza Bruta sólo para los valores de entrada mayores que un determinado umbral $n_0$. Por ello, **conocer el umbral es clave para saber si nuestros resultados van a ser más eficientes con un algoritmo o con el otro según el número de valores de entrada que tengamos**. Es decir, una vez más, será muy importante conocer el tamaño del problema (n) que estamos tratando para saber si necesitamos aplicar el algoritmo de fuerza bruta ($n \leq n_0$) o DyV ($n > n_0$).

El umbral no es único, pero sí en cada implementación, y está entre 0 (para $n_0=1$ el algoritmo básico sólo actúa una vez y en el resto de casos se aplica DyV) e infinito (no se aplica la técnica de DyV nunca). Para hallar el umbral se puede hacer de dos formas: experimentalmente y teóricamente.

Por un lado, el **método experimental** parte de las implementaciones de ambos algoritmos y se resuelve el problema para distintos valores de n con ambos algoritmos. Así conseguimos una gráfica en la que se espera que el tiempo del algoritmo de Fuerza Bruta vaya aumentando con valores de entrada mayores; mientras que el de DyV, vaya disminuyendo. Con esto, el valor del umbral es el punto de corte de ambas funciones.

Por otro lado, el **método teórico** no necesita que previamente se hayan implementado los algoritmos. Ahora, el umbral se consigue igualando los tiempos de ejecución de los dos algoritmos para un tamaño n, calculado con la eficiencia híbrida por las constantes ocultas. En general, siendo $T(n) = h(n)$ el tiempo de ejecución del algoritmo de Fuerza Bruta y $T(n) = aT(n/b) + g(n)$ el de DyV, el umbral es $h(n) = T(n) = a\,h(n/b) + g(n)$, con $n = n_0$.

# 2. Problema 1

En este ejercicio usamos al archivo ***generador1.cpp***, que hace lo siguiente:

- Genera un vector de 2n-1 números ordenados y sin repetir desde [-(n-1), (n-1)] para un n dado como parámetro.
- Realiza una permutación aleatoria de ese vector.
- Seleccionan los n primeros elementos.
- Ordena el vector resultante de forma creciente.

El objetivo es determinar **si existe un índice i tal que v[i] = i.** Si es así, debemos encontrar uno de ellos e imprimirlo por pantalla. Ya nos proporcionan un generador de vectores con las características correspondientes, por lo que solo hemos tenido que implementar el algoritmo que resuelve el problema con dos técnicas diferentes: fuerza bruta y divide y vencerás.

## Algoritmo de fuerza bruta

- **Pseudocódigo**

```
Función fb(vector)
        Inicializar indice=NOINDEX y encontrado=false
        Recorrer vector mientras !encontrado y estemos dentro de vector
                Si vector[i] == i, indice=i y encontrado=true
        Devolver indice
Fin fb
```

- **Código con explicación**

El algoritmo funciona de forma que el bucle for recorre todo el vector hasta el final del mismo o hasta encontrar el valor que coincide con el índice de su posición. Si se encontrase, se ejecutaría el if y el bucle finalizaría, por lo que se devolvería su posición. En caso contrario se devolvería un valor que nunca podría ser índice.

Cabe destacar que este algoritmo siempre encontrará el *primer* índice que cumpla la condición que buscamos, aunque en posiciones más altas también haya otros que lo cumplan.

- **Eficiencia teórica**

```
double fb (vector <int> v){ //O(n)
        int indice= NOINDEX; //O(1)
        bool encontrado = false;    //O(1)

        for (int i=0; i<v.size() && !encontrado; i++){ //O(n)
            if (v[i]==i){     //O(1)

                    indice= i; //O(1)
                    encontrado = true; //O(1)
            }
        }

        return indice; //O(1)
}
```

La eficiencia de este algoritmo viene dada por el bucle for, ya que el resto del código es de eficiencia O(1). Como vemos, el bucle for se ejecutaría n veces en el peor de los casos, por tanto, su eficiencia sería **O(n)**, siendo esta la eficiencia del algoritmo de **fuerza bruta.**

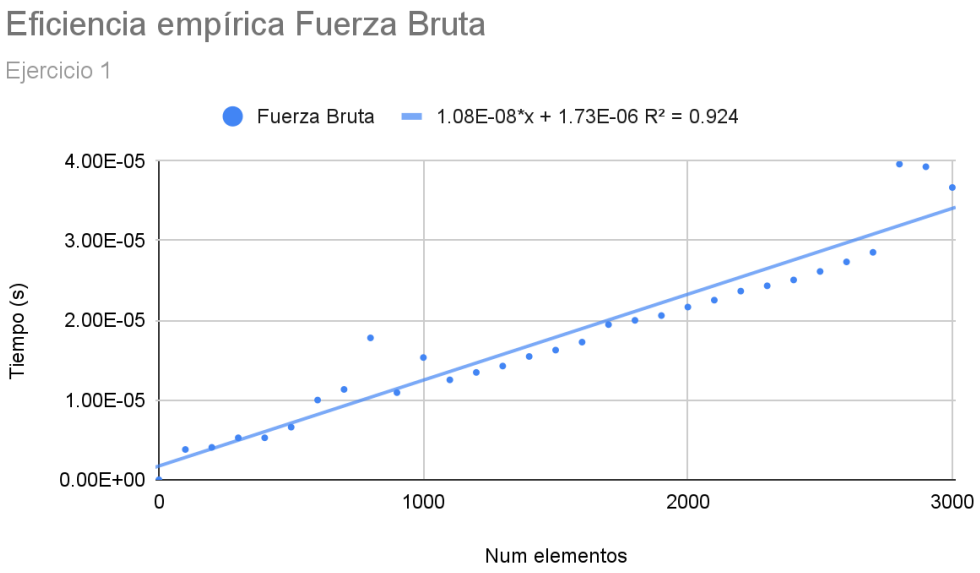
- **Eficiencia empírica**

La eficiencia empírica, con la que hemos trabajado en la práctica anterior, consiste en medir el tiempo de un algoritmo para cada tamaño de entrada y poder estudiar así su comportamiento ante un determinado problema.

En el caso del algoritmo de fuerza bruta, observamos que su comportamiento es lineal, coincidiendo así con la eficiencia teórica. Dicho comportamiento se aprecia mejor a mayor tamaño de los datos, siendo el coeficiente de determinación ($R^2$) prácticamente igual a 1.

A continuación, se muestra la tabla de ejecución de generador1.cpp para valores comprendidos entre 100 y 3000 y el tiempo que tarda en ejecutar el algoritmo de fuerza bruta el código para dichos tamaños. Así mismo, podemos ver cómo efectivamente el algoritmo de fuerza bruta es **lineal**.

La gráfica siguiente muestra la función lineal ajustada a los tiempos dados.




Algoritmo Fuerza Bruta

| Tamaño | Tiempo (s) |
|---|---|
| 100 | 3.80E-06 |
| 200 | 4.07E-06 |
| 300 | 5.27E-06 |
| 400 | 5.27E-06 |
| 500 | 6.60E-06 |
| 600 | 1.00E-05 |
| 700 | 1.13E-05 |
| 800 | 1.78E-05 |
| 900 | 1.09E-05 |
| 1000 | 1.53E-05 |
| 1100 | 1.25E-05 |
| 1200 | 1.35E-05 |
| 1300 | 1.43E-05 |
| 1400 | 1.55E-05 |
| 1500 | 1.63E-05 |
| 1600 | 1.73E-05 |
| 1700 | 1.95E-05 |
| 1800 | 2.00E-05 |
| 1900 | 2.06E-05 |
| 2000 | 2.17E-05 |
| 2100 | 2.25E-05 |
| 2200 | 2.37E-05 |
| 2300 | 2.43E-05 |
| 2400 | 2.51E-05 |
| 2500 | 2.61E-05 |
| 2600 | 2.73E-05 |
| 2700 | 2.85E-05 |
| 2800 | 3.96E-05 |
| 2900 | 3.93E-05 |
| 3000 | 3.67E-05 |

- **Eficiencia híbrida**

Esta eficiencia nos va a permitir obtener una función exacta de la eficiencia, hallando también las **constantes ocultas**. Para ello, vamos a utilizar los datos de la salida de gnuplot.

La función correspondiente al orden O(n) es: f(x)= a0*x+a1.

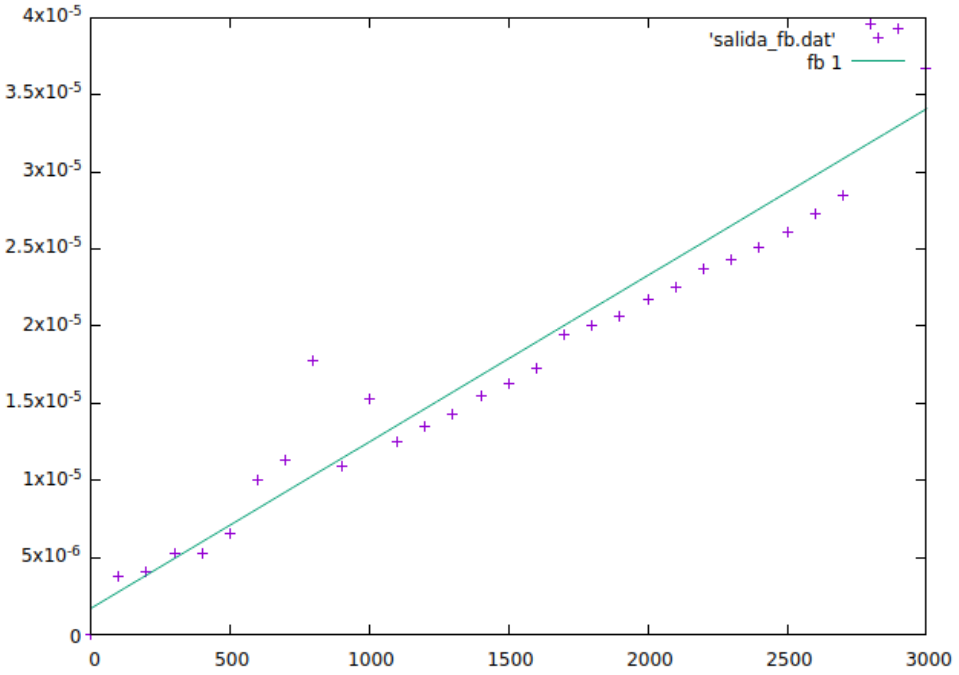
Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=a0*x+a1**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Siendo 'salida.dat' el archivo con los tiempos obtenidos para cada tamaño *n* cuándo se ejecuta el generador1.cpp mediante el algoritmo de fuerza bruta, es decir, un archivo con los datos de la tabla anterior. Obtenemos las siguientes constantes:

Final set of parameters
==========
a0 =1.07757e-08
a1 = 1.72706e-06

Con este resultado, la función quedaría de la forma: **f(x)= 1.07757e-08 *x+1.72706e-06,** y en la gráfica extraída de gnuplot se observa también el comportamiento lineal, siendo la función que hemos calculado la línea punteada.



## Algoritmo con divide y vencerás

- **Pseudocódigo**

```
Función dyv(vector, inicio del vector, final del vector)
        Bucle while mientras inicio del vector <= final del vector
        Buscar mitad del vector = "m"
           Si el vector en la posición m es menor que el propio m, inicio = m+1
           Si el vector en la posición m es igual al propio m, devolver m
           En otro caso, final del vector= m-1
        Actualizar valor de m
        Fin del bucle
        Si el inicio del vector fuese mayor que el final, devolver valor imposible
Fin dyv
```

- **Código con explicación**

El bucle while se ejecuta mientras el inicio del vector sea menor que el final del vector y en cada iteración definimos la mitad del vector. Distinguimos entonces tres casos dependiendo del valor obtenido como mitad ("m"):

- El valor en la posición m es igual a m → devolvemos el índice m ya que es el que buscamos.
- El valor en la posición m es menor que m → modificamos el valor del inicio del vector a "m+1". Con esto, nos quedamos con la parte derecha del vector para la siguiente iteración que irá desde [m+1,f]
- El valor en la posición m es mayor que m → modificamos el valor del final del vector a "m-1". Con esto, nos quedamos con la parte izquierda del vector para la siguiente iteración que irá desde [i,m-1]

Si el valor del inicio fuese mayor que el del final del vector, se devolvería un valor imposible, indicando que no se ha encontrado ningún índice que cumpla v[i]= i.

Cabe mencionar que en este caso, el algoritmo de dyv no tiene por qué encontrar el primer índice que cumpla la condición que buscamos debido a que vamos partiendo el vector por mitades y no lo recorremos de izquierda a derecha. No obstante, ambos algoritmos realizan correctamente la tarea pedida de encontrar *un* índice que cumpla v[i]=i en caso de que lo haya.

- **Eficiencia teórica**

```
double dyv (vector <int> v, int i, int f){

        while (i<=f){   //O(logn)
                int m=(i+f)/2; //O(1)
                //medio

                if (v[m]< m){
                        i=m+1; //O(1)
                }
                else {
                        if (v[m]== m)    //O(1)
                                return (m);   //O(1)
                else
                        f = m - 1;      //O(1)
                }

                m = (i+f)/2;    //O(1)
                // se actualiza valor medio
        }

        if (i>f)  //O(1)
        //no se encuentra
                return 0.1;     //O(1)
                //valor imposible
}
```

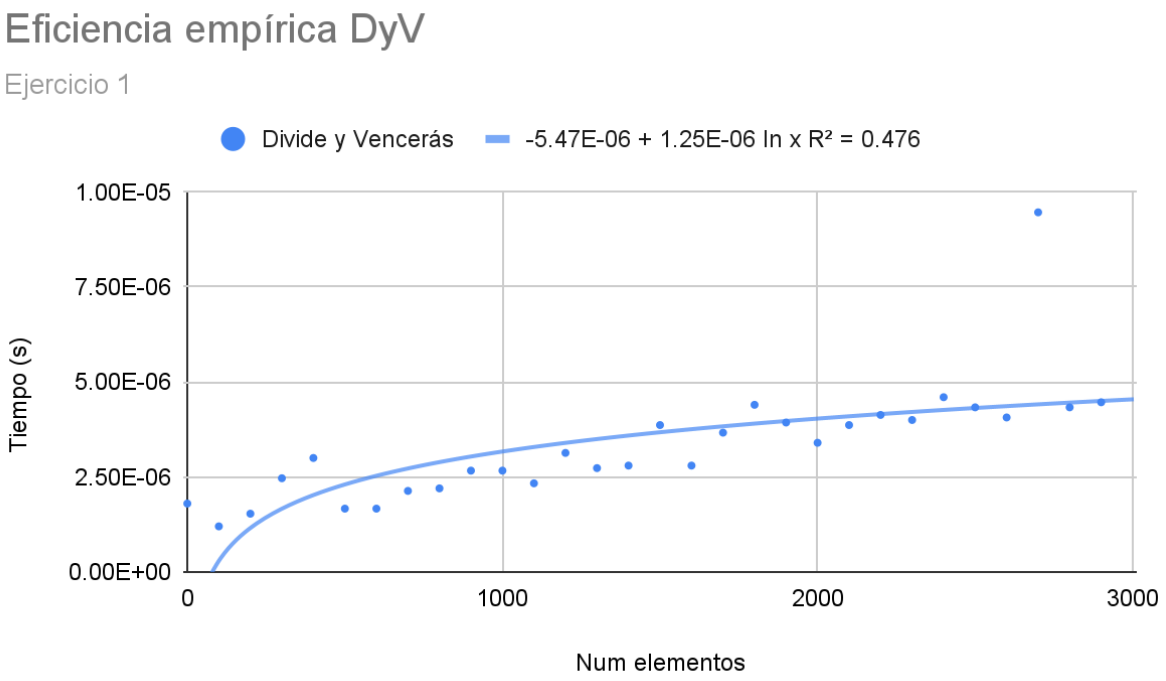
La eficiencia de este algoritmo viene dada por el bucle while, ya que el resto del código es de eficiencia O(1). Como vemos, dentro del bucle while todo son asignaciones O(1) pero observamos que en cada iteración se parte el vector por la mitad actualizando el valor medio del mismo. Por tanto, su orden de eficiencia es **O(logn)**, siendo esta la eficiencia del algoritmo de **Divide y Vencerás.**

- **Eficiencia empírica**

En este caso, observamos que su comportamiento es logarítmico, coincidiendo también con la eficiencia teórica. Dicho comportamiento se aprecia mejor a menor tamaño de los datos, ya que a valores grandes, si observamos la gráfica, visualmente parece lineal. A medida que realizamos una tabla con datos más pequeños, vamos observando cómo la línea de la gráfica toma forma logarítmica.

A continuación, se muestra la tabla de ejecución de generador1.cpp para valores comprendidos entre 100 y 3000 y el tiempo que tarda en ejecutar el algoritmo de divide y vencerás el código para dichos tamaños. Podemos ver cómo efectivamente el algoritmo divide y vencerás es **logarítmico**.

En la siguiente tabla se muestra cómo se ajustan los tiempos dados a la eficiencia logarítmica calculada.



| Algoritmo Divide y Vencerás | |
|---|---|
| Tamaño | Tiempo (s) |
| 100 | 1.80E-06 |
| 200 | 1.20E-06 |
| 300 | 1.53E-06 |
| 400 | 2.47E-06 |
| 500 | 3.00E-06 |
| 600 | 1.67E-06 |
| 700 | 1.67E-06 |
| 800 | 2.13E-06 |
| 900 | 2.20E-06 |
| 1000 | 2.67E-06 |
| 1100 | 2.67E-06 |
| 1200 | 2.33E-06 |
| 1300 | 3.13E-06 |
| 1400 | 2.73E-06 |
| 1500 | 2.80E-06 |
| 1600 | 3.87E-06 |
| 1700 | 2.80E-06 |
| 1800 | 3.67E-06 |
| 1900 | 4.40E-06 |
| 2000 | 3.93E-06 |
| 2100 | 3.40E-06 |
| 2200 | 3.87E-06 |
| 2300 | 4.13E-06 |
| 2400 | 4.00E-06 |
| 2500 | 4.60E-06 |
| 2600 | 4.33E-06 |
| 2700 | 4.07E-06 |
| 2800 | 9.47E-06 |
| 2900 | 4.33E-06 |
| 3000 | 4.47E-06 |

- **Eficiencia híbrida**

La función correspondiente al orden O(log n) sería: f(x)= a1+ log(a0*x).
Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=a0+ log(a1*x).**
- **gnuplot> fit f(x) 'salida.dat' via a0, a1.**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

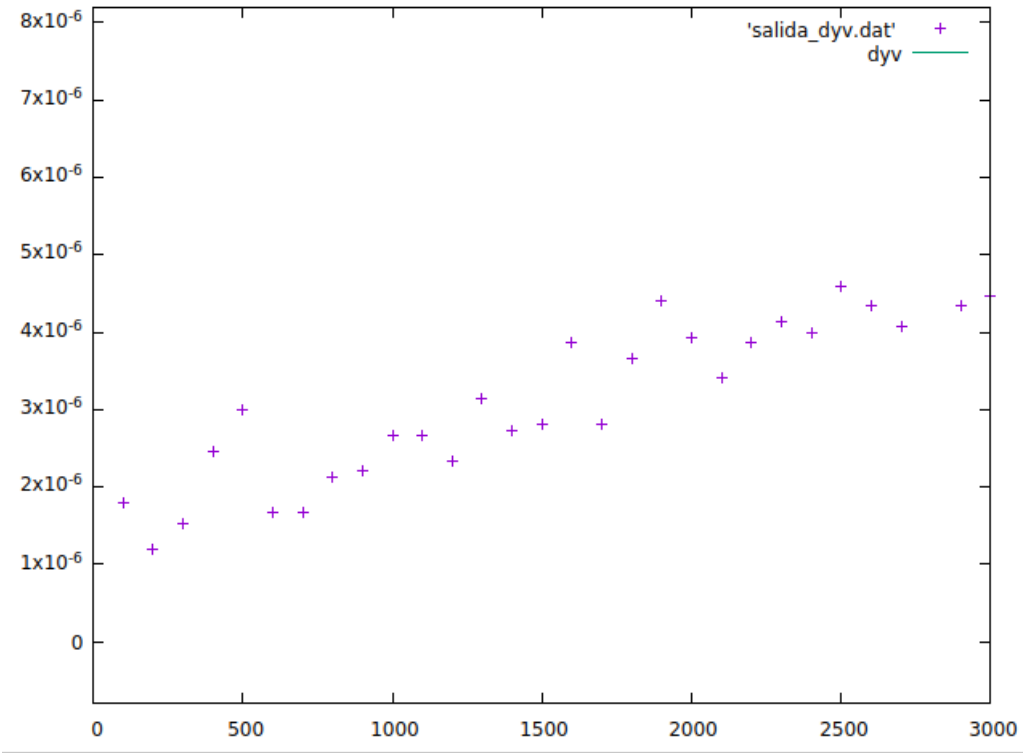
Siendo 'salida.dat' el archivo con los tiempos obtenidos para cada tamaño *n* cuándo se ejecuta el generador1.cpp mediante el algoritmo de dyv, es decir, un archivo con los datos de la tabla anterior. Obtenemos las siguientes constantes:

Final set of parameters
==========
a0 = -7.51057
a1 = -1.51709

Con este resultado, la función quedaría de la forma:

**f(x)= -7.51057+ log( -1.51709*x).**

En la gráfica obtenida en gnuplot se aprecia como tiene una tendencia logarítmica, lo que cabía de esperar:



## Comparación Fuerza Bruta y Divide y Vencerás

En la comparación de ambos algoritmos vemos que para tamaños grandes, el algoritmo de divide y vencerás es mucho mejor que el de fuerza bruta, ya que el algoritmo lineal queda muy por encima del logarítmico para las mismas entradas.

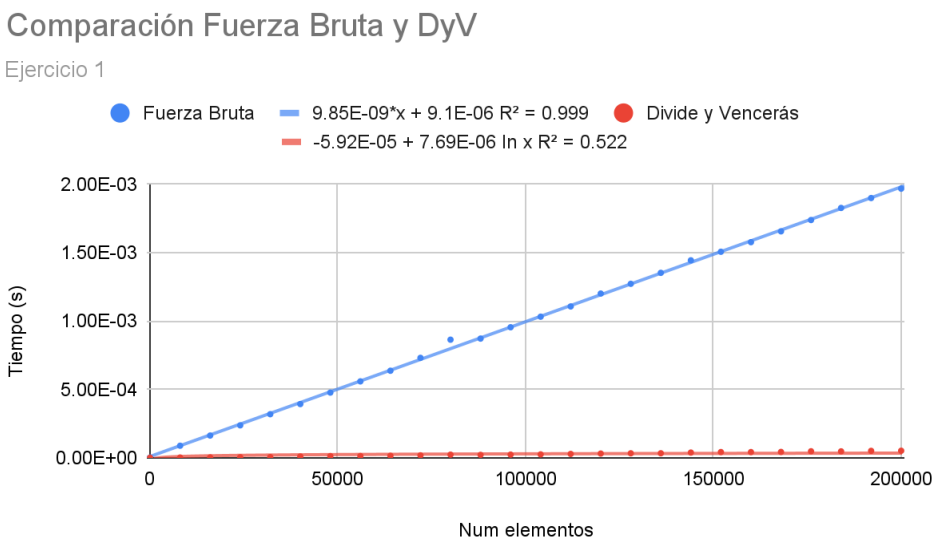
No obstante, si recogemos datos para una muestra menor, como por ejemplo de tamaño hasta 10, se observa que las dos funciones se cortan en el punto n=1, lo que nos dice que para dicho valor el algoritmo de fuerza bruta iguala al de divide y vencerás en tiempo de ejecución, constituyendo el umbral.

| | Tiempos (s) | |
|---|---|---|
| Tamaño | Fuerza Bruta | DyV |
| 100 | 2.00E-06 | 8.00E-07 |
| 8096 | 8.86E-05 | 2.40E-06 |
| 16092 | 0.000163 | 3.07E-06 |
| 24088 | 0.000237533 | 4.27E-06 |
| 32084 | 0.000318867 | 5.47E-06 |
| 40080 | 0.000393067 | 9.93E-06 |
| 48076 | 0.000476867 | 1.26E-05 |
| 56072 | 0.0005588 | 1.30E-05 |
| 64068 | 0.000636533 | 1.51E-05 |
| 72064 | 0.000730733 | 1.82E-05 |
| 80060 | 0.000863467 | 2.33E-05 |
| 88056 | 0.000871533 | 2.17E-05 |
| 96052 | 0.000954267 | 2.37E-05 |
| 104048 | 0.00103147 | 2.53E-05 |
| 112044 | 0.0011064 | 2.86E-05 |
| 120040 | 0.00120047 | 3.13E-05 |
| 128036 | 0.00127173 | 3.33E-05 |
| 136032 | 0.00135187 | 3.35E-05 |
| 144028 | 0.0014434 | 3.87E-05 |
| 152024 | 0.00150593 | 4.22E-05 |
| 160020 | 0.00157607 | 4.21E-05 |
| 168016 | 0.00165513 | 4.31E-05 |
| 176012 | 0.00173773 | 4.79E-05 |
| 184008 | 0.00182607 | 4.73E-05 |
| 192004 | 0.00189847 | 5.10E-05 |
| 200000 | 0.00196713 | 5.19E-05 |

***Tabla de valores muestra mayor***

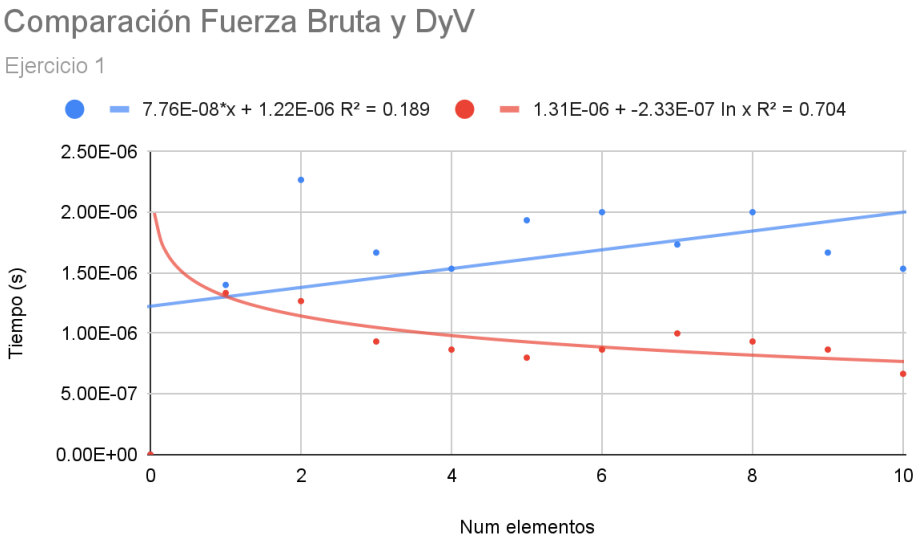
| | Tiempos (s) | |
|---|---|---|
| Tamaño | Fuerza Bruta | DyV |
| 1 | 1.40E-06 | 1.33E-06 |
| 2 | 2.27E-06 | 1.27E-06 |
| 3 | 1.67E-06 | 9.33E-07 |
| 4 | 1.53E-06 | 8.67E-07 |
| 5 | 1.93E-06 | 8.00E-07 |
| 6 | 2.00E-06 | 8.67E-07 |
| 7 | 1.73E-06 | 1.00E-06 |
| 8 | 2.00E-06 | 9.33E-07 |
| 9 | 1.67E-06 | 8.67E-07 |
| 10 | 1.53E-06 | 6.67E-07 |

***Tabla de valores muestra menor***




En la gráfica de arriba se muestran los tiempos de ejecución del algoritmo Divide y Vencerás en rojo y del de Fuerza Bruta en azul. Se puede ver con claridad cuál es más eficiente. Claro, porque hemos utilizado valores muy grandes para que se vea bien la diferencia.

En el gráfico de abajo, hemos utilizado valores mucho más pequeños, lo que nos permite observar que el umbral se encuentra en el 1 y que, a partir de ahí, el algoritmo de Divide y Vencerás es más eficiente.



## Determinación del umbral

En el apartado anterior ya hemos calculado el umbral de **forma experimental** al comparar los dos algoritmos con los mismos valores de entrada en una misma gráfica y viendo el **punto de corte.** Para valores grandes no se observa el punto de corte pero si lo estudiamos hasta 10 sí.

Como se ve en la gráfica justo anterior, el umbral es $\mathbf{n_0=1}$. Podemos observar que la función logarítmica tiene una pendiente que decrece cada vez más lento; mientras que la función lineal crece con una pendiente constante mayor. Por ello, estas gráficas se acaban cortando en un valor, el umbral.

¿Qué quiere decir que el umbral sea 1? Nos indica que sólo para el valor de entrada n=1 es recomendable usar el algoritmo de fuerza bruta, pues para el resto de casos n≥2, Divide y Vencerás es más eficiente.

## Algoritmo con repeticiones. ¿Qué ocurre?

**El algoritmo de fuerza bruta**, al recorrer el vector entero, **seguirá siendo válido aun habiendo repeticiones**, pues evalúa cada casilla del vector sin descartar en ningún momento una parte del mismo, como hace el divide y vencerás. Es por esto que **Divide y Vencerás sí necesitaría una modificación** para tener en cuenta si una de las repeticiones está o no en su posición.

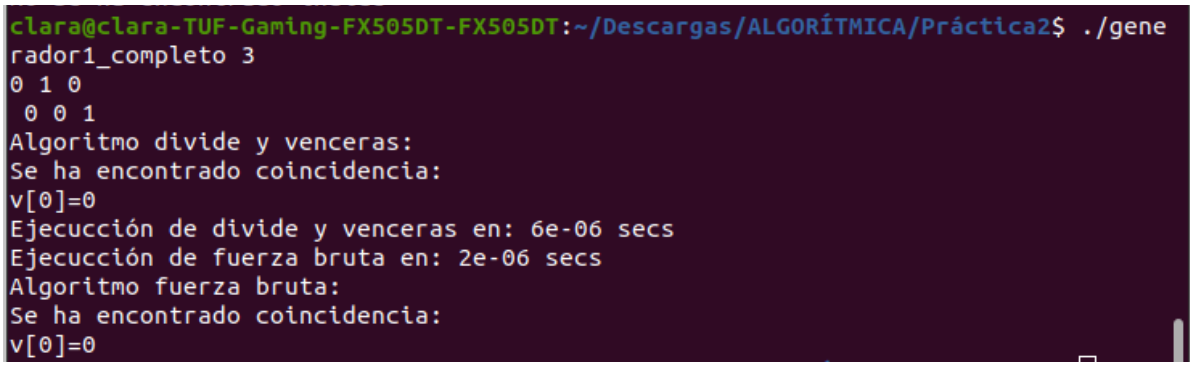
A continuación mostramos un ejemplo de ejecución en el que el algoritmo de fuerza bruta sí encuentra un índice que cumple v[i]=i pero Divide y Vencerás no encuentra ninguno.

```
clara@clara-TUF-Gaming-FX505DT-FX505DT:~/Descargas/ALGORÍTMICA/Práctica2$ ./gene
rador1_completo 10
1 -6 8 8 -5 -9 -9 5 8 0
 -9 -9 -6 -5 0 1 5 8 8 8
Algoritmo divide y venceras:
No se ha encontrado indice
Ejecucción de divide y venceras en: 5e-06 secs
Ejecucción de fuerza bruta en: 2e-06 secs
Algoritmo fuerza bruta:
Se ha encontrado coincidencia:
v[8]=8
```

Para que, aun habiendo repeticiones, el algoritmo de dyv funcione correctamente, realizamos una modificación en la implementación del código del dyv como sigue:

```
double dyv (vector <int> v, int i, int f){    //O(nlogn)
int k=1; //O(1)
int j=1; //O(1)

while (i<=f){  //O(nlogn)
        int m=(i+f)/2;   //O(1)
        //medio
        if (v[m]< m){ //O(n)
                while(v[m-k] == v[m]){      //O(n)
                        if(v[m-k]== m-k)     //O(1)
                                return m-k;  //O(1)

                        k++;  //O(1)
                }
                i=m+1; //O(1)
                k=0;   //O(1)
        }
        else {          //O(n)
                if (v[m]== m)   //O(1)
                        return (m);    //O(1)
                else{  //O(n)
                        while(v[m+j] == v[m]){      //O(n)
                                if(v[m+j]== m+j)     //O(1)
                                        return m+j;  //O(1)

                                j++; //O(1)
                        }
                        f = m - 1;       //O(1)
                        j=0;             //O(1)
                }
        }
        m = (i+f)/2;    //O(1)
}

if (i>f) //O(1)
        return 0.1; //O(1)
return 0;
}
```

En ella, el cambio funciona de manera que la parte que va a ser descartada se revisa antes de desecharla. De esta forma, como el vector está ordenado, si los números de entorno del tomado como mitad son iguales a dicho número se revisarían con respecto a su posición (por ejemplo, si se va a descartar la parte derecha, se revisaría la derecha y viceversa).




Como podemos ver en esta captura, el algoritmo de divide y vencerás encuentra un índice que cumple lo pedido, al igual que el de fuerza bruta, reflejando que el algoritmo de dyv modificado es correcto, pues la mitad del vector sería el segundo cero, y a partir de él pasaríamos a evaluar el primer cero encontrando la coincidencia antes de pasar a la parte derecha y descartar la izquierda, como se habría hecho en la versión original.

Como hemos podido observar en el código, la eficiencia del nuevo algoritmo de Divide y Vencerás es O(nlogn) que viene dado por el bucle while. Por tanto, los tiempos del algoritmo en Fuerza Bruta serán ligeramente menores que los del algoritmo Divide y Vencerás.

# 3. Problema 2

En este ejercicio lo que tenemos son **k vectores ordenados** (de menor a mayor), cada uno **con n elementos**, y queremos combinarlos **en un único vector ordenado (con kn elementos)**.

Como en el ejercicio anterior, lo hemos resuelto de dos maneras diferentes: una con fuerza bruta y otra con la técnica de divide y vencerás. En ambos casos hemos utilizado memoria dinámica y nos hemos ayudado de la herramienta de Valgrind para comprobar que habíamos liberado memoria correctamente y poder resolver las violaciones de segmento que nos han aparecido.

## Algoritmo de fuerza bruta

- **Código con explicación**

La función *fuerza_bruta* es llamada con los parámetros número de elementos de los vectores, número de vectores y la matriz que contiene los datos generados aleatoriamente.

Al principio, creamos el vector que va a contener el resultado final, inicializándolo y reservando memoria. A continuación, como el vector está vacío pues metemos la primera fila de la matriz.

Hacemos dos bucles anidados, los cuales recorren una iteración por vector y una iteración por cada elemento de esos vectores, respectivamente. Dentro de esos bucles anidados, debemos mirar cada elemento y compararlo con los elementos del vector final ya que hay que ir insertándolos ordenadamente.

Una vez que encontramos la posición que le corresponde, trasladamos todos los elementos de la derecha a una posición a la derecha para que se quede un hueco donde debemos de insertar el elemento con el cual estamos operando e insertamos.

- **Eficiencia teórica**

```
int* fuerza_bruta (int num_elem, int num_vect, int **matriz){

        //Vector final ordenado con el resultado de mezclar todos los vectores
        int* vect_final; //Inicializamos //O(1)
        vect_final = new int [num_elem*num_vect]; //O(1)

        bool encontrado;  //O(1)
        int k; //O(1)

        for (int i=0; i<num_elem; i++){ //O(n)
                vect_final[i]=matriz[0][i];
        }

        for (int i =1; i<num_vect; i++){ //O(k²n²)
                for (int j=0; j<num_elem; j++){ //O(kn²)
                        encontrado=false; //O(1)
                        k=0; //O(1)
                        while (!encontrado && k<num_elem*i+j){ //O(kn)
                                if (vect_final [k]> matriz[i][j])
                                        encontrado=true; //O(1)
                                else
                                        k++; //O(1)
                        }

                        if (encontrado){
                                for (int p=num_elem*i+j-1; p>=k; p--){ //O(kn)
                                        vect_final[p+1]= vect_final[p]; //O(1)
                                }
                                vect_final[k]=matriz[i][j]; //O(1)
                        }
                        else {  // !encontrado
                                vect_final[num_elem*i+j]=matriz[i][j]; //O(1)
                        }
                }
        }
        return vect_final; //O(1)
}
```

La eficiencia de este algoritmo viene dada por el bucle while que, en el peor de los casos, obliga a k a pasar desde 0 a n*k+k, luego, su eficiencia es O(kn). Este bucle se encuentra dentro de un bucle for que pasa por todos los valores de n, por tanto, tendríamos una eficiencia de $O(k*n^2)$. Finalmente, este bucle se encuentra dentro de otro que pasa por todos los valores de k. Finalmente, podemos concluir con que la eficiencia de la función es **$O(n^2*k^2)$**, siendo esta la eficiencia del algoritmo de **fuerza bruta**.

- **Eficiencia empírica**

Para el estudio de la eficiencia empírica hemos observado dos comportamientos.

En primer lugar, cuando dejamos el número de vectores constante y vamos incrementando el número de elementos por vector. En este caso, el comportamiento es **polinómico de grado 2**, como bien se muestra en la eficiencia teórica.

A continuación, mostramos una tabla con los tiempos de ejecución de la solución del problema 2 con fuerza bruta. Se aprecia cómo mantenemos k=10 constante y el número de elementos (n) se va incrementando de cien en cien hasta 3300.

| Nº elementos (k=5) | Tiempo (s) |
|---|---|
| 100 | 0.001108 |
| 200 | 0.004459 |
| 300 | 0.01188 |
| 400 | 0.017914 |
| 500 | 0.029407 |
| 600 | 0.040503 |
| 700 | 0.055097 |
| 800 | 0.072375 |
| 900 | 0.090962 |
| 1000 | 0.112409 |
| 1100 | 0.134817 |
| 1200 | 0.17301 |
| 1300 | 0.188402 |
| 1400 | 0.216939 |
| 1500 | 0.249403 |
| 1600 | 0.290932 |
| 1700 | 0.322494 |
| 1800 | 0.368528 |

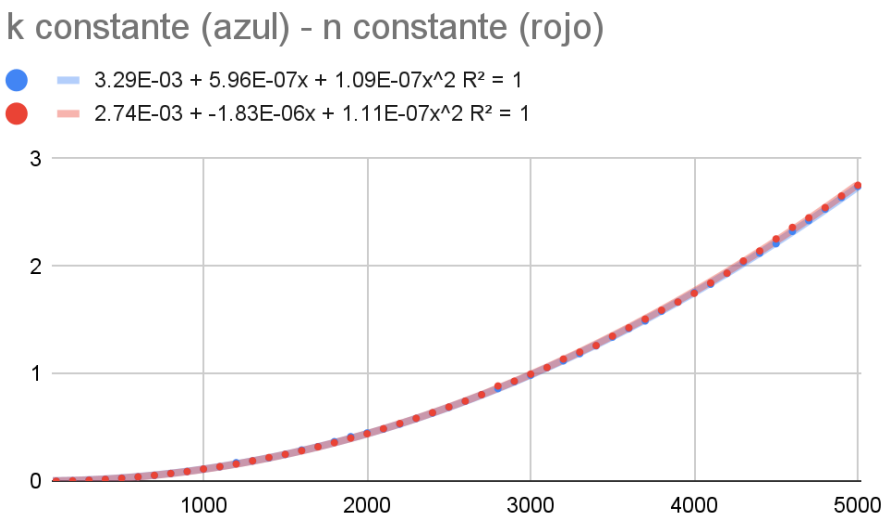
| | |
|---|---|
| 1900 | 0.414864 |
| 2000 | 0.449492 |
| 2100 | 0.485233 |
| 2200 | 0.531798 |
| 2300 | 0.583751 |
| 2400 | 0.634814 |
| 2500 | 0.687559 |
| 2600 | 0.744887 |
| 2700 | 0.80493 |
| 2800 | 0.861918 |
| 2900 | 0.925451 |
| 3000 | 0.985487 |
| 3100 | 1.05627 |
| 3200 | 1.11988 |
| 3300 | 1.1869 |

Por otro lado, mostramos la tabla cuando dejamos n=10 constante y el número de vectores (k) va incrementando de cien en cien hasta 3300. Los tiempos son muy similares. Esto se debe a que la eficiencia sigue siendo polinómica de grado 2.

| Nº vectores (n=10) | Tiempo (s) |
|---|---|
| 100 | 0.001102 |
| 200 | 0.004477 |
| 300 | 0.010064 |
| 400 | 0.017908 |
| 500 | 0.027962 |
| 600 | 0.04048 |
| 700 | 0.055288 |
| 800 | 0.072195 |
| 900 | 0.090393 |
| 1000 | 0.113926 |
| 1100 | 0.135757 |
| 1200 | 0.161042 |
| 1300 | 0.189647 |
| 1400 | 0.22052 |
| 1500 | 0.250416 |
| 1600 | 0.284486 |
| 1700 | 0.320436 |

| | |
|---|---|
| 1800 | 0.359159 |
| 1900 | 0.402216 |
| 2000 | 0.441934 |
| 2100 | 0.488621 |
| 2200 | 0.536784 |
| 2300 | 0.585345 |
| 2400 | 0.63771 |
| 2500 | 0.690651 |
| 2600 | 0.745405 |
| 2700 | 0.805139 |
| 2800 | 0.886167 |
| 2900 | 0.930595 |
| 3000 | 0.996237 |
| 3100 | 1.0575 |
| 3200 | 1.13675 |
| 3300 | 1.20124 |

Finalmente, en la siguiente gráfica se aprecia muy bien lo comentado anteriormente. Ambas gráficas son polinómicas y con tiempos realmente similares.



- **Eficiencia híbrida**

La función correspondiente al orden $O(k^2*n^2)$. Por tanto, para los datos en los que el número de vectores es constante (k=10), nos va a quedar una función de la siguiente forma: $f(x) = 100*a0*x^2+a1*x+a2$.

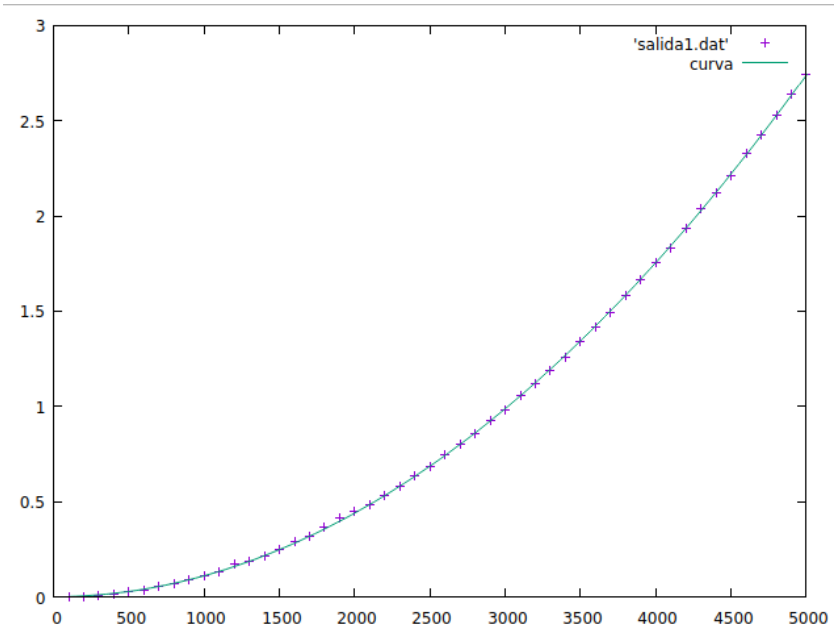
Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=100*a0*x*x+a1*x+a2**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1,a2**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Final set of parameters
==========
a0 = 1.09309e-09
a1 = 5.95811e-07
a2 = 0.0032876

La función quedaría de la forma:
**$f(x)=100*(1.09309e-09)*x^2+(5.95811e-07)*x+0.0032876$**

Y la gráfica sería la siguiente:



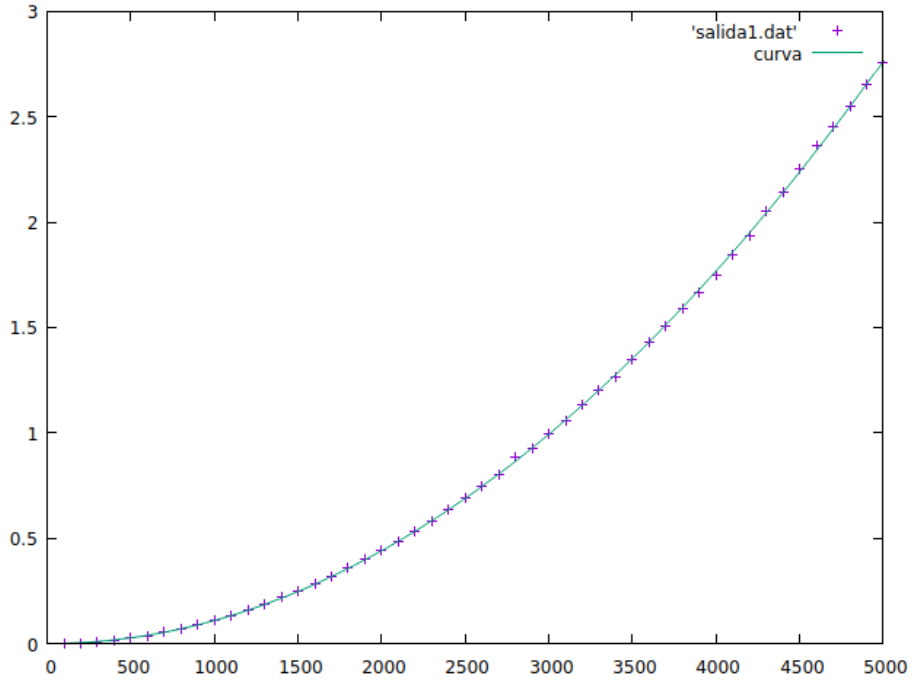

En el caso en el que el número de elementos es constante (n=10), nos va a quedar una función de la siguiente forma: **$f(x) = 100*a0*x^2+a1*x+a2$**.

Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=100*a0*x*x+a1*x+a2**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1,a2**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Final set of parameters
==========
a0 = 1.10745e-09
a1 = -1.83265e-06
a2 = 0.0027426

Y la función resultante: **f(x)=100*(1.10745e-09)*x²+(-1.83265e-06)*x+0.0027426**
Y la gráfica resultante:



## Algoritmo con divide y vencerás

- **Código con explicación**

Para resolver el problema 2 con Divide y Vencerás se utilizan las dos funciones detalladas abajo. En el main se llamará a la función divide con los parámetros l y r, que representan el intervalo de vectores que se van a mezclar; n es el número de elementos; el vector salida es el que se devolverá como solución del problema; y la matriz arr es aquella que contiene los vectores generados aleatoriamente en sus filas.

Cada vez que se llama a divide, dividimos el intervalo en la mitad izquierda y derecha recursivamente, de manera que conseguimos que se vayan ordenando y almacenando en el vector salida. Una vez conseguido, se llama a la función mezcla que se encarga de fusionar las dos mitades.

- **Eficiencia teórica**

```
void mezclar(int l, int r, int n, vector<int>& salida)     //O(n*k)
{
        // almacena el inicio de los dos vectores
        int inic_i = l * n;                         //O(1)
        int inic_d = ((l + r) / 2 + 1) * n;         //O(1)

        // almacena el tamaño de los dos vectores
        int tam_i = ((l + r) / 2 - l + 1) * n;      //O(1)
        int tam_d = (r - (l + r) / 2) * n;          //O(1)

        // vectores temporales
        int vector_i[tam_i], vector_d[tam_d]; //O(1)

        // guardando los datos en el primer vector
        for (int i = 0; i < tam_i; i++) //O(n)
                vector_i[i] = salida[inic_i + i];

        // guardando los datos en el segundo vector
        for (int i = 0; i < tam_d; i++)   //O(n)
                vector_d[i] = salida[inic_d + i];

        // contadores de los vectores
        int cont_i = 0, cont_d = 0;   //O(1)

        // cotador del vector salida
        int in = inic_i;      //O(1)
```

```
        // mezclar los dos vectores
        while (cont_i + cont_d < tam_i + tam_d)   //O(n*k)
        {
                if ( cont_d == tam_d || (cont_i != tam_i && vector_i[cont_i] <
        vector_d[cont_d]))
                {
                        salida[in] = vector_i[cont_i];          //O(1)
                        cont_i++;                               //O(1)
                        in++;                                   //O(1)
                }
                else{
                        salida[in] = vector_d[cont_d];          //O(1)
                        cont_d++;                               //O(1)
                        in++;                                   //O(1)
                }
        }
}

void divide(int l, int r, int n, vector<int>& salida, int** arr){
        if (l == r) {
                // inicializar el vector salida
                for (int i = 0; i < n; i++)                     //O(n)
                        salida[l * n + i] = arr[l][i];          //O(1)
        }
        else{
                // ordena la mitad izquierda                    //T(n/2)
                divide(l, (l + r) / 2, n, salida, arr);

                // ordena la mitad derecha
                divide((l + r) / 2 + 1, r, n, salida, arr);     //T(n/2)

                // mezclar las dos mitades
                mezclar(l, r, n, salida);                       //O(n*k)
        }
}
```

Por un lado, la función *mezclar* tiene una eficiencia O(n*k), ya que en su interior sólo hace asignaciones O(1), dos bucles for de n iteraciones y un bucle while de n*k pasos, luego por la regla del máximo, se tiene que *mezclar* es O(n*k).

Por otro lado, la función *divide* en el peor caso entra al bloque else, donde aparece una estructura recursiva para dar con el vector final ordenado de n*k elementos. Primero realiza dos llamadas recursivas con cada mitad del vector y luego se mezclan llamando a *mezclar*. Quedando la siguiente ecuación recursiva:

**T(n) = 2*T(n/2) + O(n*k)**, de donde obtenemos → **O(n*k*logk)** por ser del grupo de eficiencias con coeficiente 2.

- **Eficiencia empírica**

Cuando resolvemos el problema con Divide y Vencerás, la eficiencia empírica resulta ligeramente distinta cuando lo ejecutamos con k constante y n incrementando, que cuando lo ejecutamos con la n constante.

En la siguiente tabla mostramos los tiempos de ejecución cuando el número de elementos es constante y el número de vectores va creciendo de cien en cien. En este caso, la eficiencia sería lineal.

| Nº elementos (k=5) | Tiempo (s) |
|---|---|
| 100 | 8.60E-05 |
| 200 | 0.000194 |
| 300 | 0.000622 |
| 400 | 0.000431 |
| 500 | 0.000537 |
| 600 | 0.00073 |
| 700 | 0.000898 |
| 800 | 0.000948 |
| 900 | 0.001097 |
| 1000 | 0.00119 |
| 1100 | 0.00132 |
| 1200 | 0.001554 |
| 1300 | 0.001671 |
| 1400 | 0.001737 |
| 1500 | 0.001953 |
| 1600 | 0.002068 |
| 1700 | 0.002437 |
| 1800 | 0.002291 |
| 1900 | 0.002433 |
| 2000 | 0.002587 |
| 2100 | 0.002779 |
| 2200 | 0.002921 |
| 2300 | 0.003152 |
| 2400 | 0.003248 |
| 2500 | 0.003428 |
| 2600 | 0.003533 |

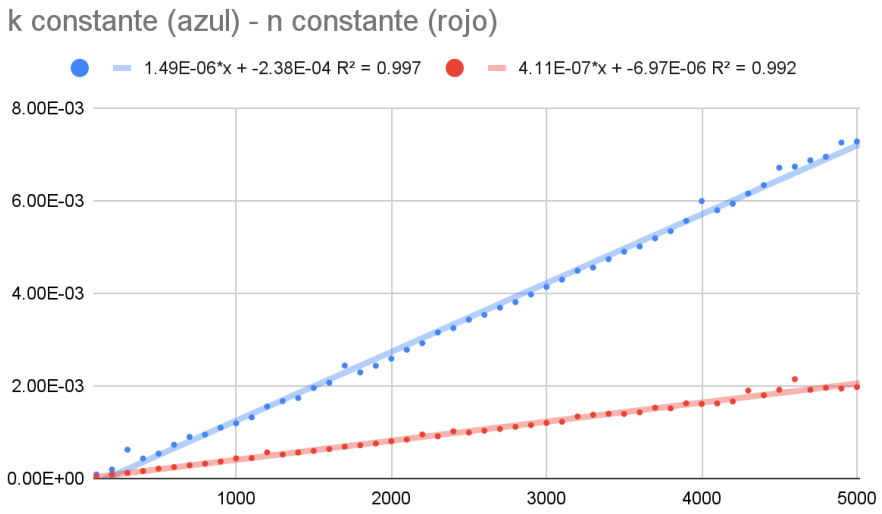
| | |
|---|---|
| 2700 | 0.003688 |
| 2800 | 0.003808 |
| 2900 | 0.003973 |
| 3000 | 0.004137 |
| 3100 | 0.004293 |
| 3200 | 0.004486 |
| 3300 | 0.004554 |

En la siguiente tabla mostramos los tiempos cuando el número de elementos es constante y es el número de vectores el que va cambiando. En este caso, la eficiencia sería O(nlogn).

| Nº vectores (n=10) | Tiempo (s) |
|---|---|
| 100 | 4.30E-05 |
| 200 | 8.60E-05 |
| 300 | 0.000121 |
| 400 | 0.00016 |
| 500 | 0.000213 |
| 600 | 0.000246 |
| 700 | 0.000287 |
| 800 | 0.00032 |
| 900 | 0.000365 |
| 1000 | 0.000436 |
| 1100 | 0.00044 |
| 1200 | 0.000564 |
| 1300 | 0.000518 |
| 1400 | 0.000561 |
| 1500 | 0.000596 |
| 1600 | 0.000635 |
| 1700 | 0.000687 |
| 1800 | 0.000718 |
| 1900 | 0.000755 |
| 2000 | 0.000804 |
| 2100 | 0.00084 |
| 2200 | 0.00095 |
| 2300 | 0.000911 |
| 2400 | 0.001018 |
| 2500 | 0.000992 |

| | |
|---|---|
| 2600 | 0.001031 |
| 2700 | 0.00107 |
| 2800 | 0.001116 |
| 2900 | 0.001151 |
| 3000 | 0.0012 |
| 3100 | 0.001225 |
| 3200 | 0.00134 |
| 3300 | 0.001376 |

En la siguiente gráfica vemos más claramente lo comentado anteriormente y cómo cambian los tiempos cuando la variable que dejamos constante es otra.



- **Eficiencia híbrida**

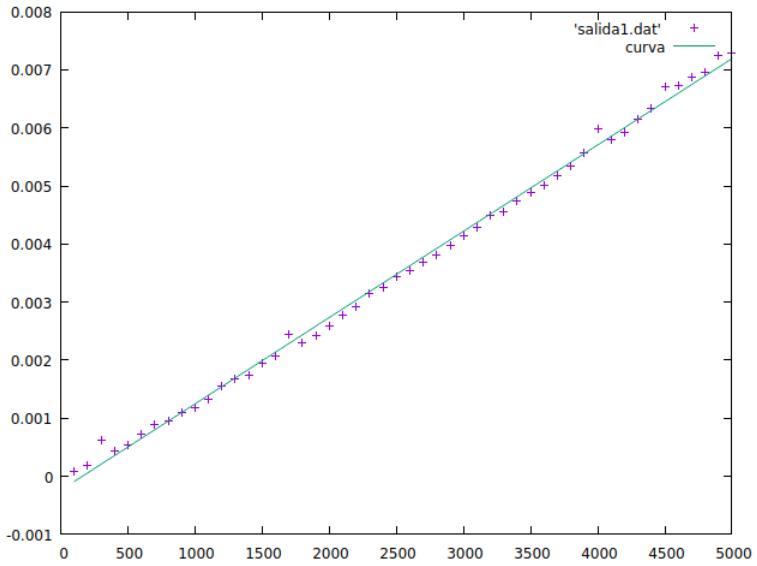
La función correspondiente al orden O(n*k*log(k)). Por tanto, para los datos en los que el número de vectores es constante (k=10), nos va a quedar una función de la siguiente forma: f(x) = a0*x*10*log(10)+a1, por tanto, lineal.

Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=a0*x*10*log(10)+a1**
- **gnuplot> fit f(x) 'salida.dat' via a0, a1**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Final set of parameters
==========
a0 = 6.45196e-08
a1 = -0.00023827

Con este resultado, la función quedaría de la forma:
**f(x)=(6.45196e-08)*x*10*log(10)-0.00023827**.
Y con la última instrucción conseguimos la siguiente gráfica:



Por otro lado, para los datos en los que el número de elementos es constante (n=10), nos va a quedar una función de la siguiente forma: f(x)=10*a0*x*log(x)+a1.

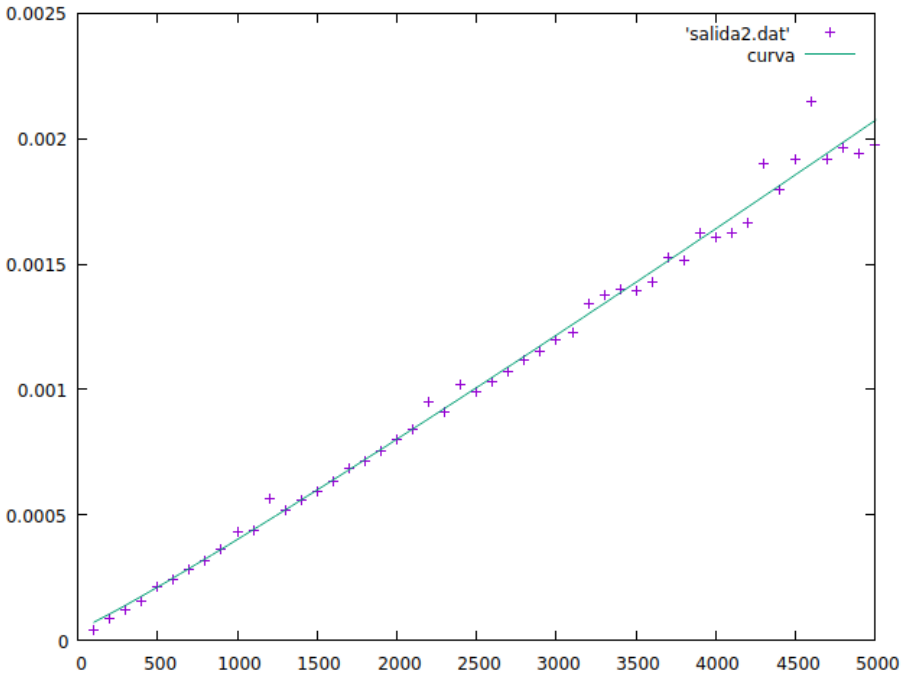
Introducimos entonces los siguientes comandos en gnuplot:

- **gnuplot> f(x)=10*a0*x*log(a1*x)+a2**
- **gnuplot> fit f(x) 'salida.dat' via a0, a1,a2**
- **gnuplot> plot 'salida1.dat', f(x) title 'Curva ajustada'**

Final set of parameters
==========
a0 = 2.90333e-09
a1 = 233.468
a2 = 4.40115e-05

Luego, la función quedaría como:
**f(x)=10*(2.90333e-09)*x*log(233.468*x)+4.40115e-05**.

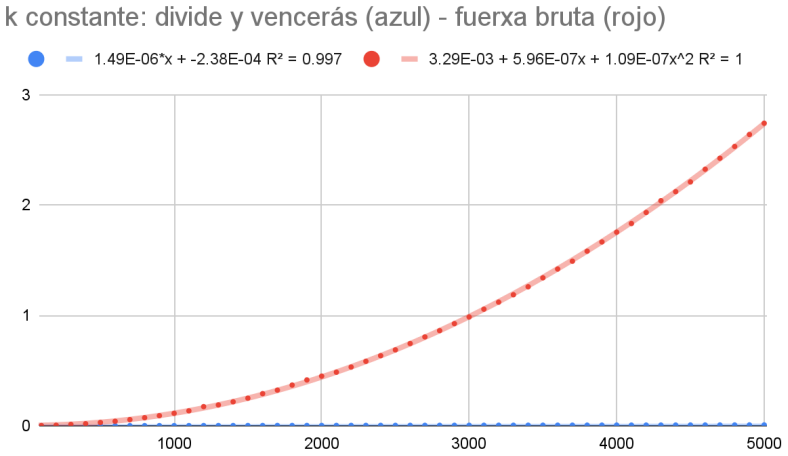
La gráfica sería la siguiente:



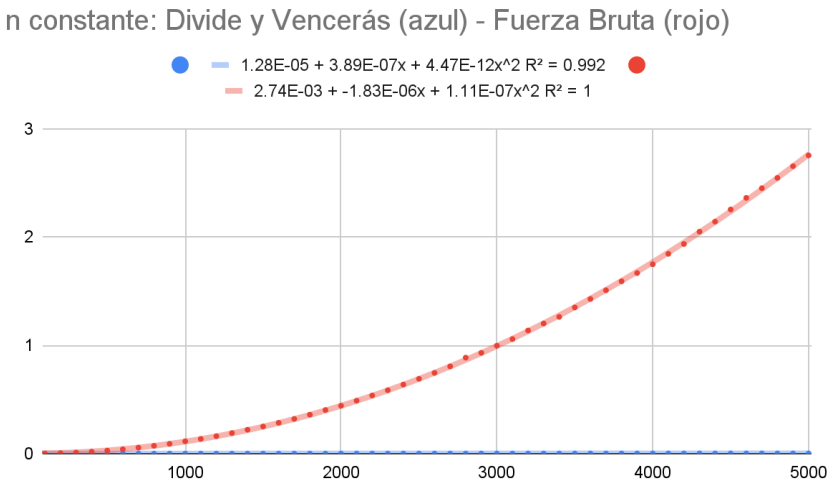
## Comparación Fuerza Bruta y Divide y Vencerás

Como veremos a continuación, en este caso, el algoritmo de Divide y Vencerás da tiempos mucho menores que el algoritmo de Fuerza Bruta.

En el caso en el que dejamos el número de vectores constante, el algoritmo de Fuerza Bruta se corresponde con una eficiencia de $O(n^2)$, mientras que el de Divide y Vencerás se corresponde con $O(n)$. En la gráfica veremos la gran diferencia de tiempos:




Cuando es el número de elementos por vector el que dejamos constante, el algoritmo de Fuerza Bruta sigue quedando con eficiencia de $O(n^2)$, mientras que el de Divide y Vencerás tiene una eficiencia de $O(n\log n)$. Luego, los tiempos siguen siendo muchos menores para el segundo tipo. También se aprecia en la gráfica:



# 4. Conclusión

Tras la realización de los ejercicios de esta práctica, hemos llegado a una serie de conclusiones muy interesantes, ya que hemos podido aplicar la teoría vista en clase.

Hemos llegado a ver la gran importancia de la implementación de un algoritmo eficiente, pues en tiempo de ejecución reduce mucho el trabajo. No obstante, hay que evaluar bien si nos interesa utilizar algoritmos tan fuertes en problemas simples, teniendo en cuenta que para valores por debajo de un determinado umbral pueden tener mayor o igual eficiencia algoritmos de fuerza bruta.

Seguimos recalcando la importancia de la eficiencia, ya que nosotras aquí solo hemos trabajado con una cantidad de datos muy pequeña y, esto en la vida real, con problemas reales no es así, ya que se trabaja con un volumen de datos muy grande. Por lo tanto, utilizar un algoritmo bajo en eficiencia nos llevaría a no poder resolver el problema planteado. En consecuencia, nos ha parecido interesante entender y aprender a cómo se resolvería un problema inicial grande subdividiendo en varios subproblemas que, con la tecnología adecuada, los pudiésemos ejecutar a la vez y así poder ahorrar un tiempo de ejecución que sería clave.

Una de las cosas que más nos ha sorprendido es que al ejecutar el programa para tamaños pequeños, el algoritmo divide y vencerás tardaba más que el de fuerza bruta, llegando a pensar que habíamos planteado el problema de manera errónea, ya que tendemos a fijarnos únicamente en la eficiencia del algoritmo y nunca habíamos considerado estudiar su comportamiento para distintos tamaños de entrada.

Sobre el Ejercicio 1 hemos visto que el problema planteado da resultados más rápidos con Divide y Vencerás cuando $n \geq 1$. Además, cuando los enteros se pueden repetir, el algoritmo de Divide y Vencerás aumenta el tiempo de ejecución, hasta el punto en que el algoritmo de Fuerza Bruta es más eficiente.

En el Ejercicio 2 se nos pregunta si hay un algoritmo más eficiente que el de Fuerza Bruta y, como hemos visto, sí lo hay. Un algoritmo Divide y Vencerás es notablemente más eficiente, de ahí la importancia de esta técnica y saber utilizarla correctamente, ya que nos puede llevar a la posibilidad de resolver un problema que con Fuerza Bruta, por la eficiencia que tiene, es imposible de resolverlo.

En definitiva, no nos podemos quedar sin ahondar lo suficiente en un problema, es decir, cuando nos dan un ejercicio que debemos hallar su solución, lo más común es ir a lo más sencillo de implementar o lo que primero se nos ocurre, que en este caso sería la resolución por fuerza bruta. Pero eso sería quedarnos solo en la superficie y no adentrarnos en el problema para poder hallar una solución óptima. Por lo tanto, a la hora de hacer un algoritmo, debemos de optar siempre por la técnica que más nos convenga para poder reducir el tiempo de ejecución lo máximo posible. En este caso, hemos aplicado la técnica de Divide y Vencerás, que ha conseguido que el tiempo de ejecución de un algoritmo mejore considerablemente.