UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2022 - 2023

Algorítmica

Memoria de prácticas

Práctica 2: Divide y vencerás

Autores:

- Mario Martín Rodríguez mario10@correo.ugr.es
- Juan Ayuso Arroyave juanayuso@correo.ugr.es
- Mario Líndez Martínez mariolindez@correo.ugr.es

Índice

1. Autores:

- Mario Líndez Martínez: 40%
- Mario Martín Rodríguez: 30%
- Juan Ayuso Arroyave: 30%

El algoritmo específico fue realizado entre Mario Líndez y Mario Martín y el algoritmo Divide y Vencerás entre Mario Líndez y Juan Ayuso.

El cálculo de umbrales nos lo hemos repartido, realizando uno cada uno.

La memoria la ha realizado Juan Ayuso con cierta ayuda de los demás.

El PowerPoint lo ha hecho Mario Martín.

La eficiencia ha sido realizada por los tres con su propio portátil, pero al final nos hemos quedado con los datos del de Juan Ayuso.

La eficiencia teórica del DyV la ha calculado Mario Líndez.

Mario Líndez ha hecho el dibujo explicativo del método de las tangentes.

2. Objetivos

El objetivo de esta práctica es el uso de la técnica conocida como “Divide y Vencerás” sobre un algoritmo que debe ser solución al problema planteado. Para ello, en primer lugar, hemos diseñado un algoritmo que resuelve el problema y sobre dicho algoritmo hemos aplicado la técnica “Divide y Vencerás” para aprender su utilización correcta y poder comprobar cómo la eficiencia mejora respecto al algoritmo original.

Para ver esto último hemos implementado un generador de casos y hemos estudiado la eficiencia teórica, empírica e híbrida (tanto del algoritmo específico como del “Divide y Vencerás”). Para finalizar hemos calculado los umbrales.

3. Definición del problema:

3.1. Enunciado

Los diseñadores de un videojuego ubicado en la “Tierra Media” de J.R.R. Tolkien quieren implementar el reparto de vituallas a hobbits y enanos dispersos en territorio hostil, controlado por los orcos. Los enanos han conseguido domar varios dragones y los utilizan para sobrevolar el territorio enemigo y lanzar paquetes con provisiones desde ellos. Tras cada incursión pueden conocer la ubicación exacta en que aterriza cada uno de sus paquetes. Pero quieren caracterizar la superficie total cubierta en cada envío. Es decir, considerando como entrada el conjunto P de puntos pi = (xi, yi) donde han caído cada uno de los n paquetes lanzados, determinar el polígono convexo de menor superficie que incluye todos los puntos.

Esto se traduce de la siguiente manera:

Nosotros contamos con una serie indeterminada de puntos aleatorios y desordenados, de los cuales queremos conocer la envolvente conexa, es decir, queremos saber de qué forma tenemos que unir los puntos para que todos ellos estén contenidos en el polígono formado por la unión y que el área sea la menor posible.

3.2. Entorno de análisis

El análisis se realizará desde el ordenador de Juan Ayuso Arroyave, cuyas especificaciones son las siguientes:

- Nombre: HP Laptop 15s-eq2xxx
- Procesador: AMD® Ryzen 7 5700u with radeon graphics × 16
- Disco Duro Sólido (SSD) de 1 TB
- Memoria RAM: 16 GB
- Sistema Operativo: Ubuntu 22.04.1 LTS de 64 bits
- Compilador: g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

3.3. Medición de tiempos

Para medir los tiempos hemos usado la biblioteca de la STL chrono, la cual permite calcular los tiempos de ejecución de forma más precisa. La implementación es la siguiente:

```
#include <chrono>
using std::chrono;

high_resolution_clock::time_point t_antes, t_despues;
duration<double> transcurrido;
t_antes = high_resolution_clock::now();
//sentencia o programa a medir
t_despues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>> (t_despues - t_antes);
cout << "el tiempo empleado es " << transcurrido.count() << " s." << endl;
```

4. Algoritmo Específico:

4.1. Realización:

Para la resolución de ambos problemas nos apoyaremos de una creada clase Punto, la cual se implementa de la siguiente forma:

```
class Punto {
private:
    int x;
    int y;
    offset {x,y}
}
```

Donde offset es una estructura de apoyo que nos servirá a la hora de ordenar el vector por su ángulo, ya lo veremos más adelante.

Para la resolución del problema planteado hemos utilizado el algoritmo de Graham, el cual tiene una complejidad O(nlogn). El algoritmo consiste en lo siguiente:

En primer lugar, debemos encontrar el punto de menor valor en el eje Y. Si hay dos puntos con el mismo valor, se buscará el que tenga menor valor en el eje X entre estos puntos. Ese punto será el menor ordenado.

```
Punto MenorOrdenado_lims(const vector<Punto> & p, int inicio, int final) {
    Punto salida = p.at(inicio);
    int minimo = salida.getY();

    for(int i = inicio+1; i < final; ++i) {
        if(p.at(i).getY() <= minimo){
            if (p.at(i).getY() == minimo) {
                if(salida.getX() > p.at(i).getX()) {
                    minimo = p.at(i).getY();
                    salida=p.at(i);
                }
```

```
            } else {
                minimo = p.at(i).getY();
                salida = p.at(i);
            }

        }
    }

    return salida;
}
```

Tras esto, debemos ordenar el resto de los puntos (Pi) de forma creciente en función del ángulo formado entre el segmento $\overrightarrow{Ap_i}$ y el eje X. Sin embargo, no hace falta calcular dicho ángulo ya que simplemente podemos ordenarlos calculando tangentes y cotangentes (si los puntos están en el primer y tercer cuadrante usamos la tangente, si están en el segundo y cuarto cuadrante utilizamos la cotangente). Para ordenar usamos el método de ordenación Quicksort aplicado sobre el operator< de la clase Punto.

```
bool Punto::operator< (const Punto & otro) const{
    bool es_menor = false;

    int cuadrante = getCuadrante();

    if (cuadrante != otro.getCuadrante()) {  // Distinto Cuadrante
        if (cuadrante < otro.getCuadrante()) {
            es_menor = true;
        }
    } else {    // Mismo cuadrante
        if (cuadrante == 1 || cuadrante == 3){
            float esta_tan = abs((origen.y - this->y)/(abs(origen.x - this->x)*1.0));
            float otra_tan = abs((origen.y - otro.y)/(abs(origen.x - otro.x)*1.0));

            if (esta_tan <= otra_tan){
                es_menor = true;
            }
        } else {
            float esta_cotan = abs(origen.x - this->x)/(abs(origen.y - this->y)*1.0);
            float otra_cotan = abs(origen.x - otro.x)/(abs(origen.y - otro.y)*1.0);

            if (esta_cotan <= otra_cotan){
                es_menor = true;
            }
        }

    }
    return (es_menor);
}
```

Por último, calculamos la envolvente conexa. Para ello, vemos si para cada punto, el movimiento desde los dos puntos anteriores se trata de un giro a la derecha o a la izquierda. Si gira a la derecha, el segundo punto no pertenece a la envolvente y si gira a la izquierda sí.

Para ver hacia dónde es el giro, dados 3 puntos (x1,y1), (x2,y2), (x3,y3), usamos la siguiente fórmula:

$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

Si el resultado es 0, están alineados, si es positivo giras a la izquierda y si es negativo giras a la derecha.

```
bool GiroALaDerecha(Punto p1, Punto p2, Punto p3){

    bool salida = false;            // usamos la fórmula
```

```
    if((((p2.getX()-p1.getX())*(p3.getY()-p1.getY()))-((p2.getY()-p1.getY())*(p3.getX()-
p1.getX())))<0){

        salida = true;
    }

    return salida;
}
```

Esta función ordenará el vector de Puntos dado entre las posiciones inicial y final. Como bien indica su nombre la ordenación se realizará a partir del ángulo en función de su punto Menor Ordenado

```
void OrdenaPorAngulo_lims (vector<Punto> & p, int inicial, int final){
    // Primero ordenamos el vector según el ángulo formado, tomando como origen al punto
    // con la menor ordenada
    // Buscamos el punto con la menor ordenada y la seleccionamos como nuestro origen
    Punto origen = MenorOrdenado_lims(p, inicial, final);

    // O(n)
    for (int i = inicial; i < final; ++i){
        p.at(i).setOrigen(origen);
    }

    // O(nlog(n))
    quicksort_lims(p, inicial, final);
}
```

Implementación del cálculo de la envolvente conexa usando el algoritmo de Graham a parte de un vector.

```
vector<Punto> EnvolventeConexa_lims(vector<Punto> p, int inicial, int final){
    // Primero ordenamos según el angulo
    OrdenaPorAngulo_lims(p, inicial, final);

    //Calculamos la envolvente convexa
    Punto p1 = p.at(inicial);
    Punto p2 = p.at(inicial+1);
    Punto p3 = p.at(inicial+2);

    vector<Punto> salida;

    salida.push_back(p1);
    salida.push_back(p2);
    salida.push_back(p3);

    // cogemos los tres primeros puntos y partir de ahí vamos comprobando si el segmento que
    // forman p1 y p2 y el que forman p2 y p3 representan un dextrogiro (giro a la derecha)

    for(int i = inicial + 3; i < final; ++i){
        p1 = salida.at(salida.size() - 2);
        p2 = salida.back();
        p3 = p.at(i);
        while (salida.size() > 1 && GiroALaDerecha(p1,p2,p3)) {
            salida.pop_back();

            p1 = salida.at(salida.size() - 2);
            p2 = salida.back();
            p3 = p.at(i);
        }

        salida.push_back(p.at(i));

    }

    return (salida);
}
```

4.2. Eficiencia teórica:

La eficiencia teórica del algoritmo de Graham es de O(nlog(n)), como hemos mencionado previamente. Adaptándolo a nuestra implementación, debemos tener en cuenta la ordenación previa por el ángulo el cual posee una eficiencia O(nlog(n))Por lo tanto nuestro algoritmo será de eficiencia:

$$nlog(n) + nlog(n) = 2nlog(n) \in O(nlog(n))$$

4.3. Eficiencia empírica:

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Para este algoritmo, lo ejecutaremos para diferentes tamaños del vector de puntos para el cual calcularemos su envolvente conexa obteniendo así el tiempo de ejecución. Estos tiempos los almacenamos en un fichero salidaE.dat. El código es el siguiente:

```
#!/bin/bash
printf "" > salidaE.dat

i=50000
while [ "$i" -le 1300000 ]
do
          # Generamos los puntos
          ./generador $i

          # Ejecutamos los puntos
          ./especifico data.txt >> salidaE.dat

          echo "Terminado $i"

          i=$(( $i + 50000 ))
done
```

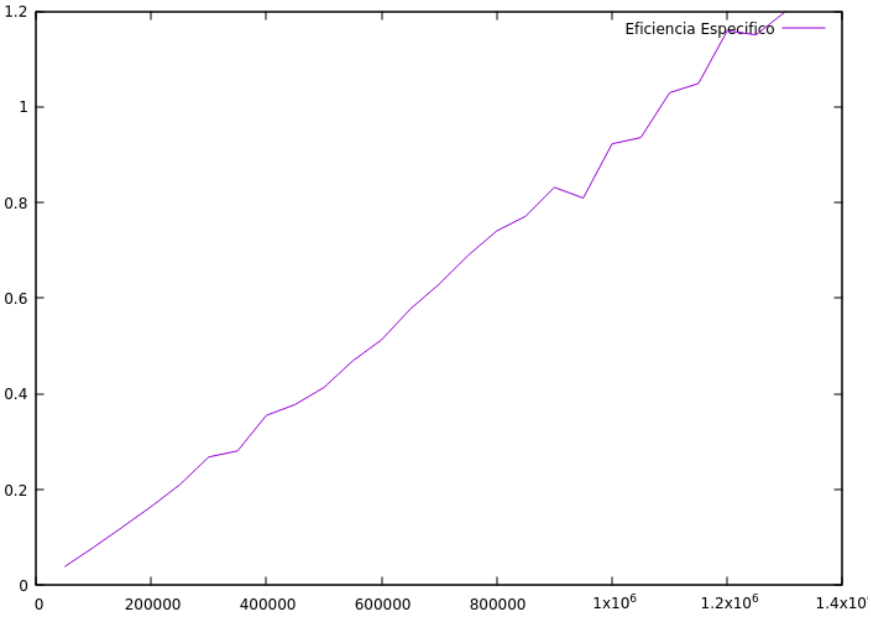
Empezamos con un tamaño base de 50000 puntos y vamos aumentándolo de 50000 en 50000 hasta llegar al tamaño de 1300000 puntos.

Los tiempos obtenidos son los siguientes:

Tamaño	Tiempo (seg)
50000	0,0389882
100000	0,079321
150000	0,121519
200000	0,164437
250000	0,210639
300000	0,268356
350000	0,280973
400000	0,355062
450000	0,377885
500000	0,413531
550000	0,469318
600000	0,513199
650000	0,577639
700000	0,629695
750000	0,689323
800000	0,740768
850000	0,771335
900000	0,832012
950000	0,809482
1000000	0,922623
1050000	0,935901
1100000	1,02946

1150000	1,04901
1200000	1,15917
1250000	1,15069
1300000	1,19815

A partir de dichos tiempos, queda la siguiente gráfica:



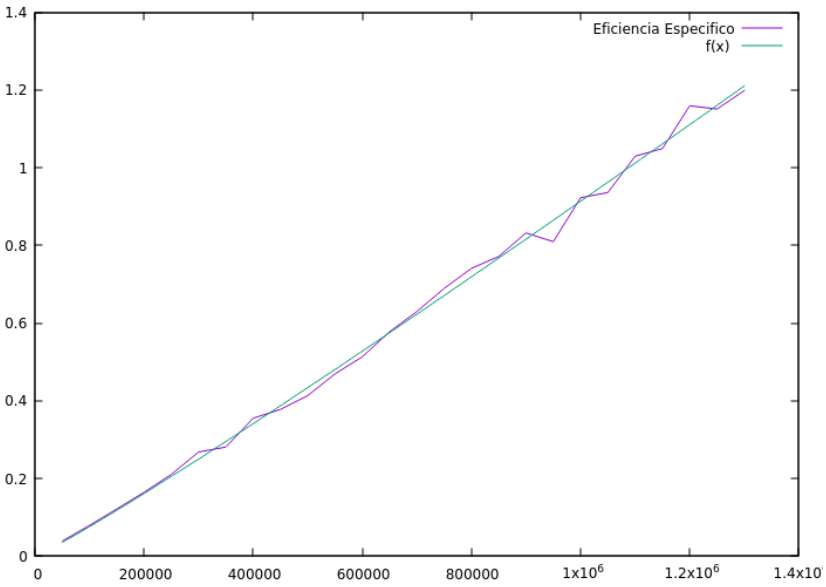
Gráfica de tiempos del algoritmo específico

4.4. Eficiencia híbrida:

Para la eficiencia híbrida, hemos realizado un ajuste con la función $f(x) = a_0 x log(x)$ y como resultado hemos obtenido las siguientes constantes ocultas:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 4.5831e-08       +/- 2.515e-10    (0.5487%)
```

Cuyo coeficiente de correlación es de 0,9982 por lo que el ajuste es casi perfecto



5. Algoritmo Divide y Vencerás

5.1. Realización:

Para aplicar el Divide y Vencerás, deberemos tener nuestro vector de puntos ordenado según la coordenada X, de forma creciente. Para la ordenación utilizamos el método Quicksort.

```
int comparePuntos (const void * a, const void * b) {
    Punto * p = (Punto *) a;
    Punto * q = (Punto *) b;

    int retorno = 0;

    if (p->getX() < q->getX()){
        retorno = -1;
    } else if (p -> getX() > q->getX()) {
        retorno = 1;
    }
    return (retorno);
}
```

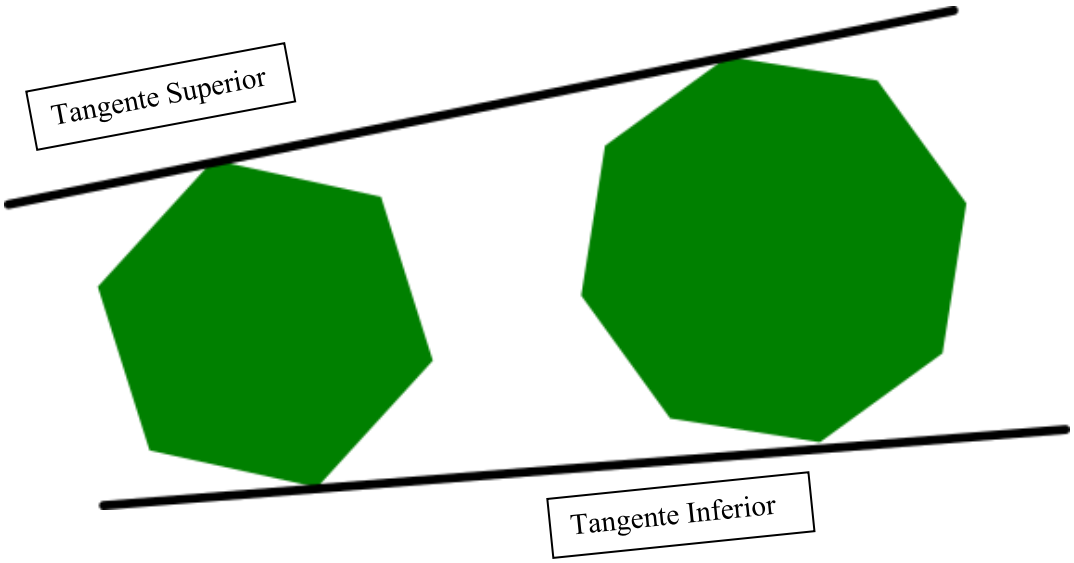
```
void OrdenaPorOrdenada (vector<Punto> & p){
    qsort(p.data(), p.size(), sizeof(Punto), comparePuntos);
}
```

Una vez ordenado, dividimos el vector por la mitad hasta que el número de puntos sea menor que 100, el cuál es el mínimo número de puntos que hemos considerado que debe tener cada partición.

A cada una de las divisiones le aplicamos el algoritmo de Graham explicado previamente en el algoritmo específico.

```
vector <Punto> DivideyVenceras_lims (vector<Punto> p, int inicial, int final){
    vector<Punto> solucion;
    if (final - inicial <= UMBRAL){
        solucion = EnvolventeConexa_lims(p, inicial, final);
    } else {
        int k = (final - inicial)/2;

        vector<Punto> U (p.begin(), p.begin()+k);

        vector<Punto> V (p.begin()+k, p.end());

        solucion = Fusion(DivideyVenceras_lims(U, 0, k), DivideyVenceras_lims(V, 0, final-k));
    }

    return (solucion);
}
```

Una vez calculadas las envolventes conexas, unimos las envolventes de todas las divisiones. Para ello calculamos la tangente superior e inferior entre los polinomios contiguos tanteando hasta obtenerlos. Con las tangentes obtenidas, sólo nos queda fusionar las envolventes, que bastaría con añadir los puntos desde la tangente inferior hasta la tangente superior del polinomio derecho y después añadir desde la tangente superior hasta la tangente inferior del izquierdo.




```
vector<int> CalculaTangentes(const vector<Punto> & izquierda, const vector<Punto> & derecha){
    // Vemos cuántos puntos hay en cada polinomio
    int n1 = izquierda.size();
    int n2 = derecha.size();

    // Buscamos el punto más a la derecha del polinomio izquierdo y el punto más a
    // la izquierda del polinomio derecho

    int a = 0;

    while (izquierda.at(a).getX() <= izquierda.at(a+1).getX() && a < n1){
        ++a;
    }

    int b = n2-1;
    while (derecha.at(b).getX() >= derecha.at(b-1).getX() && b > 0){
        --b;
    }

    // Calculamos la tangente superior
    int tsup_a = a, tsup_b = b;
    bool done = false;

    while (!done){
        done = true;

        // A partir del punto b, calcularemos el primer giro que no sea a la derecha
        // con puntos del polinomio de la izquierda
        while (GiroALaDerecha(derecha.at(tsup_b), izquierda.at(tsup_a),
izquierda.at((tsup_a+1)%n1))){
            tsup_a = (tsup_a + 1) % n1;
        }

        // Si tras encontrarlo el giro que se produce desde a, es a la izquierda, realizamos
        //el mismo proceso pero ahora para el polinomio de la derecha
        while (!GiroALaDerecha(izquierda.at(tsup_a), derecha.at(tsup_b), derecha.at((n2+tsup_b-
1)%n2))){
            tsup_b = (n2+tsup_b-1)%n2;
            done = false;
        }
    }

    // Calculamos la tangente inferior, cuyo proceso es análogo solo que a la inversa
    int tinf_a = a, tinf_b = b;
    done = false;

    while (!done){
        done = true;
```

```
        while (GiroALaDerecha(izquierda.at(tinf_a), derecha.at(tinf_b),
derecha.at((tinf_b+1)%n2))){
            tinf_b = (tinf_b + 1)%n2;
        }

        while (!GiroALaDerecha(derecha.at(tinf_b), izquierda.at(tinf_a),
izquierda.at((n1+tinf_a-1)%n1))){
            tinf_a = (n1+tinf_a-1)%n1;
            done = false;
        }

    }

    vector<int> salida = {tsup_a, tsup_b, tinf_a, tinf_b};

    return (salida);

}
```

Para unir ambas envolventes conexas, iteraremos desde la tangente inferior en círculo hasta cerrar la nueva envolvente. Dejaremos por el camino todos los puntos fuera de las tangentes.

```
vector<Punto> Fusion (const vector<Punto>& U, const vector<Punto> & V){
     vector<int> tangentes = CalculaTangentes(U, V); // O(n)

     int n1 = U.size();
     int n2 = V.size();

     vector<Punto> aux;

     int tsup_i = tangentes.at(0);   // Tangente superior izquierda
     int tsup_d = tangentes.at(1);   // Tangente superior derecha

     int tinf_i = tangentes.at(2);   // Tangente inferior izquierda
     int tinf_d = tangentes.at(3);   // Tangente inferior derecha

     // Aniadimos a salida desde el punto de la tangente inferior derecha hasta la
     // superior derecha     O(n/2)
     for (int i = tinf_d; i != tsup_d; i=(i+1)%n2){
         aux.push_back(V.at(i));
     }

     aux.push_back(V.at(tsup_d));

     // Insertamos en salida desde la tangente superior izquierda hasta la inferior
     // izquierda   O(n/2)
     for (int j = tsup_i; j != tinf_i; j=(j+1)%n1){
         aux.push_back(U.at(j));
     }

     aux.push_back(U.at(tinf_i));

     Punto menor_ordenada = MenorOrdenado(aux);

     int i = 0;
     bool encontrado_indice = false;

     // O(n)
     for (auto it = aux.begin(); it != aux.end(); ++it){
         it->setOrigen(menor_ordenada);
         if (!encontrado_indice && (*it) != menor_ordenada){
             ++i;
         } else {
             encontrado_indice = true;
         }
     }

     vector<Punto> salida;
     salida.push_back(aux.at(i));
```

```
    int tamanio = aux.size();

    // O(n)
    for (int j = (i+1)%tamanio; j != i; j=(j+1)%tamanio){
        salida.push_back(aux.at(j));
    }

    return (salida);
}
```

5.2. Eficiencia teórica:

La eficiencia teórica de nuestro algoritmo divide y vencerás vendrá dada por:

$$T(n) = \begin{cases} t(n) & si\ n < 100 \\ 2T\left(\frac{n}{2}\right) + F(n) & si\ n \geq 100 \end{cases}$$

Donde $t(n)$ es la eficiencia del caso base, es decir la eficiencia teórica del algoritmo de Graham, que es $O(nlog_2(n))$, como hemos mencionado previamente, mientras que F(n) es la eficiencia de la función de fusión, cuyo tiempo de ejecución será de $F(n) = 4n\ \epsilon\ O(n)$

Para hallar la eficiencia resolveremos la recursividad: $T(n) = 2T\left(\frac{n}{2}\right) + n$

Como F(n) tiene una eficiencia lineal podemos aplicar la fórmula maestra de tal forma que podemos concluir con que nuestro algoritmo es de orden $O(nlog_2(n))$ ya que $I = 2 = 2^1 = b^k$

5.3. Eficiencia empírica:

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Para el este algoritmo, lo ejecutaremos para diferentes tamaños del vector de puntos al cual debemos calcular la envolvente conexa y obtendremos el tiempo. Estos tiempos los almacenamos en un fichero dyv.dat. El código es el siguiente:

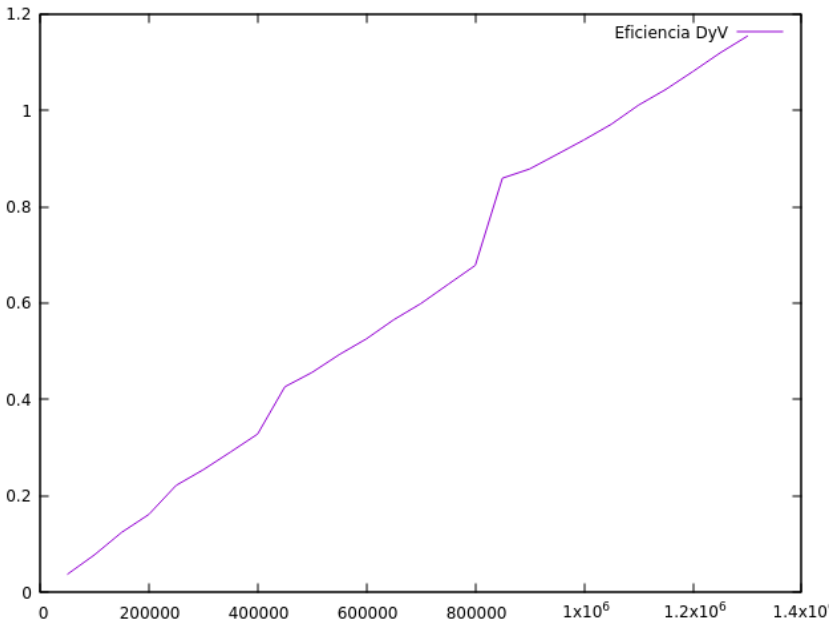
```
#!/bin/bash
printf "" > dyv.dat

i=50000
while [ "$i" -le 1300000 ]
do
          # Generamos los puntos
          ./generador $i

          # Ejecutamos los puntos
          ./dyv data.txt >> dyv.dat

          echo "Terminado $i"

          i=$(( $i + 50000 ))
done
```

Empezamos con un tamaño base de 50000 puntos y vamos aumentándolo de 50000 en 50000 hasta llegar al tamaño de 1300000 puntos.

Los tiempos obtenidos son los siguientes:

Tamaño	Tiempo (seg)
50000	0,0379006
100000	0,078322
150000	0,124648
200000	0,161875
250000	0,222204
300000	0,254597
350000	0,291347
400000	0,328745
450000	0,426585
500000	0,45631
550000	0,493431
600000	0,525796
650000	0,565312
700000	0,599045
750000	0,638797
800000	0,678233
850000	0,859093
900000	0,877915
950000	0,908031
1000000	0,938378
1050000	0,970747
1100000	1,01011
1150000	1,04272
1200000	1,0801
1250000	1,11873
1300000	1,15312

A partir de dichos tiempos, queda la siguiente gráfica:



Gráfica de tiempos del algoritmo DyV

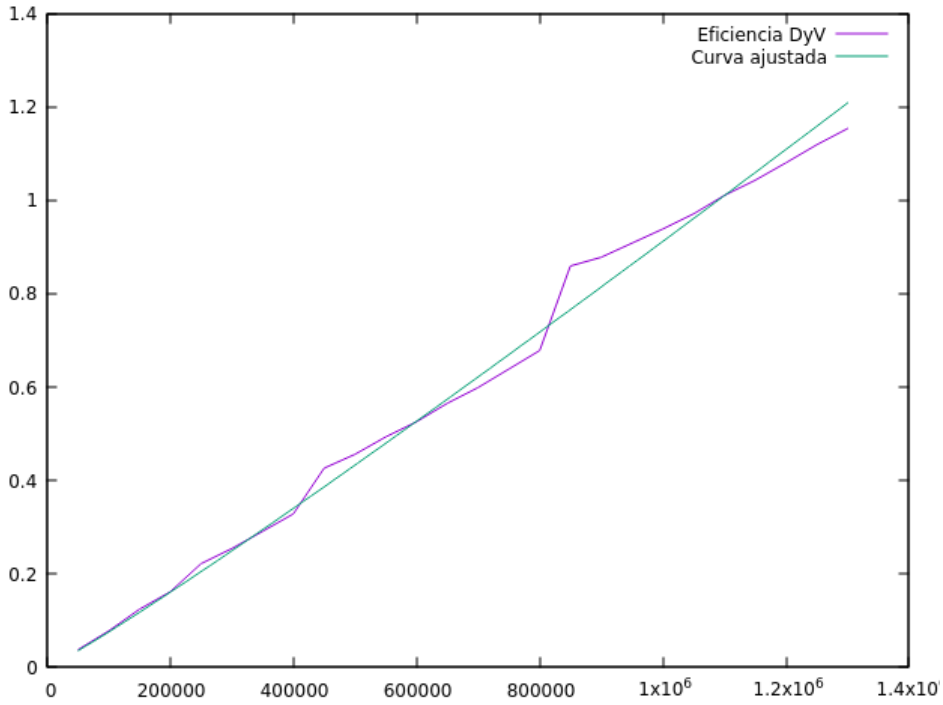
5.4. Eficiencia híbrida:

Para la eficiencia híbrida, hemos realizado un ajuste con la función f(x) = xlog(x) y como resultado hemos obtenido las siguientes constantes ocultas:

Constantes ocultas

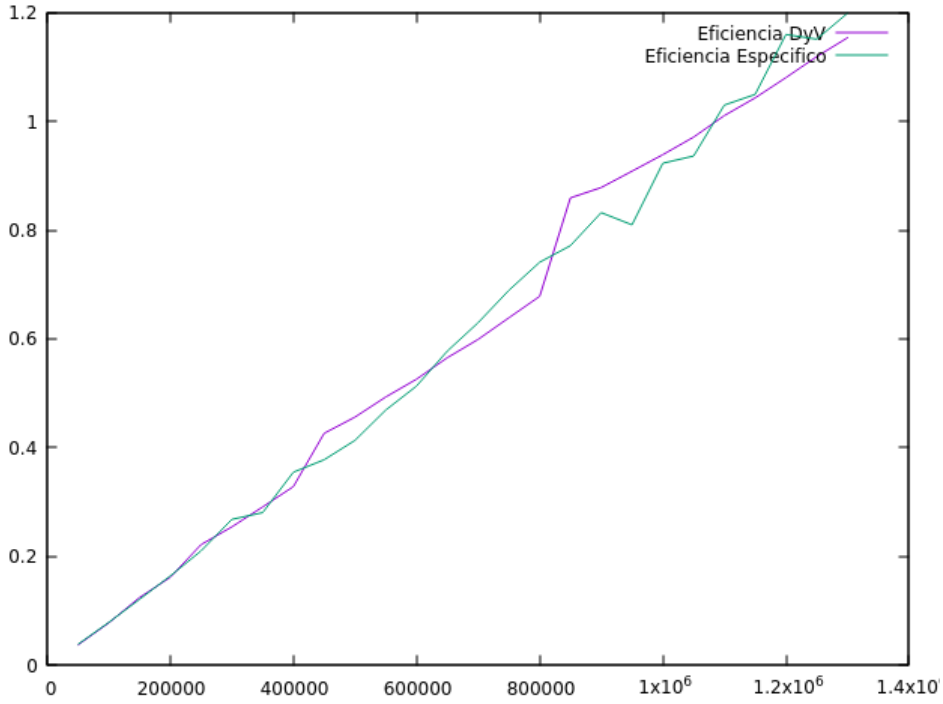
```
Final set of parameters            Asymptotic Standard Error
======================             ==========================
a0              = 4.57445e-08      +/- 4.213e-10    (0.921%)
```

Cuyo coeficiente de correlación es de 0.9962.



5.5. Cálculo de umbrales:

5.5.1. Umbral de tanteo:



En la gráfica podemos observar que conforme aumenta el tamaño del vector de puntos, el tiempo del algoritmo específico aumenta respecto al algoritmo DyV, aunque la diferencia es muy leve entre ambas.

5.5.2. Umbral teórico:

Compararemos el tiempo del algoritmo básico con el del Divide y Vencerás usando un solo nivel de recursividad.

$$t(n) = \begin{cases} h(n) = 2nlog_2(n) & si\ n < n_0 \\ 2t\left(\frac{n}{2}\right) + 4n & si\ n \geq n_0 \end{cases}$$

Por tanto:

$$2nlog_2(n) = nlog\left(\frac{n}{2}\right) + 4n \iff 2\log_2(n) = log\left(\frac{n}{2}\right) + 4 \iff \log_2(n^2) - \log_2\left(\frac{n}{2}\right) = 4 \iff$$

$$\iff \log_2\left(\frac{n^2}{\frac{n}{2}}\right) = \log_2\left(\frac{2n^2}{n}\right) = \log_2(2n) = \log_2(2) + \log_2(n) = 1 + \log_2(n) = 4 \iff$$

$$\iff \log_2(n) = 3 \iff n = 2^3 = 8$$

5.5.3. Umbral óptimo:

Para calcular el umbral óptimo tendremos que igualar las expresiones calculadas previamente en el análisis híbrido.

$$4{,}5831 \cdot 10^{-8} x log_2(x) = 4{,}57445 \cdot 10^{-8} x log_2(x)$$

Lo cual se cumple para 4 valores:

- 0.9999999991836

- 0.9999999991859

- 0.9999999992047

- 0.9999999992007

Luego podemos tomar como umbral n=1.

6. Conclusión

Hemos llegado a la conclusión de que al aplicar el método Divide y Vencerás sobre un algoritmo, no siempre vamos a tener garantizado mejorar su eficiencia frente al de una implementación específica.

Para que sí que nos sea útil hemos comprobado la importancia de analizar nuestro problema previamente, así como de conocer previamente el valor del umbral para poder decidir entre usar el método Divide y Vencerás y el específico.