

# Práctica 2

## Divide y Venceras

---

MARIO LÍNDEZ MARTÍNEZ

JUAN AYUSO ARROYAVE

MARIO MARTÍN RODRÍGUEZ

# *INDICE*

---

- Objetivos
- Definición del problema
- Algoritmo Específico
- Algoritmo Divide y Vencerás
- Comparación Eficiencias
- Conclusión

# Objetivos

---

# 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 para 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.

# Definición del problema

---

# 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  $p_i = (x_i, y_i)$  donde han caído cada uno de los  $n$  paquetes lanzados, determinar el polígono convexo de menor superficie que incluye todos los puntos.

# Traducción

---

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.

# Algoritmo Específico

---

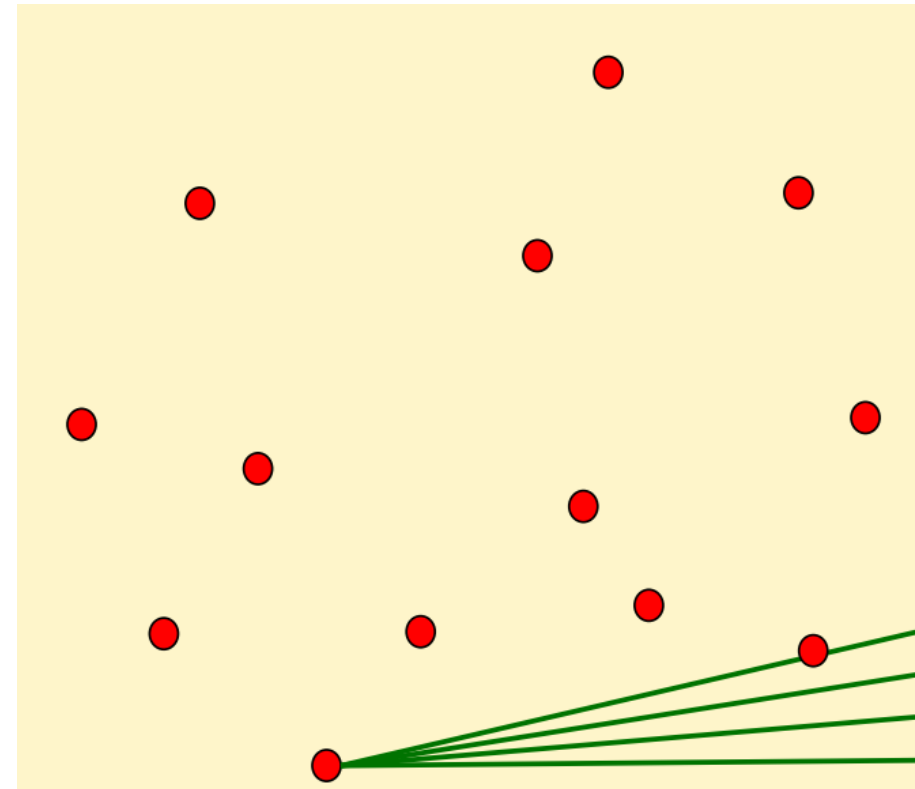


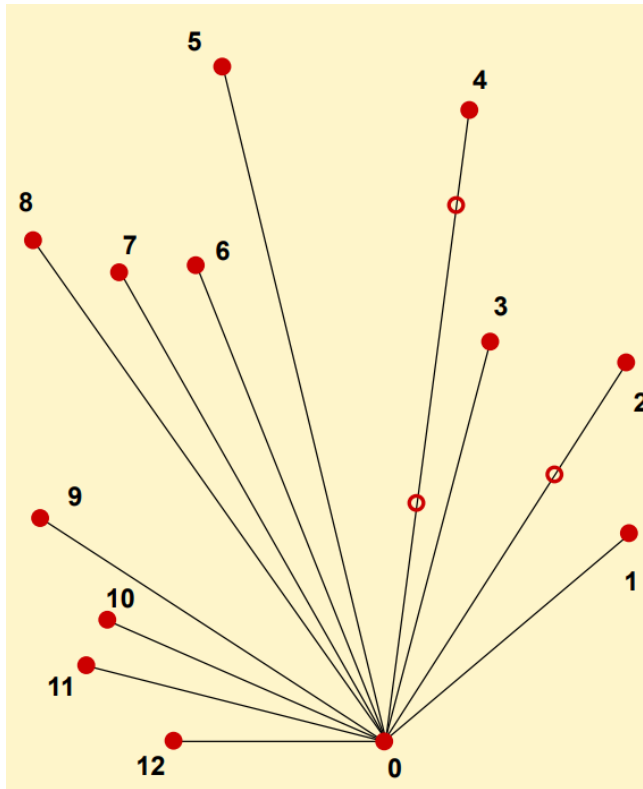
# Algoritmo Específico

---

Para la resolución del problema planteado hemos utilizado el algoritmo de Graham, el cual tiene una complejidad  $O(n \log n)$ . 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 busca el que tenga menor valor en el eje X entre estos puntos. A ese punto lo llamaremos A.



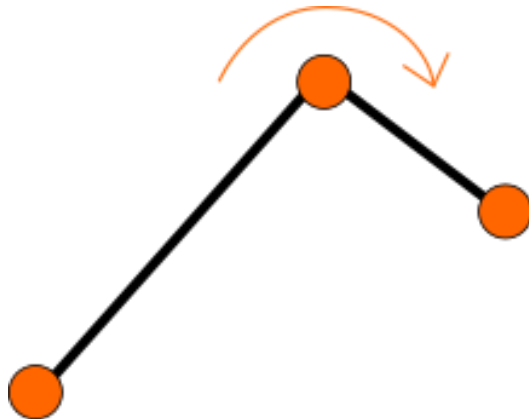


Tras esto, debemos ordenar el resto de los puntos ( $P_i$ ) de forma creciente en función del ángulo formado entre el segmento  $AP_i$  y el eje  $X$ .

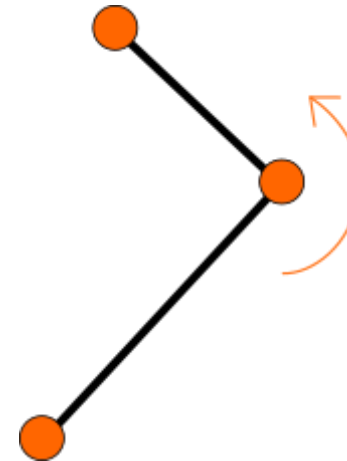
Sin embargo, no hace falta calcular dicho ángulo ya que simplemente podemos ordenarlos calculando tangentes y cotangentes (si los puntos están en el primer y tercer cuadrante usamos la tangente, si están en el segundo y cuarto cuadrante utilizamos la cotangente). Para ordenar usamos el método de ordenación quicksort.

---

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.



Giro a la derecha



Giro a la izquierda

---

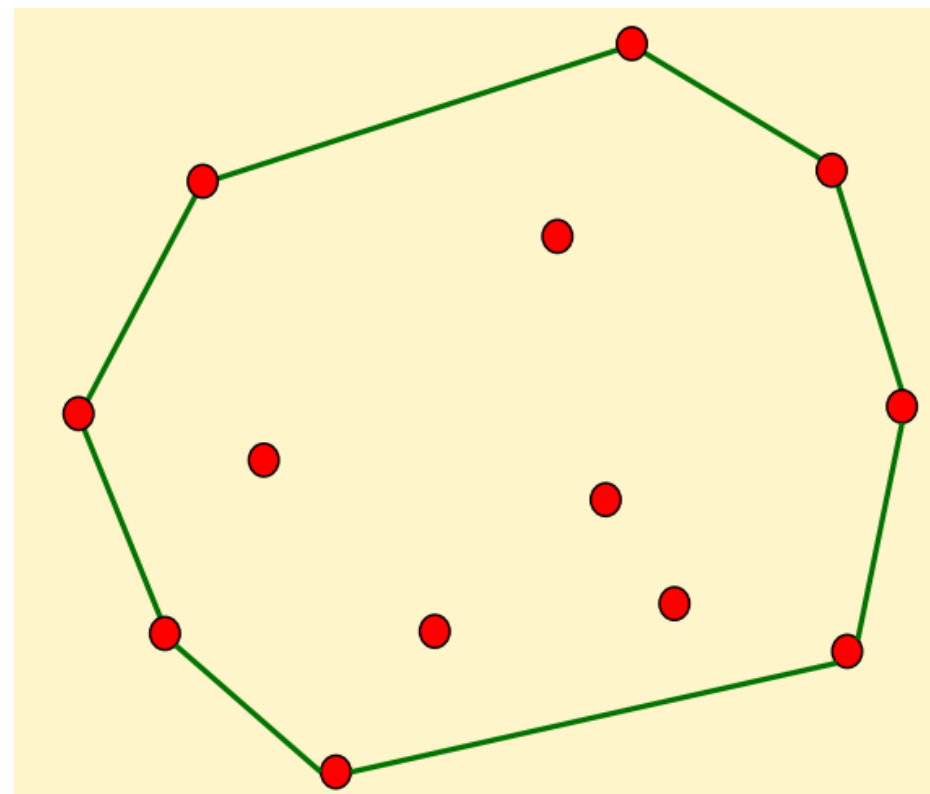
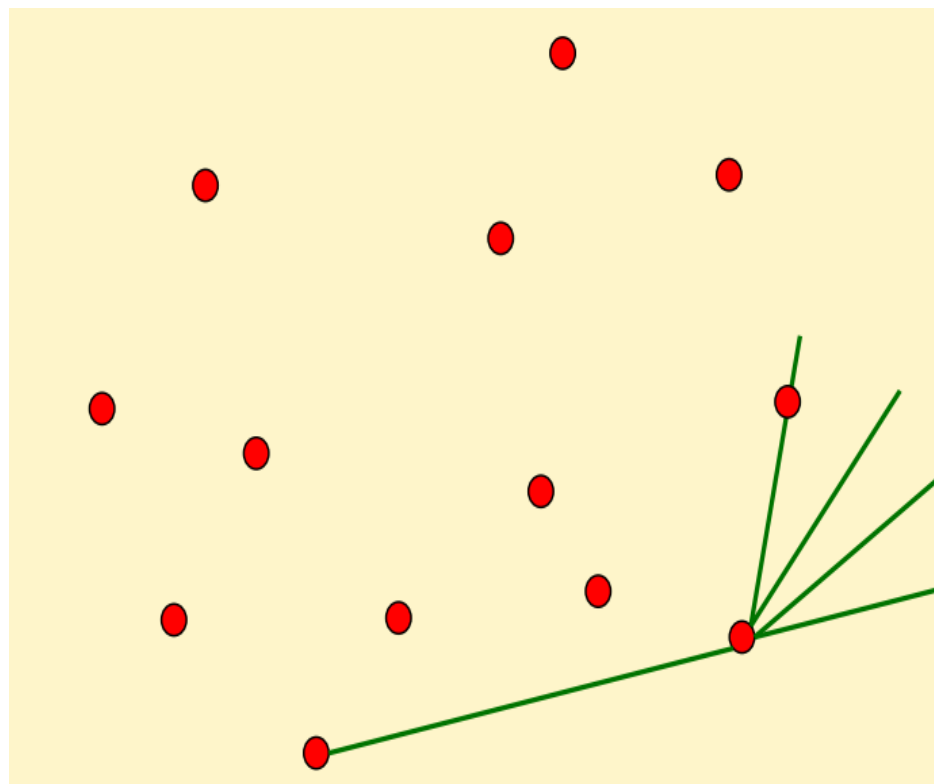
Para ver hacia dónde es el giro, dados 3 puntos  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ , usamos la siguiente fórmula:

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

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;           // uso la fórmula  
  
    if((((p2.getX()-p1.getX())*(p3.getY()-p1.getY()))-((p2.getY()-p1.getY())*(p3.getX()-p1.getX())))<0){  
        salida = true;  
    }  
  
    return salida;  
}
```



# Eficiencia Teórica

---

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

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

# 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 al cual debemos calcular la envolvente conexa y obtendremos el tiempo. 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
```



# Eficiencia Empírica

---

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:

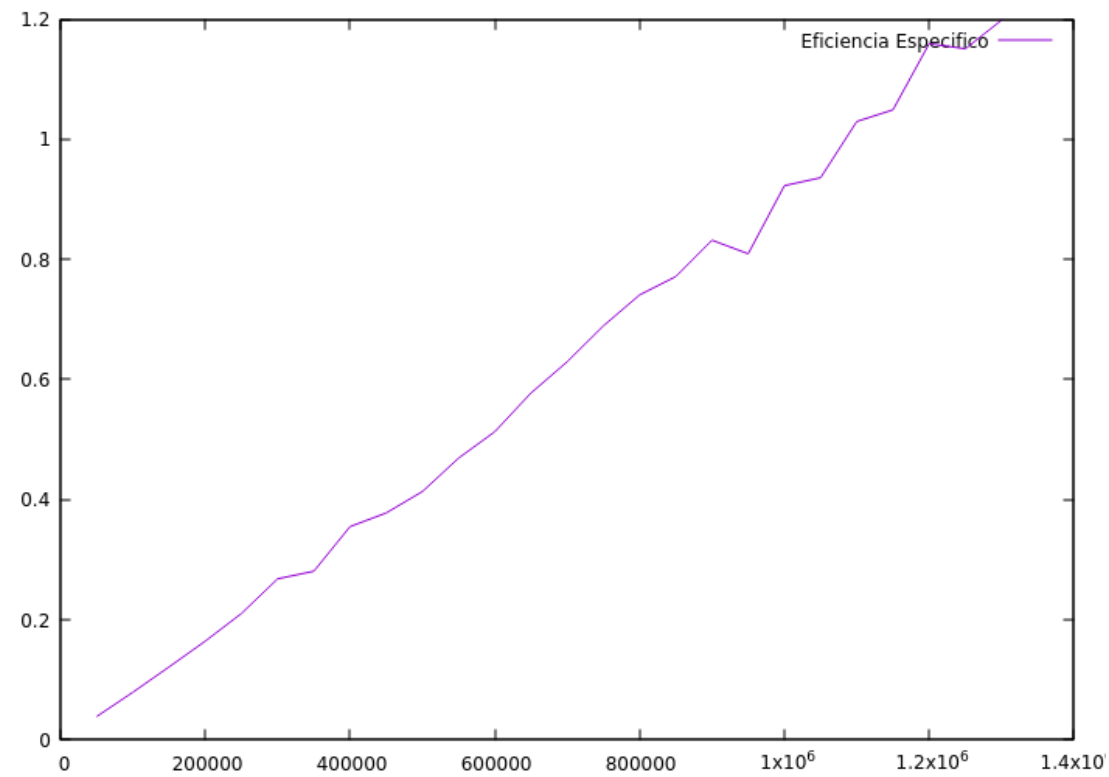
# Eficiencia Empírica

---

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

# Eficiencia Empírica

---



# Eficiencia Híbrida

---

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

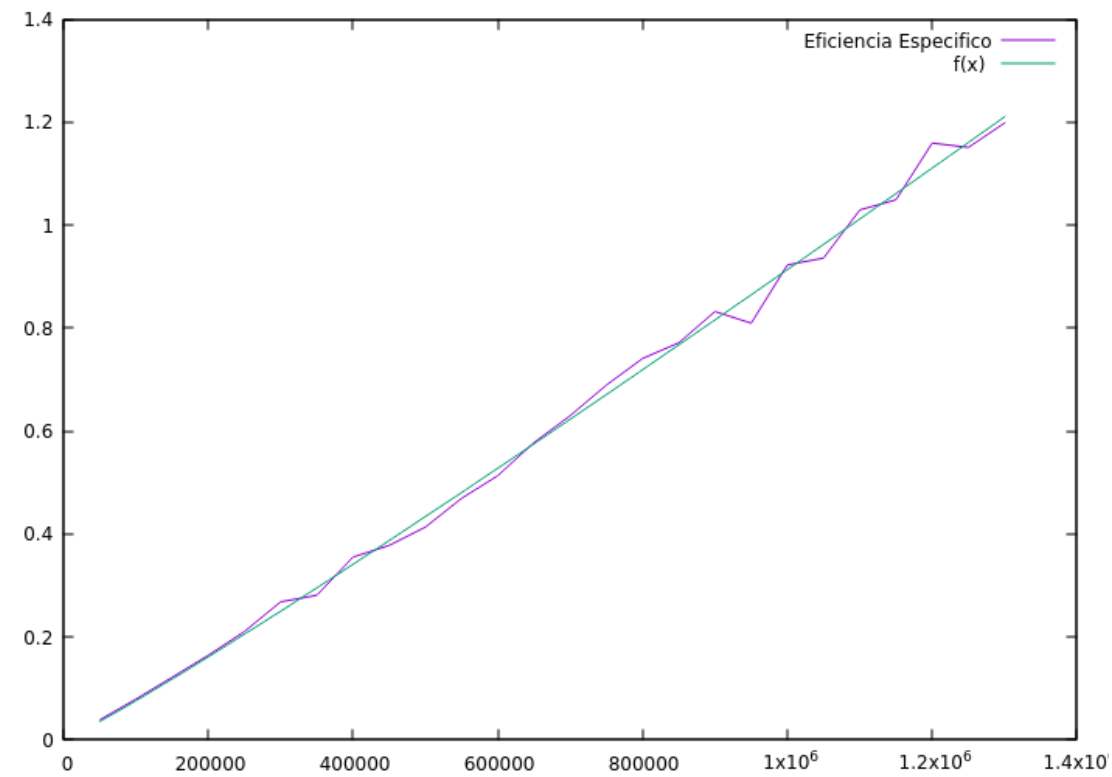
## Constantes ocultas

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

Cuyo coeficiente de correlación es de 0,9982

# Eficiencia Híbrida

---



# Algoritmo Divide y Vencerás

---

# Algoritmo Divide y Vencerás

---

Para aplicar el Divide y Vencerás, comenzamos ordenando el vector de forma creciente en función del punto con menor coordenada en el eje X. Para la ordenación utilizamos el método quicksort.

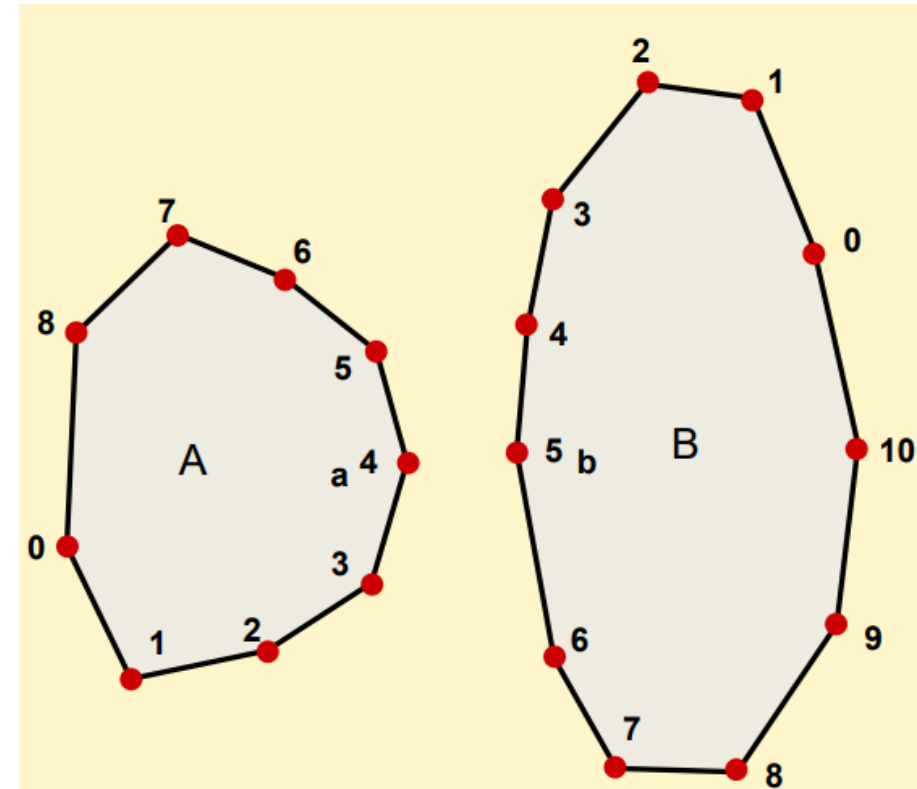
```
int comparePuntos (const void * a, const void * b) {  
    Punto * p = (Punto *) a;  
    Punto * q = (Punto *) b;  
  
    int retorno = 0;  
  
    if (p->getX() < q->getX()){  
        retorno = -1;  
    } else if (p -> getX() > q->getX()) {  
        retorno = 1;  
    }  
    return (retorno);  
}
```

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

---

Una vez ordenado, dividimos el vector en partes iguales hasta que el número de puntos sea  $< 100$ , 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.





---

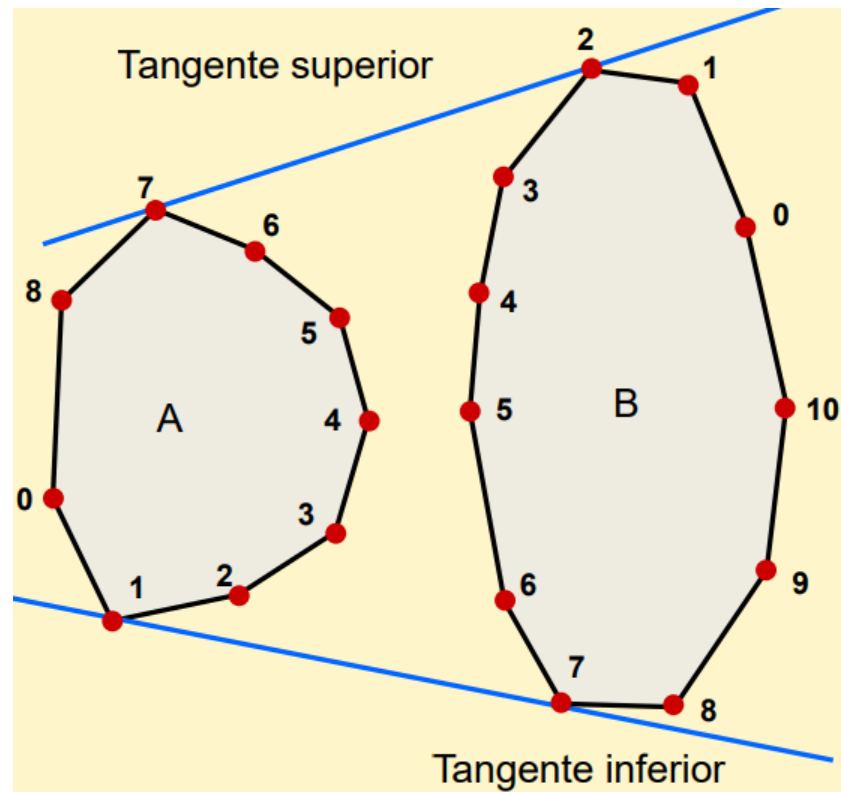
```
vector<Punto> DivideyVenceras_lims (vector<Punto> p, int inicial, int final){
    vector<Punto> solucion;
    if (final - inicial <= UMBRAL){
        solucion = EnvolventeConexa_lims(p, inicial, final);
    } else {
        int k = (final - inicial)/2;

        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));
    }

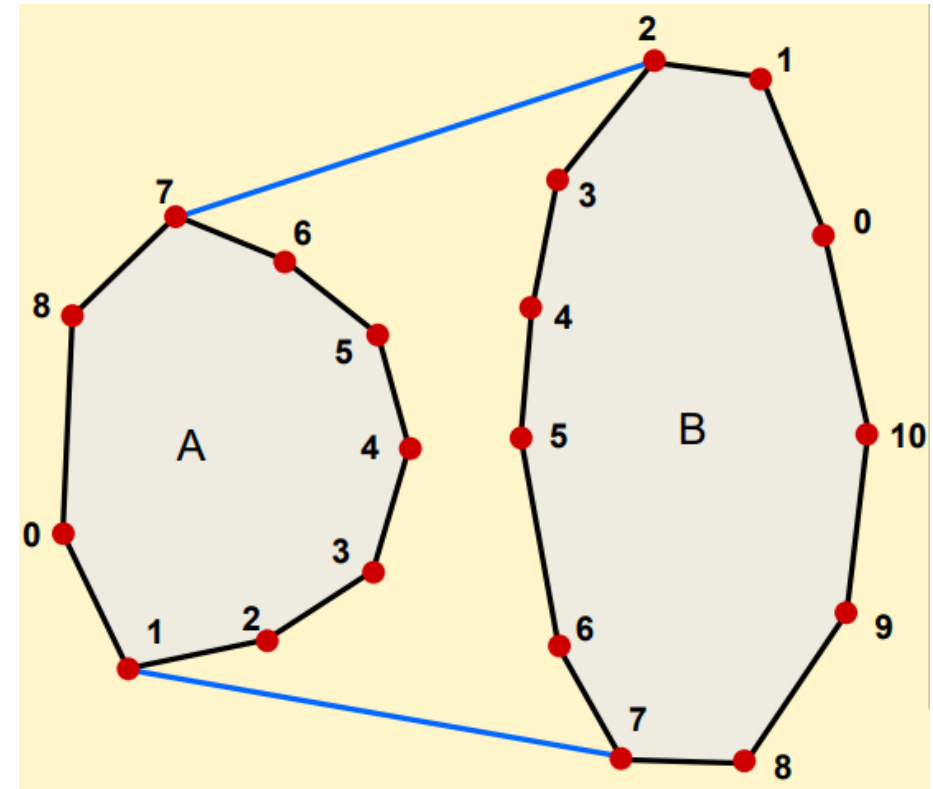
    return (solucion);
}
```



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.



# Eficiencia Teórica

---

La eficiencia teórica de nuestro algoritmo divide y vencerás vendrá dada por:  $T(n) = \begin{cases} t(n) & \text{si } n < 100 \\ 2T(n/2) + F(n) & \text{si } n \geq 100 \end{cases}$  Donde  $t(n)$  es la eficiencia del caso base, es decir la eficiencia teórica del algoritmo de Graham, que es  $O(n \log_2(n))$ , como hemos mencionado previamente, mientras que  $F(n)$  es la eficiencia de la función de fusión, cuyo tiempo de ejecución será de  $F(n) = 4n \in O(n)$  Para hallar la eficiencia resolveremos la recursividad:  $T(n) = 2T(n/2) + n$  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  $O(n \log_2(n))$  ya que  $I = 2 = 2^1 = b$  

# Eficiencia Empírica

---

```
#!/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
```

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:

# Eficiencia Empírica

---

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:

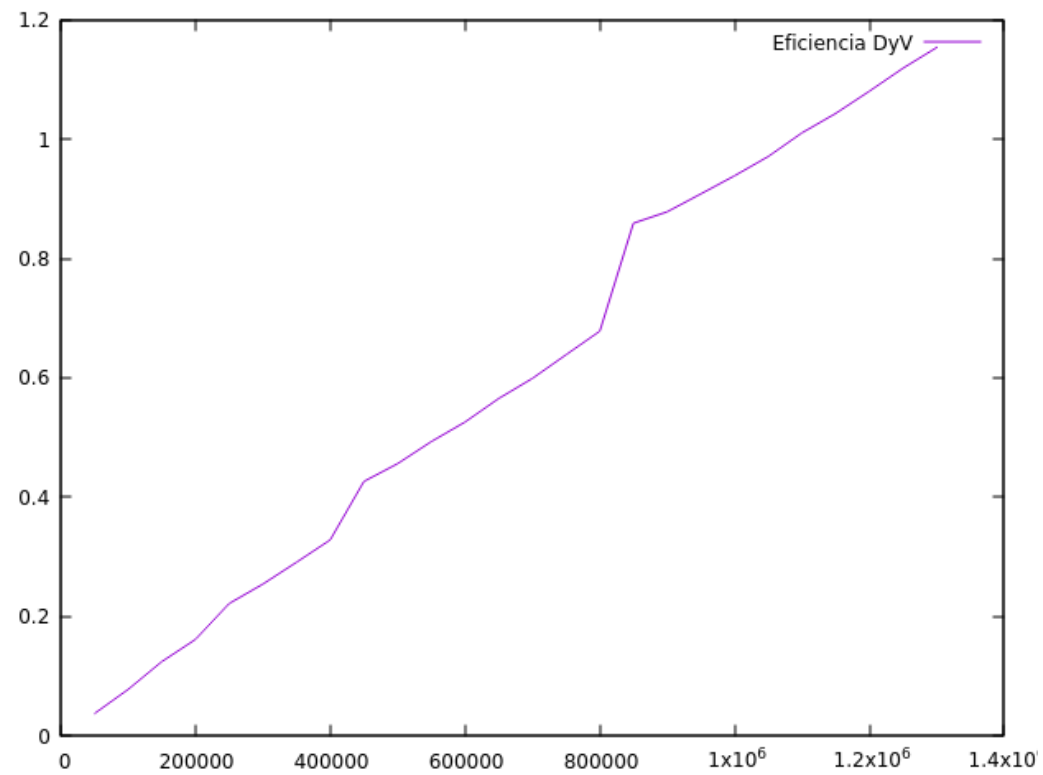
# Eficiencia Empírica

---

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

# Eficiencia Empírica

---





# Eficiencia Híbrida

---

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

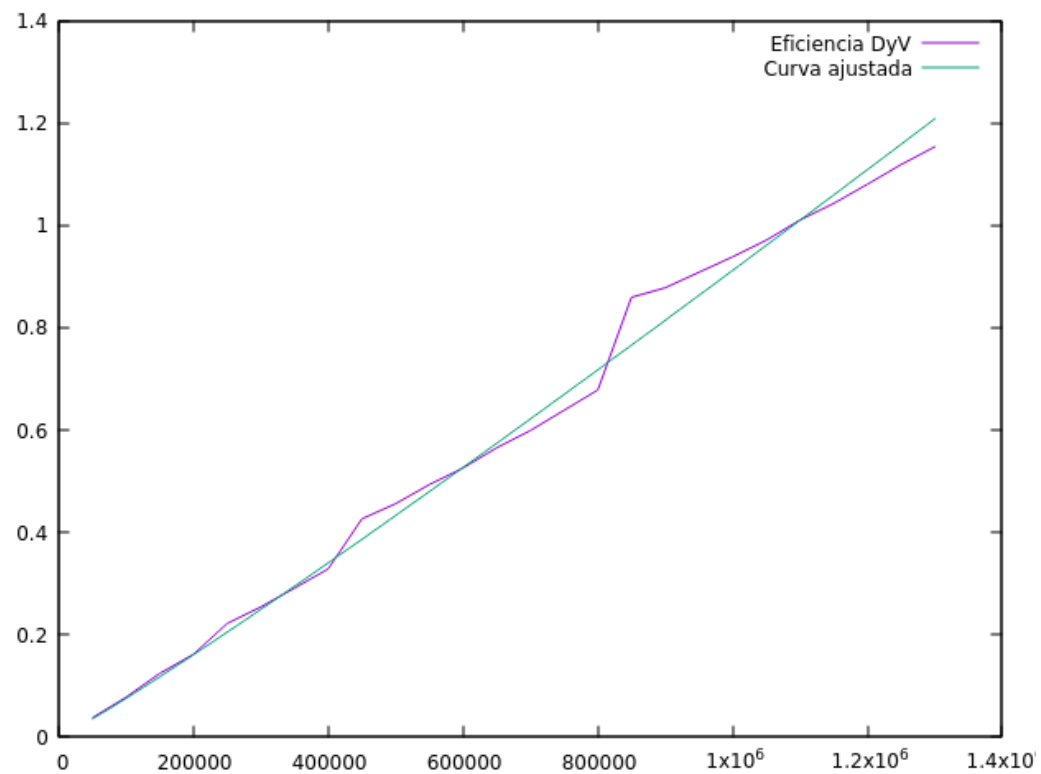
Constantes ocultas

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

Cuyo coeficiente de correlación es de 0.9962.

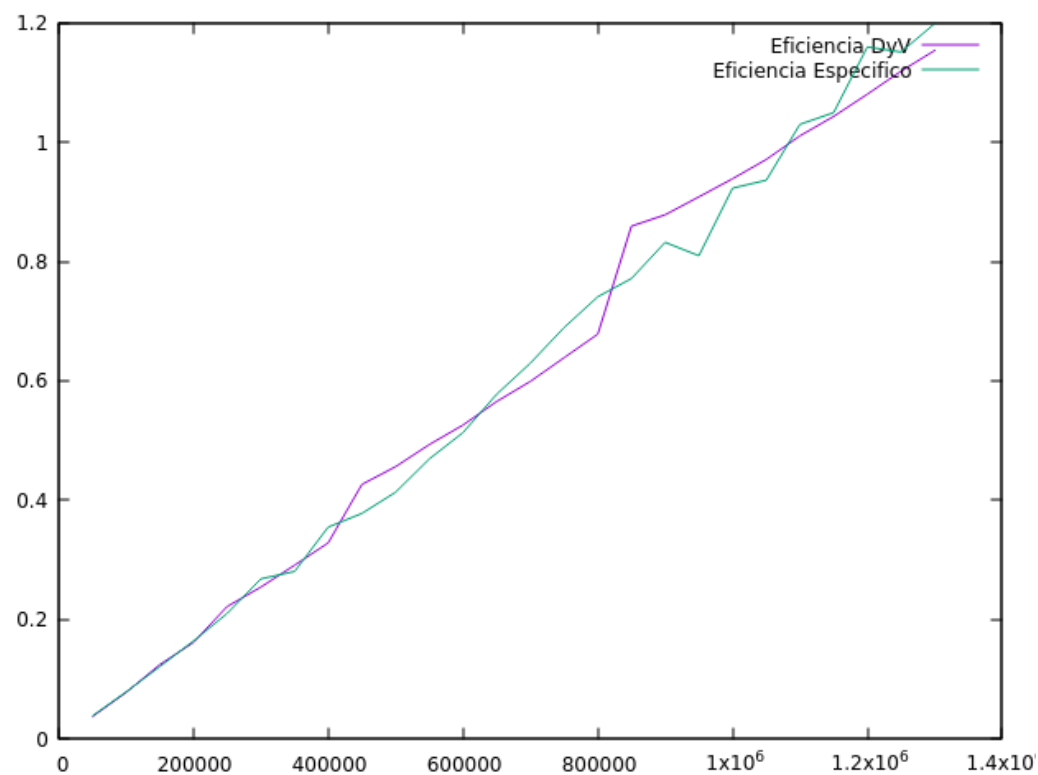
# Eficiencia Híbrida

---



# Comparación Eficiencias

---



# Umbral teórico

---

Calcularemos el umbral teórico usando un solo nivel de recursividad:

$$t(n) = \begin{cases} h(n) = 2n \log_2(n) & \text{si } n < n_0 \\ 2t\left(\frac{n}{2}\right) + 4n & \text{si } n \geq n_0 \end{cases}$$

Calcularemos el umbral teórico usando un solo nivel de recursividad:

$$2n \log_2(n) = n \log\left(\frac{n}{2}\right) + 4n \quad \Leftrightarrow \quad [...] \quad \Leftrightarrow n = 2^3 = 8$$

Esto tiene sentido pues como ya hemos visto las gráficas son prácticamente coincidentes

# 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

Por lo que podemos tomar como umbral  $n = 1$

# 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.