




[FECHA]

# PRACTICA 1 AMC

3º GII UHU

MIGUEL Á. SÁNCHEZ DE LA ROSA  
BORJA LÓPEZ PINEDA



# Índice

## Análisis teórico

Análisis Búsqueda de triángulos exhaustiva

Análisis Búsqueda de triángulos DyV

Análisis Cálculo ARM Kruskal

Análisis Cálculo ARM Prim

## Coste empírico

Tiempos y graficas para búsqueda de triángulos

Tiempos y gráficas para cálculo del ARM

## Resultado de los conjuntos de datos

Resultados de buscar líneas

Resultados de buscar triángulos

Resultados de calcular el ARM

## Implementación de la aplicación

# Análisis teórico

Para el análisis de la complejidad temporal en algoritmos de búsqueda de triángulos se tomará la comparación de triángulos como instrucción crítica en lugar de contar operaciones elementales.

En los algoritmos para el cálculo del árbol de recubrimiento mínimo se realizará una estimación del conteo de operaciones elementales.

## Análisis Búsqueda de triángulos exhaustiva

```
Procedimiento Exhaustivo (puntos[p ... q] : Punto) : Triangulo
  Triangulo mejor = Triangulo(puntos[p], puntos[p+1], puntos[p+2])
  Para i = p Hasta q-2 Hacer
    Para j = i+1 Hasta q-1 Hacer
      Para k = j+1 Hasta q Hacer
        tmp = Triangulo(puntos[i], puntos[j], puntos[k])
        Si tmp < mejor Hacer
          mejor = tmp
        fSi
      fPara
    fPara
  fPara
  Devolver mejor
fProcedimiento
```

$$\begin{aligned} n &= q - p + 1 \\ T(n) &= \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^n 1 = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} (n-j) = \sum_{i=1}^{n-2} \frac{(1-n)(i-n+1)}{2} = \frac{n(n^2-3n+2)}{6} \\ &= n \frac{n-1}{2} \frac{n-2}{3} \in O(n^3) \end{aligned}$$

Este algoritmo acrece de casos, sin importar la distribución de los puntos el número de comprobaciones realizado es siempre el mismo.

## Análisis Búsqueda de triángulos DyV

**Procedimiento** *DyV* (*puntos*[*p* ... *q*] : Punto) : Triangulo

*puntosX* = **ordenarX**(*puntos*)

*puntosY* = **ordenarY**(*puntos*)

**Devolver** *DyV*(*puntosX*, *puntosY*)

**fProcedimiento**

**Procedimiento** *DyV* (*puntosX*[*p* ... *q*] : Punto, *puntosY*[*p* ... *q*]) : Triangulo

$n = q - p + 1$

**Si**  $n \leq 3$  **Haz**

**Devolver Exhaustivo**(*puntosX*)

**fSi**

$mid = (q + p) / 2$

**Para**  $i = p$  **Hasta**  $q$  **Hacer**

**Si** *puntosY*[*i*].*x* < *puntosX*[*mid*].*x* **Haz**

*puntosYizq*[] = *puntosY*[*i*]

**SiNo**

*puntosYder*[] = *puntosY*[*i*]

**fSi**

**fPara**

*minIzq* = **DyV**(*puntosX*.sub(*p*, *mid*), *puntosYizq*)

*minDer* = **DyV**(*puntosX*.sub(*mid*+1, *q*), *puntosYder*)

*min* = **min**(*minIzq*, *minDer*)

**Para**  $i = p$  **Hasta**  $q$  **Haz**

**Si** **abs**(*puntosY*[*i*].*x* - *puntosX*[*mid*].*x*) < *min*.perimetro / 2 **Haz**

*crossover*[] = *puntosY*[*i*]

**fSi**

**fPara**

**Para**  $i = 1$  **Hasta** *crossover*.n-2 **Hacer**

**Para**  $j = i+1$  **Hasta** *crossover*.n-2 **Y** **abs**(*crossover*[*i*].*x* - *puntosY*[*j*].*x*) < *min*.perimetro / 2 **Hacer**

**Para**  $k = j+1$  **Hasta** *crossover*.n-2 **Y** **abs**(*crossover*[*i*].*x* - *puntosY*[*k*].*x*) < *min*.perimetro / 2 **Hacer**

*tmp* = **Triangulo**(*puntos*[*i*], *puntos*[*j*], *puntos*[*k*])

**Si** *tmp* < *min* **Hacer**

*min* = *min*

**fSi**

**fPara**

**fPara**

**fPara**

**Devolver** *min*

**fProcedimiento**

Para obtener la complejidad temporal tenemos en cuenta solo la instrucción crítica y suponemos que la búsqueda en zona crossover tiene un coste lineal ya que la cantidad de triángulos que forma partiendo de un primer punto son muy escasas y solo se comprueban unos escasos puntos. En la práctica esta búsqueda depende más de la organización de los puntos, que de la cantidad. La constante  $K_0$  representa la búsqueda crossover y 1 es la comparación necesaria para decidir cual es el mejor triangulo de entre los dos mejores encontrados en cada lado.

Sin tener en cuenta el tiempo de ordenación, solo el de búsqueda:

$$T(n) = 2T\left(\frac{n}{2}\right) + nk_0 + k_1 \text{ si } n > 3; T(3) = 3\frac{2}{3} = 1$$

$$T(n) - 2T\left(\frac{n}{2}\right) = nk_0 + 1$$

*Cambio de variable:  $n = 2^m$*

$$T(2^m) - 2T\left(\frac{2^m}{2}\right) = 2^m k_0 + 1$$

$$T(2^m) - 2T(2^{m-1}) = 2^m k_0 + 1$$

$$T_m - 2T_{m-1} = 2^m k_0 + 1$$

$2^m k_0 + 1$  es de la forma  $2^m + p(m)$  donde  $p(m)$  es un polinomio de grado 0

*Encuación característica para recursividad no homogénea:  $(x - 2)(x - 2) = 0$*

$$r_0 = 2; m_0 = 2$$

$$T_m = 2^m c_1 + 2^m m c_2$$

*Deshaciendo el cambio de variable:  $T(n) = 2^{\log(n)} c_1 + 2^{\log(n)} \log(n) c_2$*

$$T(n) = n c_1 + n \log(n) c_2$$

*Para hayas las constantes, calculamos  $T(6)$*

$$T(6) = 2T(3) + 6k_0 + 1 = 3 + 6k_0$$

$$3c_1 + 3\log(3)c_2 = 1$$

$$6c_1 + 6\log(6)c_2 = 3 + 6k_0$$

$$a = \frac{\log\left(\frac{4}{3}\right) - k_0 \log(729)}{\log(64)}, b = k_0 + \frac{1}{6}$$

*Sabiendo que la constante  $k_0$  es positiva,  $c_1$  y  $c_2$  también lo son por lo que*

$$T(n) \in O(n \log n)$$

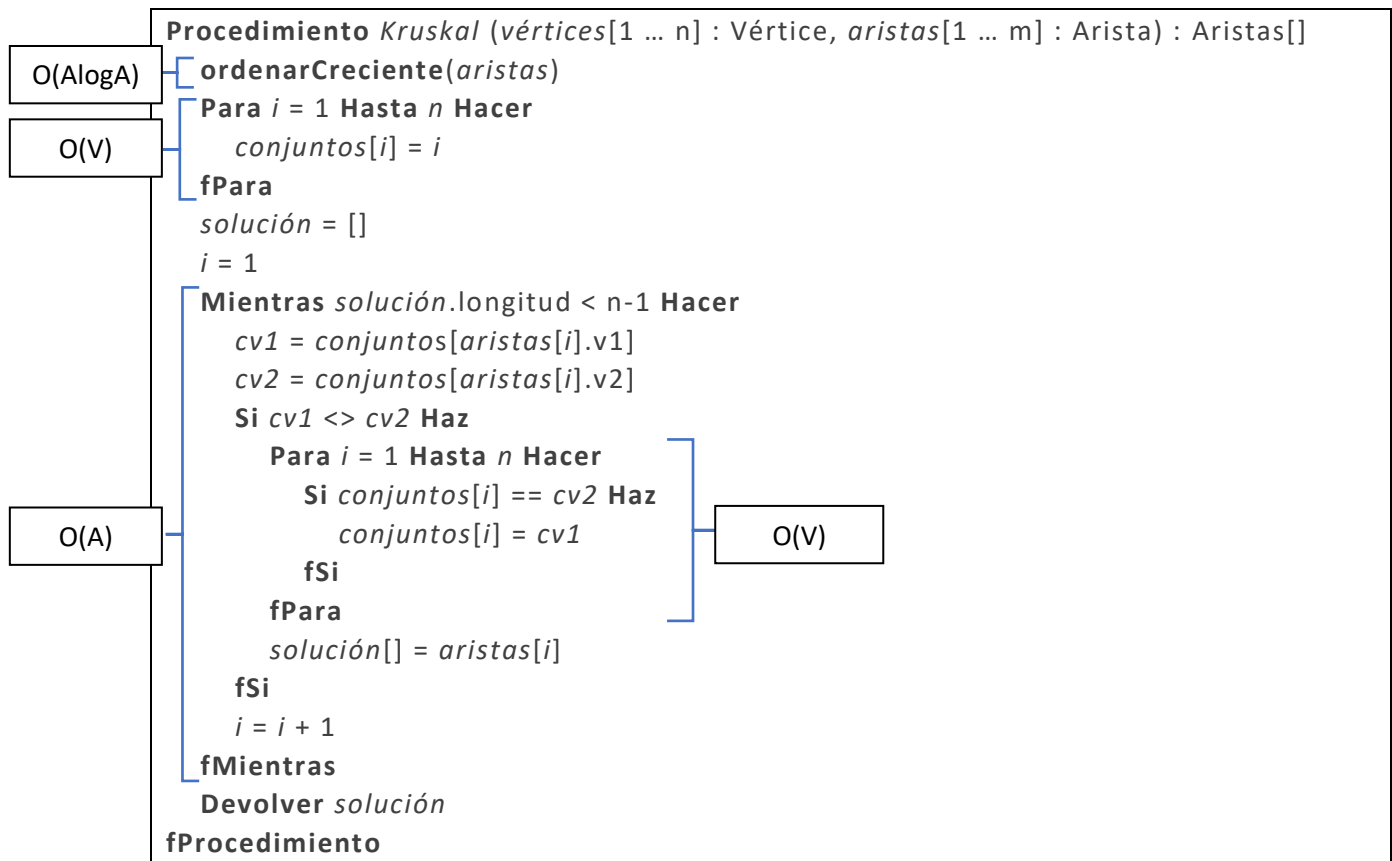
El coste de ordenar también es  $n \log n$  si se realiza por un método rápido, el coste total de algoritmo sigue siendo  $n \log n$ .

También es posible no preordenar el vector en el eje  $y$ , sino realizarlo en cada llamada. Esto aumentaría el coste del algoritmo a  $O(n \log^2 n)$  aunque consumiría menos memoria. En las pruebas prácticas se ha demostrado que en ordenadores modernos el factor limitante es la memoria stack y no la heap, por lo que preordenar el array resulta siempre beneficioso.

El caso mejor en este algoritmo se dará cuando los puntos se distribuyan de tal forma que al dividir en mitades nunca haya más de dos puntos en el centro, haciendo 0 el número de comprobaciones en la zona crossover.

El caso peor se dará cuando los puntos estén todos distribuidos en una región muy pequeña del eje  $x$ . En ese caso la búsqueda crossover se acercaría a una búsqueda exhaustiva.

## Análisis Cálculo ARM Kruskal



A es la cantidad de aristas y V la cantidad de vértices. Siempre se cumple que  $A \leq V^2$

La ordenación puede llegar a tardar  $A \log A$  si se realiza por un algoritmo rápido.

Iniciar los conjuntos siempre tarda V.

La búsqueda requiere V búsquedas y uniones de conjuntos. Las búsquedas son  $O(1)$  y la unión puede llegar a ser  $O(\log V)$ , aunque en esta implementación se utiliza un método más sencillo que requiere  $O(V)^1$ . Esta reducción de complejidad se consigue con una estructura de datos de tipo subconjunto que almacena un identificador de conjunto y un puntero padre por cada elemento.

El orden máximo (asumiendo que se implementan los conjuntos de la forma más eficiente posible) es  $A \log A$  ya que  $A \leq V^2$  podemos escribirlo como  $A \log V^2$  y por propiedades de logaritmos  $A 2 \log V$  que está en el orden  $A \log V$ .

Kruskal se implementa más fácilmente un grafo representado por conjuntos de vértices y aristas. Su complejidad mínima es  $O(A \log V)$  por que resulta mejor para grafos dispersos en los que hay un bajo número de aristas.

Existen diferenciados casos. El caso mejor se dará cuando las aristas que permitan crear el árbol de recubrimiento mínimo tengan menor coste que el resto. El caso peor se dará si ocurre lo contrario, la mayoría de las aristas que conectan vértices nuevos tienen el mayor coste.

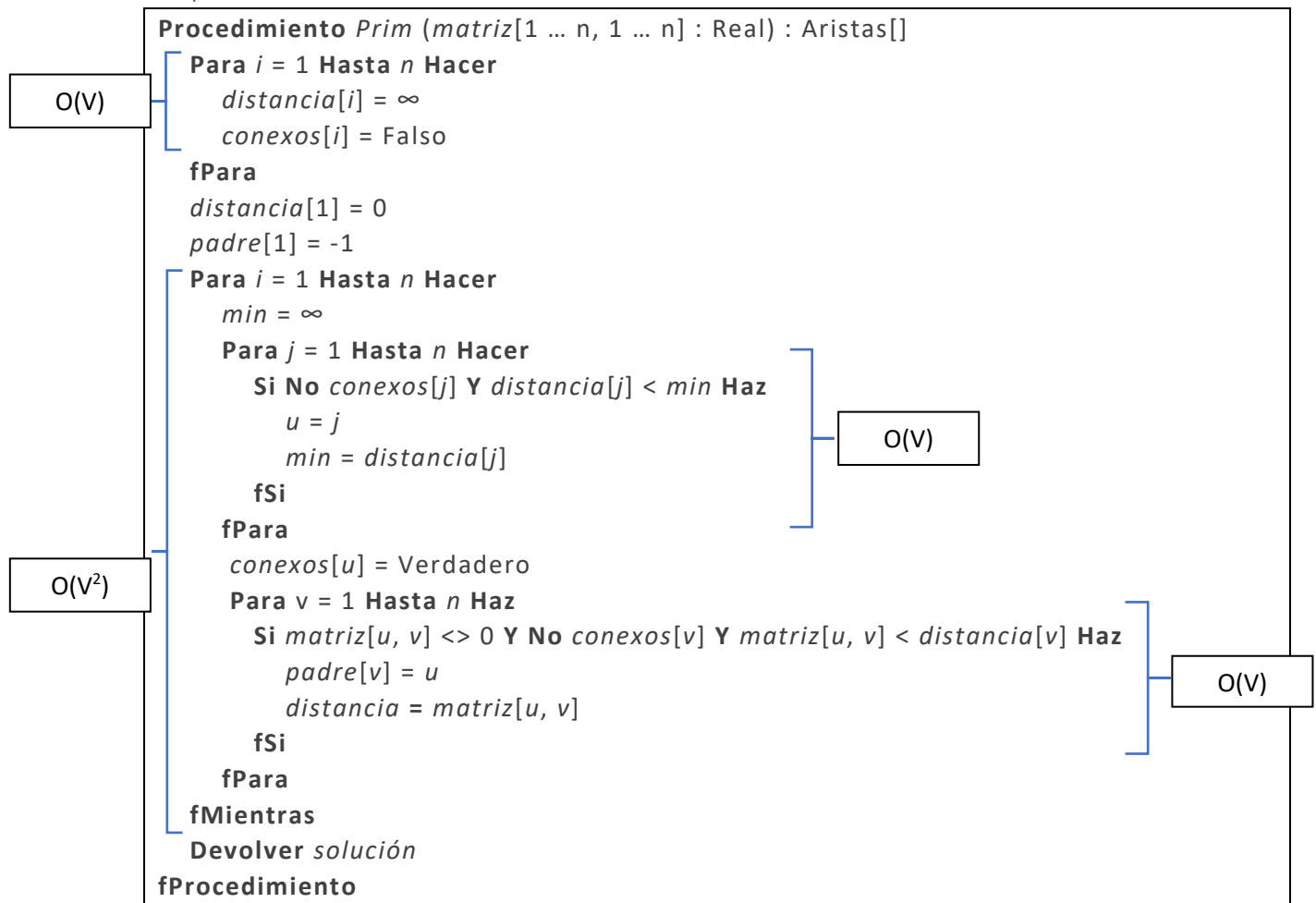
En todos casos, la complejidad temporal no varía ya que se sigue teniendo que ejecutar la ordenación que depende del número de aristas. Si el grafo es totalmente disperso ( $A = V$ ) la complejidad del algoritmo será de  $O(V \log V)$

<sup>1</sup> Ejemplo de implementación de conjuntos disjuntos con búsqueda y unión  $O(\log n)$

<https://www.geeksforgeeks.org/disjoint-set-data-structures/>

## Análisis Cálculo ARM Prim

### Implementación más eficiente



La complejidad temporal del algoritmo de Prim es de  $O(V^2)$ , aunque si se emplean montículos de Fibonacci la complejidad puede bajar a  $O(A + \log V)^2$

Prim resulta más sencillo de implementar representado el grafo con matrices de adyacencia. A pesar de esto, no funcionará con grafos dirigidos, al igual que Kruskal.

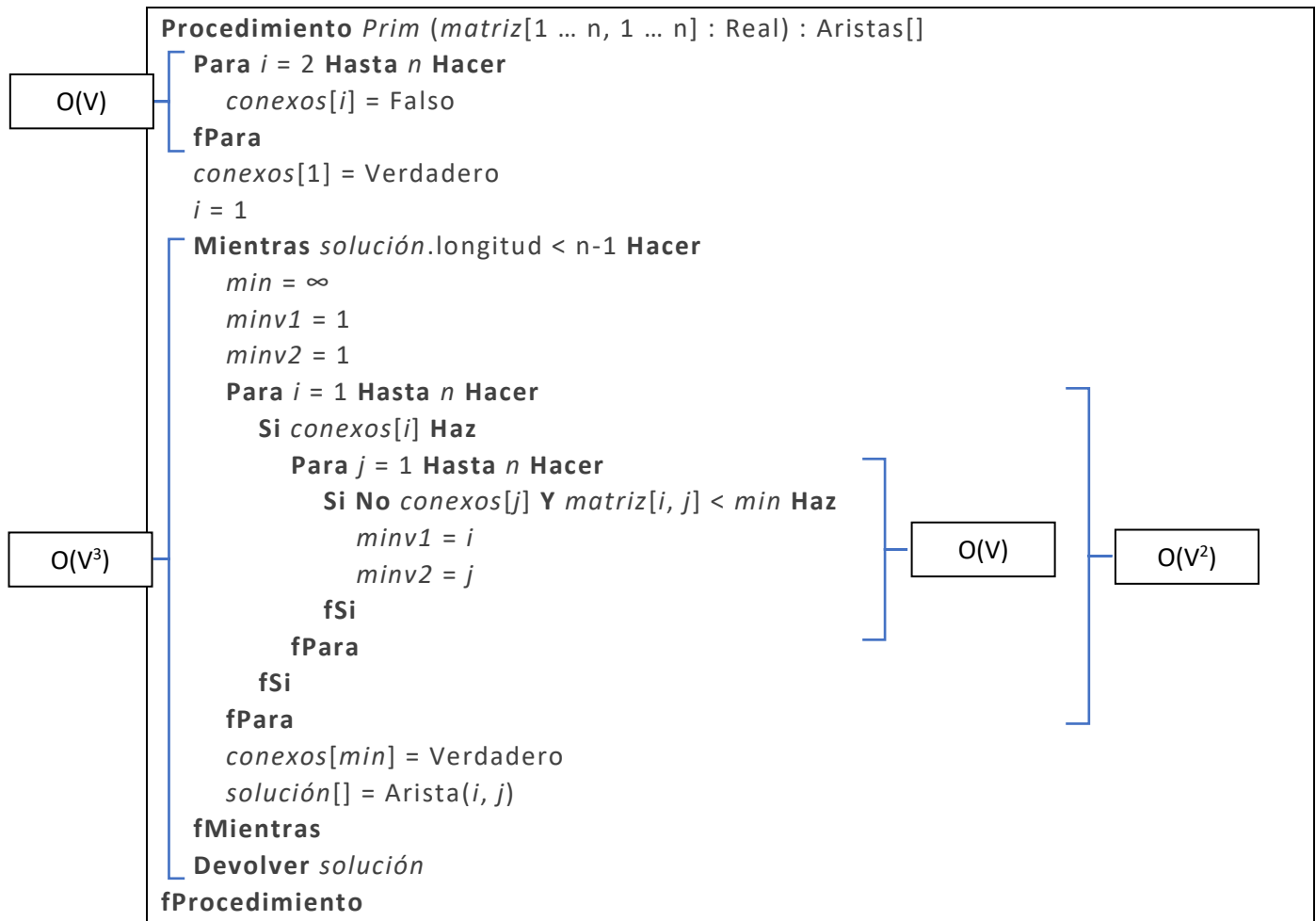
En comparación con Kruskal, el algoritmo de Prim resulta más eficiente en grafos densos donde hay una gran cantidad de aristas.

En Prim no se identifican casos si la implementación se realiza mediante matrices de adyacencia, su complejidad es la misma independientemente del número de aristas.

<sup>2</sup> Implementation del algorithm de Prim con montículos de Fibonacci

<https://keithschwarz.com/interesting/code/?dir=prim>

### Implementación sencilla



Posible implementación  $O(V^3)$  del algoritmo de Prim.

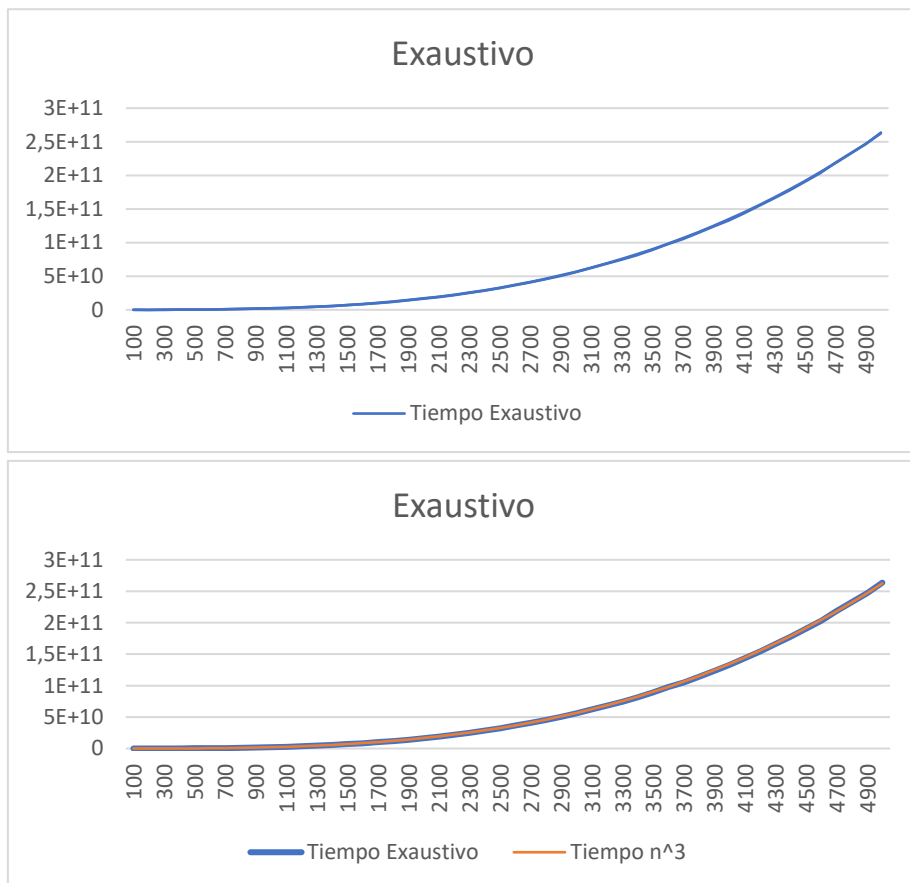


# Resultados empíricos

Todas las gráficas han sido generadas repitiendo cada taya 10 y tomando el tiempo medio en nanosegundos. Pese a esto, el tiempo medido en tayas grandes resulta notoriamente menos preciso que el medido en tayas pequeñas por la cantidad de procesos que pueden interferir mientras se ejecuta el algoritmo.

## Tiempos y graficas para búsqueda de triángulos

### Algoritmo exhaustivo

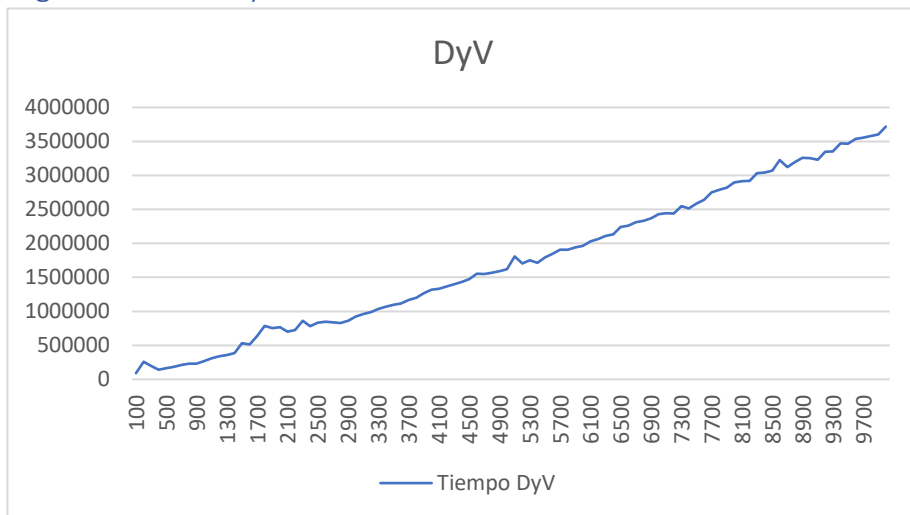


Algoritmo exhaustivo ejecutado en tallas desde 100 a 5000 con incrementos de 100.

En tallas bajas, menos de 1000 puntos, el algoritmo tarda menos de 2 segundos. Pero cuando la talla aumenta crece como cabría esperar de una complejidad  $O(n^3)$ . Con 5.000 elementos tarda más 4 minutos.

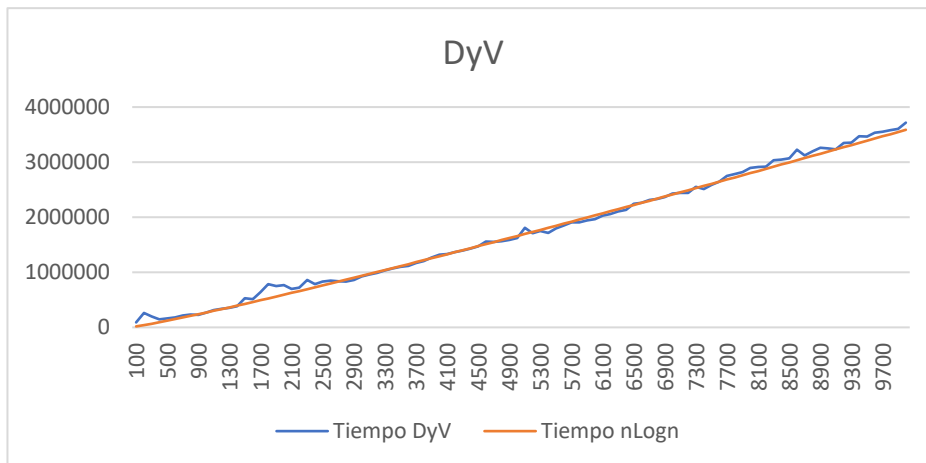
El ajuste es perfecto cuando se superpone con una función cúbica.

### Algoritmo Divide y vencerás



Algoritmo basado en la estrategia Divide y vencerás en tallas desde 100 a 10.000.

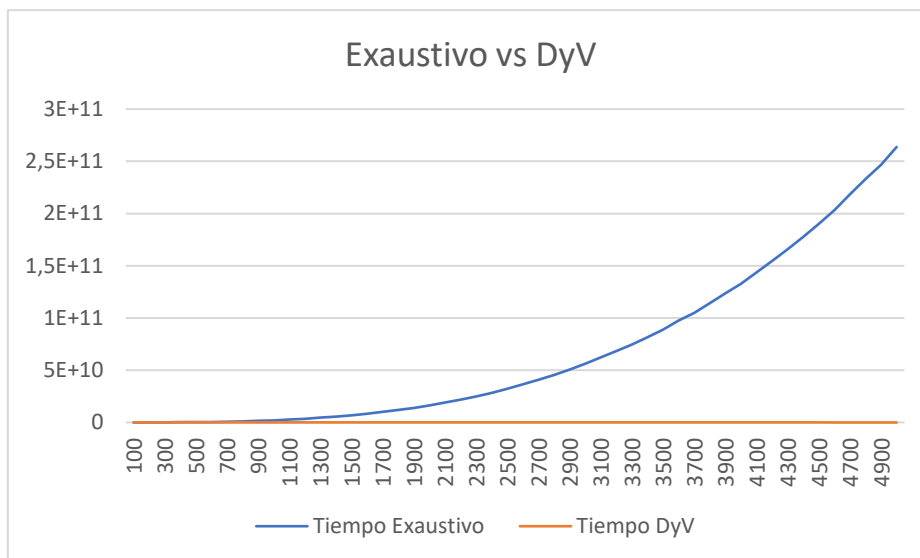
Incluso en la taya más grande de este gráfico el tiempo no supera las 4 milésimas de segundo.



Aunque pudiera parecer lineal, la complejidad temporal es  $O(n \log n)$  por el estudio teórico y se demuestra con el buen ajuste que se realiza con una función  $n \log n$ .

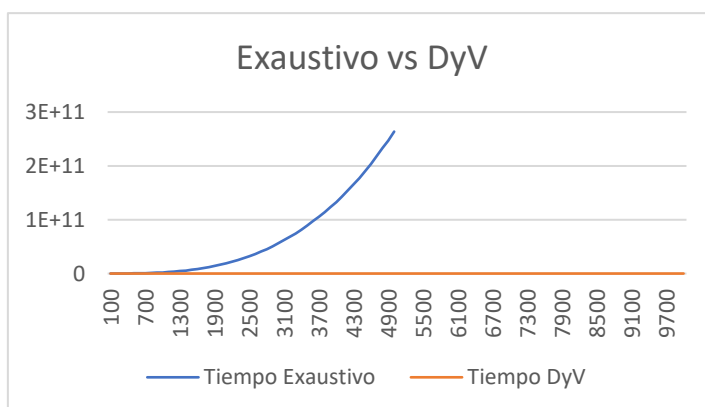
La ejecución de cada tamaño se repitió 100 veces en lugar de 10, por lo poco constante que resultaba el tiempo de ejecución. Los datos se generan aleatoriamente y eso hace que DyV tarde más o menos en función de como se agrupen los puntos, mientras que en el algoritmo exhaustivo se realiza siempre el mismo número de operaciones para una taya dada.

### Comparación Exhaustivo con Divide y Vencerás



En comparativa, DyV parece prácticamente constante es prácticamente imposible graficar correctamente estos dos algoritmos juntos.

Con 5.000 puntos Exhaustivo tarda 4 minutos, mientras que DyV tarda 4 millonésimas de segundo.



En esta última comparativa el algoritmo exhaustivo no se ejecutó hasta el final porque tardaría demasiado.

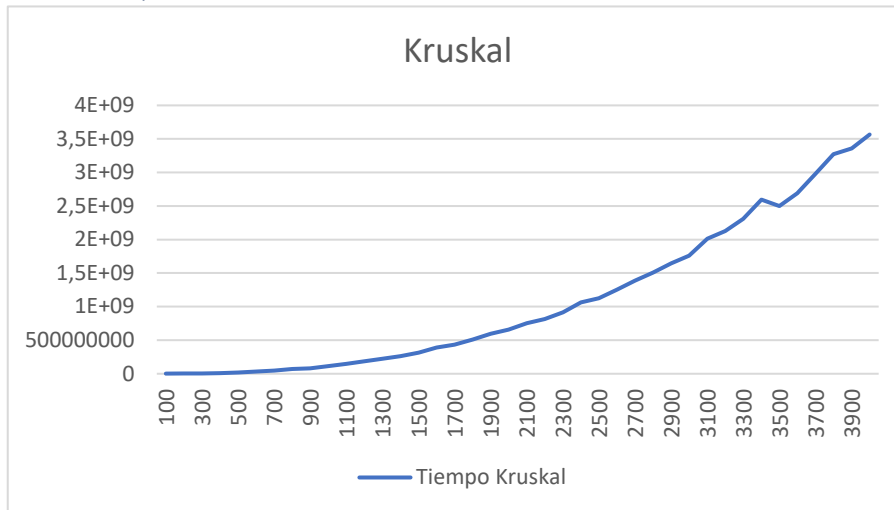
Hemos encontrado que la talla con la que el algoritmo DyV tarda entre 2 y 3 minutos es de 50 millones de puntos. No hemos completado la ejecución del algoritmo exhaustivo en esta taya porque, según la función que mejor se ajusta a la duración de este algoritmo, tardaría 8 millones de años en terminar. Esa es la mitad de la edad del universo.

## Tiempos y gráficas para cálculo del ARM

Todas las gráficas han sido generadas repitiendo 10 veces la ejecución del algoritmo en cada taya y tomando la media.

### Algoritmo de Kruskal

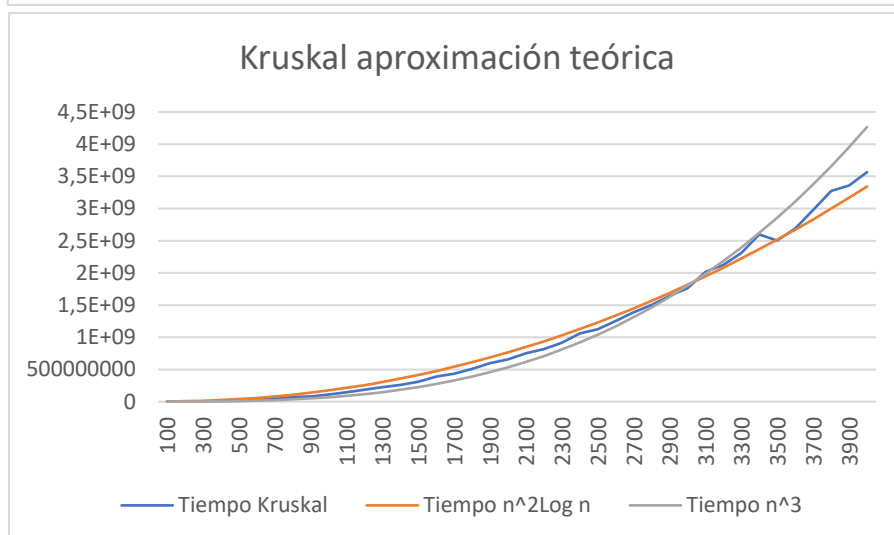
Kruskal implementado de la forma más eficiente



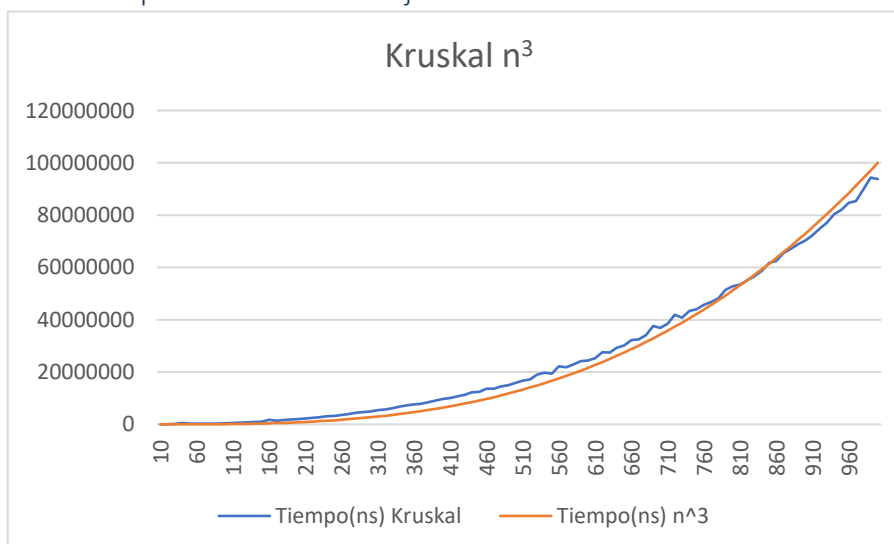
Kruskal en tayas desde 100 a 4000 en incrementos de 100.

Si intentamos aproximar los datos obtenidos con una función  $n^2 \log n$  la curva se asemeja mucho más a los datos obtenidos que si usamos una  $n^3$ .

Esto se explica por la utilización de una implementación de conjuntos disjuntos cuyo tiempo de búsqueda y unión es  $O(\log n)$



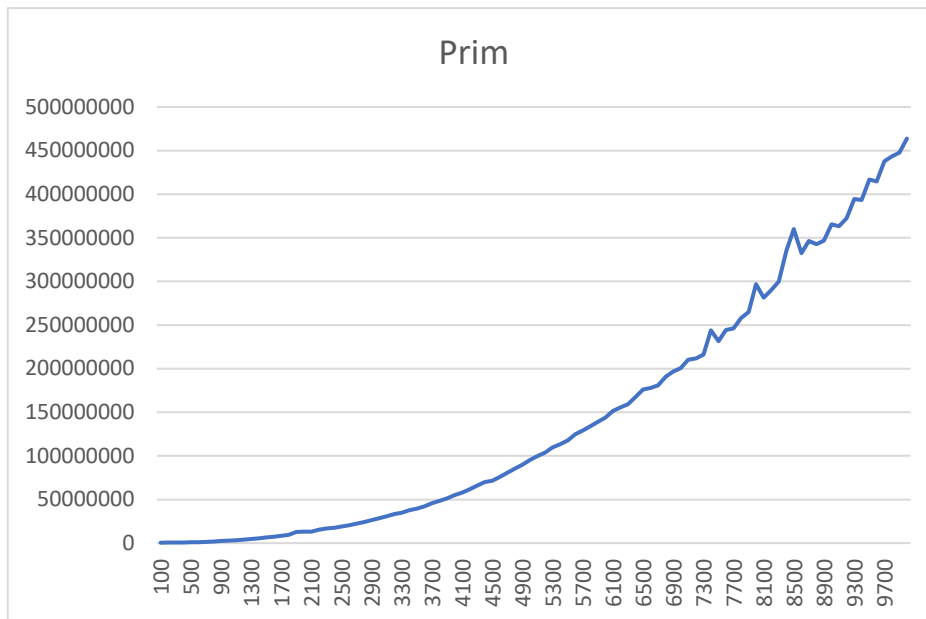
Kruskal implementado con conjuntos sencillos



Kruskal en tayas desde 10 a 1000 con incrementos de 10.

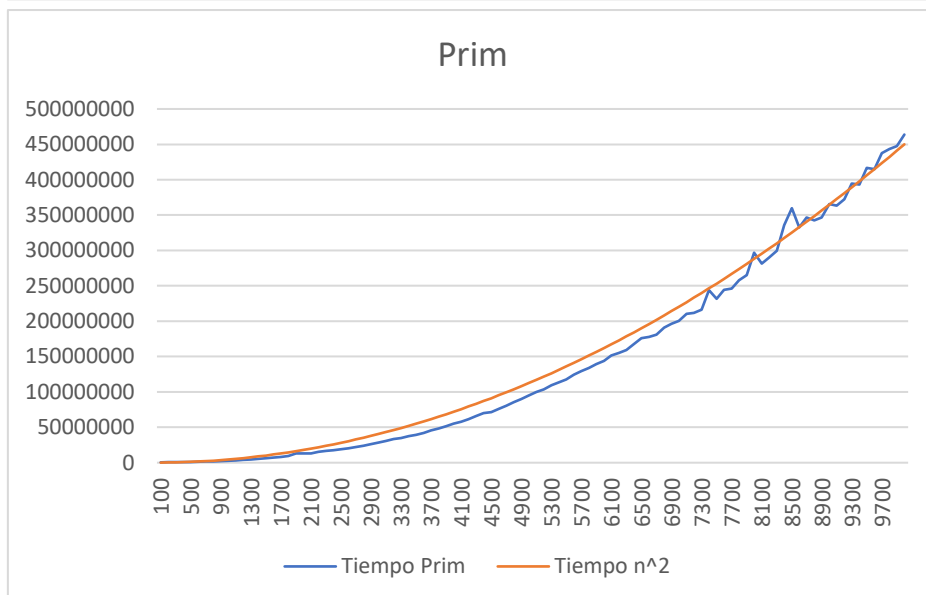
Una implementación más sencilla de los conjuntos disjuntos consigue reducir a  $O(1)$  el tiempo de búsqueda, pero aumenta a  $O(n)$  el tiempo de unión. Como resultado el algoritmo tiene complejidad  $n^3$

## Prim implementado de la forma más eficiente

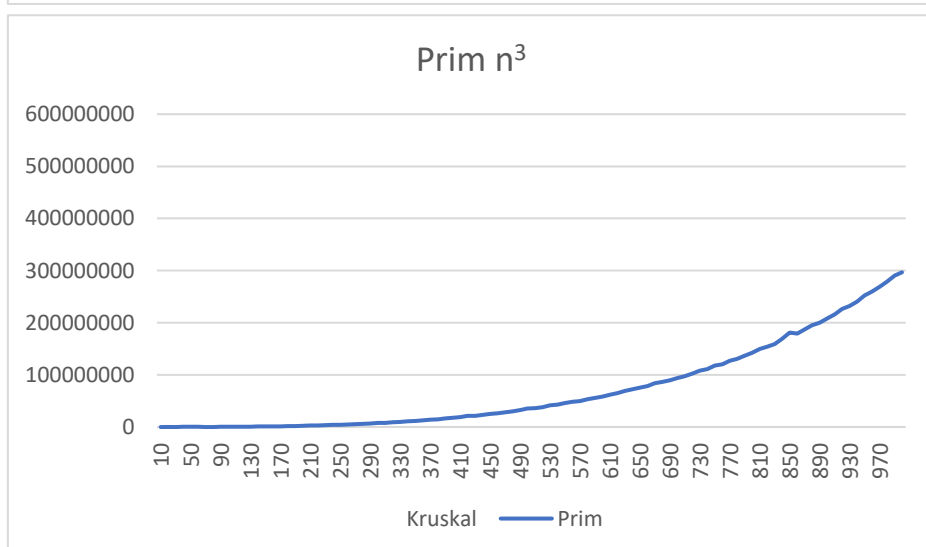


Prim en tayas desde 100 a 10000 en incrementos de 100.

El orden mínimo de Prim, sin utilizar montículos de Fibonacci, es  $O(n^2)$



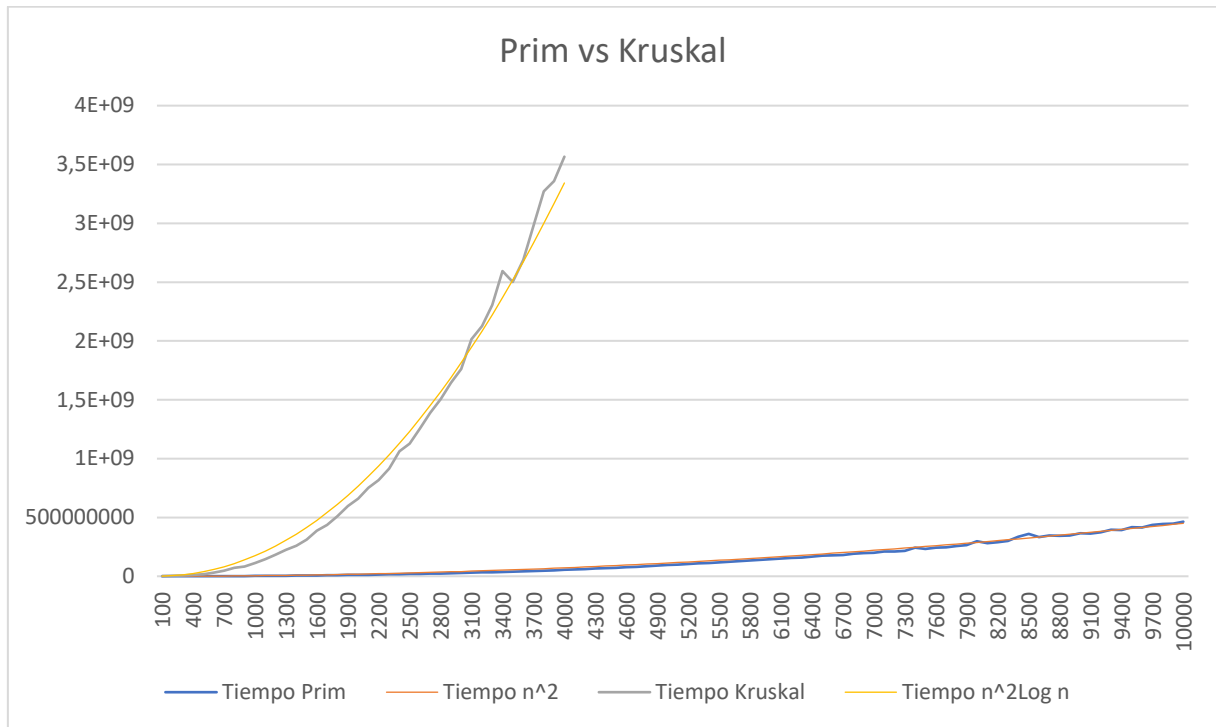
En esta gráfica puede verse la relación entre el modelo teórico y los datos empíricos



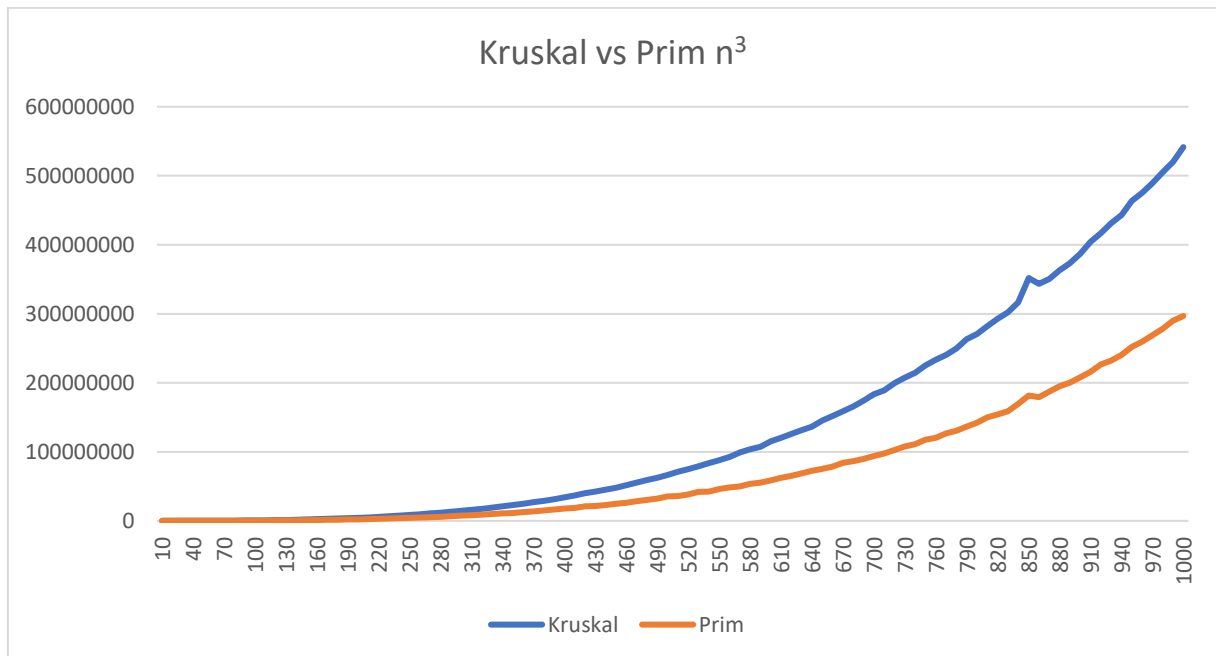
Prim puede ser implementado de manera más directa, sin utilizar otras estructuras de datos más que la matriz de adyacencia, pero su complejidad se vuelve  $O(n^3)$

Este gráfico será usado más adelante.

## Comparativa Prim contra Kruskal



En comparativa, Kruskal muestra un orden de complejidad mayor que el de Prim complejidad bastante mayor, incluso se aprecia que las constantes de Prim son menores.



Para esta prueba se implementó Kruskal y Prim en sus versiones  $O(n^3)$ , teniendo la misma complejidad temporal, Prim demuestra unas constantes menores.

# Resultado de los conjuntos de datos

## Resultados de buscar líneas

- **berlin52: 35** (685.0, 595.0) – **36** (685.0, 610.0)
- **ch130: 12** (252.7493090080, 535.7430385019) – **87** (252.4286967767, 535.1659364856)
- **ch150: 49** (334.3508329710 153.7969238040) – **147** (334.2748764383, 152.1494569394)
- **d493: 193** (3.36420e+03, 1.83460e+03) – **195** (3.35150e+03, 1.82190e+03)
- **d657: 77** (2.23400e+03, 1.50440e+03) – **84** (2.24670e+03, 1.51710e+03)

## Resultados de buscar triángulos

- **berlin52: 35** (685.0, 595.0) – **36** (685.0, 610.0) – **34** (700.0, 580.0)
- **ch130: 37** (561.4775136009, 357.3543930067) – **40** (571.7371050025, 375.7575350833) – **47** (572.7630641427, 373.3208821255)
- **ch150: 49** (334.3508329710, 153.7969238040) – **144** (352.3140807211, 140.3273323662) – **147** (334.2748764383, 152.1494569394)
- **d493: 142** (2.92610e+03, 2.22830e+03) – **140** (2.90700e+03, 2.23470e+03) – **138** (2.91340e+03, 2.25370e+03)
- **d657: 36** (1.22430e+03, 1.25040e+03) – **37** (1.21160e+03, 1.26310e+03) – **40** (1.22430e+03, 1.27580e+03)

## Resultados de calcular el ARM

- **berlin52:** 6081.630541640885
- **ch130:** 5164.052799961011
- **ch150:** 5880.955830859872
- **d493:** 29284.55098300803
- **d657:** 42490.60615581593

# Implementación de la aplicación