



UNIVERSIDAD  
DE GRANADA

Universidad de Granada

FACULTAD DE INGENIERÍA INFORMÁTICA Y  
TELECOMUNICACIONES

## PRÁCTICA 2: DIVIDE Y VENCERÁS

*Doble Grado Ingeniería Informática y Matemáticas*

Autores:

Adolfo Martínez Olmedo, Pablo Delgado Galera, Marcos Baena  
Solar

Marzo 2025

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>2</b>
<b>3. El número más pequeño</b>	<b>3</b>
3.1. Análisis de Fuerza Bruta . . . . .	3
3.2. Análisis de Divide y Vencerás . . . . .	3
3.3. Tablas y Gráficos de Rendimiento . . . . .	4
3.4. Análisis de los Resultados . . . . .	4
<b>4. El par de puntos más cercano</b>	<b>5</b>
4.1. Análisis de Fuerza Bruta . . . . .	5
4.2. Análisis de Divide y Vencerás . . . . .	5
4.3. Tablas y Gráficos de Rendimiento . . . . .	7
4.4. Análisis de los Resultados . . . . .	7
<b>5. La envolvente convexa</b>	<b>8</b>
5.1. Análisis de Fuerza Bruta . . . . .	8
5.2. Análisis de Divide y Vencerás . . . . .	9
5.3. Tablas y Gráficos de Rendimiento . . . . .	10
5.4. Análisis de los Resultados . . . . .	10
<b>6. Conclusiones</b>	<b>10</b>

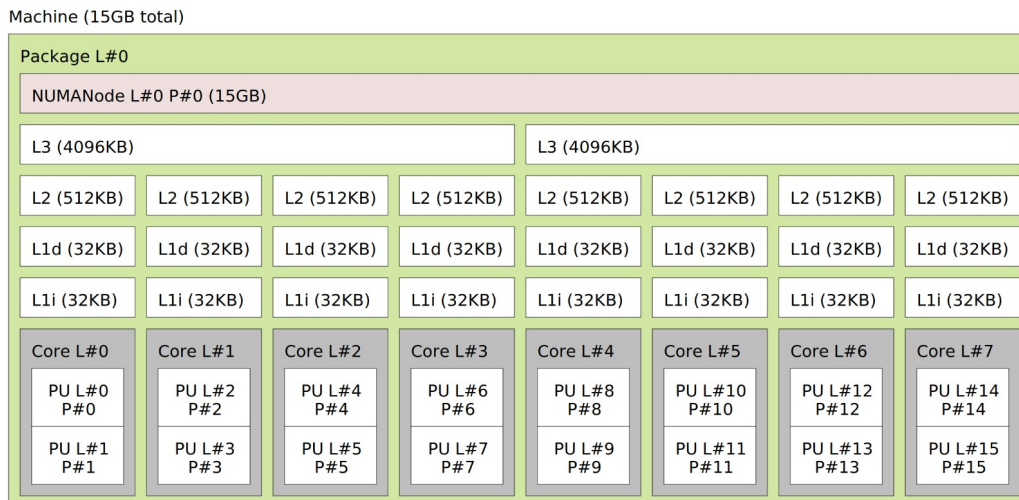
# 1. Introducción

En esta práctica se desarrolla la resolución de tres problemas diferentes:

- Menor número
- Menor distancia
- Menor envolvente convexa

Lo haremos de dos maneras distintas, primero haremos una aproximación mediante algoritmos de fuerza bruta y después aplicaremos lo aprendido en clase, desarrollando soluciones basadas en divide y vencerás. Analizaremos la eficiencia teórica de cada solución, mediante un script ejecutaremos los programas con diferentes tamaños y una cantidad determinada de veces, nos quedaremos con una media de todas las ejecuciones y podremos comparar la eficiencia de ambas soluciones, viendo a partir de que tamaños es mejor una solución a la otra en cuanto a eficiencia.

Para todas las ejecuciones de los diferentes algoritmos hemos usado el mismo portátil. Un *Asus Zephyrus G14* con un procesador *Ryzen 7 4800HS*, cuyas especificaciones se pueden ver en la siguiente imagen:



# 2. Objetivos

Los objetivos de esta práctica son los siguientes:

- Comprender la técnica de Divide y Vencerás y sus ventajas.
- Comparar con la estrategia de fuerza bruta y analizar la complejidad.
- Implementar ambos enfoques (fuerza bruta y Divide y Vencerás) para cada problema.
- Experimentar con el umbral de la técnica de Divide y Vencerás.

## 3. El número más pequeño

### 3.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en ir recorriendo el vector de puntos tantas veces como cifras tenga el número que busquemos, y en cada recorrido se busca el número más pequeño, se elimina y se procede a hacer lo mismo en la siguiente iteración, hasta obtener finalmente el número más pequeño posible, ejecutamos el programa con un valor de  $k$  entre 4 y 9, ya que el mayor entero que acepta C++ tiene 10 dígitos.

El código es el siguiente:

```
1  int getSmallestNumber(std::vector<int> &v, int k) {
2
3      int N = v.size();
4
5      vector<int> copy(v);
6      auto pos = copy.begin();
7      string smallest_str = "";
8
9      while (smallest_str.size() < k){
10         int smallest = std::numeric_limits<int>::max();
11         for (auto i = copy.begin(); i != copy.end(); ++i){ //
12             encontramos minimo
13             if (*i < smallest){
14                 smallest = *i;
15                 pos = i;
16             }
17         }
18         smallest_str.append(to_string(smallest)); //append the min
19         copy.erase(pos);
20     }
21     return (stoi(smallest_str));
22 }
```

Brute Force de “El número más pequeño”

Tras el análisis teórico comprobamos que este código tiene una eficiencia de  $O(k \cdot n)$ ,  $k$  al ser constante y menor que  $n$  no debería afectar a la eficiencia, también debido a la limitación impuesta por nosotros de que sea menor que 9, en un caso más general conforme crece la  $k$  se acercará a  $O(n)$ .

### 3.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en ordenar el vector utilizando quickSort (para dividir y vencer) y una vez lo tenemos ordenado simplemente cogemos los  $k$  primeros números.

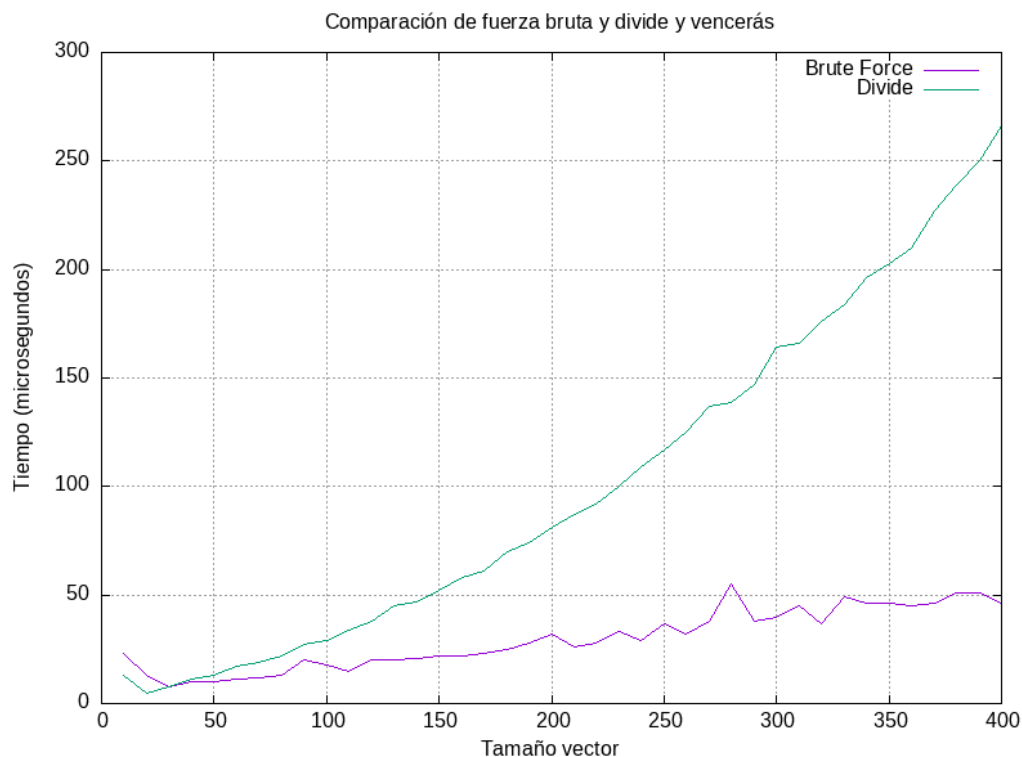
El código es el siguiente:

```
1 int getSmallestNumber(vector<int> v, int k)
2 {
3     quicksort(v, 0, v.size() - 1);
4     string result = "";
5     for (int i = 0; i < k; ++i)
6     {
7         result += to_string(v[i]);
8     }
9     return stoi(result);
10 }
```

### Divide y vencerás de “El número más pequeño”

Tras el análisis teórico comprobamos que este código tiene una eficiencia de  $O(n \cdot \log(n))$ , que ya conocíamos de la práctica anterior, aunque debido a la cantidad de dígitos repetidos su eficiencia decae hasta casi  $O(n)$ .

## 3.3. Tablas y Gráficos de Rendimiento



## 3.4. Análisis de los Resultados

Tras el análisis teórico y visualizar la gráfica comparativa, vemos que el algoritmo de Fuerza Bruta consigue resolver el problema de forma más eficiente que el algoritmo Divide y Vencerás. Esto se debe a que el primer algoritmo depende de una constante  $k$  que consideramos siempre

menor que  $n$ , y por tanto no afecta a la eficiencia. Mientras que en el segundo algoritmo podemos apreciar que quicksort tiene la eficiencia del peor caso, ya que al solo usar dígitos del 1 al 9, encontramos en el vector una gran cantidad de números repetidos lo cual afecta gravemente a la eficiencia de este algoritmo de ordenación.

## 4. El par de puntos más cercano

### 4.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en un doble bucle for el cual calcula todas las distancias de todos los puntos con todos, haciendo comparaciones hasta encontrar la mínima distancia.

El código es el siguiente:

```
1 double bruteForce(const vector<Point>& P, int l, int r) {
2     double minDist = DBL_MAX;
3     for (int i = l; i <= r; i++) {
4         for (int j = i+1; j <= r; j++) {
5             double d = dist(P[i], P[j]);
6             if (d < minDist) {
7                 minDist = d;
8             }
9         }
10    }
11    return minDist;
12 }
```

Brute Force de “El par de puntos más cercano”

Tras el análisis teórico comprobamos que este código tiene una eficiencia de  $O(n^2)$ .

### 4.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en ir haciendo divisiones de un array de puntos previamente ordenado por la coordenada  $x$  de los puntos (de derecha a izquierda). Tras esto, vamos haciendo divisiones partiendo el vector por la mitad recursivamente hasta quedarnos con 3 puntos o menos en una franja, donde se resolverá el problema usando fuerza bruta. Así obtenemos la mínima distancia en una franja, tras esto nos quedamos con la mínima distancia entre la franja de la izquierda y la derecha, y la usaremos para estudiar la mínima distancia entre una franja intermedia de tamaño la mínima distancia, creamos esta franja con los puntos que estan a una distancia menor de la menor distancia previamente calculada respecto del punto central. Una vez construido el strip comparamos los puntos con respecto a su coordenada  $y$  (esta comparación se hace con un número constante de puntos al reducirlos con la distancia mínima), se hace para evitar que puntos cercanos respecto a la coordenada  $x$  pero alejados respecto de la  $y$  sean los únicos que se comparan, buscando que sea menor que la menor distancia previa, obtenemos así la mínima distancia entre la franja izquierda, derecha y central. Resolviendo recursivamente obtenemos la mínima distancia en un

conjunto de puntos.

El código es el siguiente:

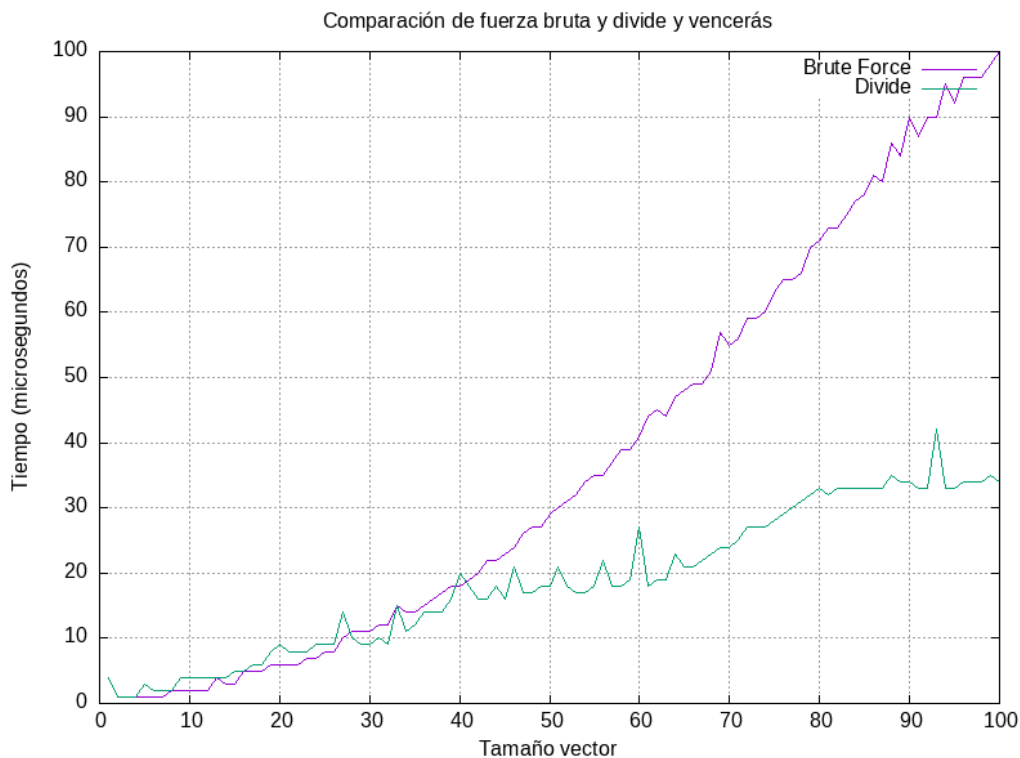
```
1 double closestUtil(vector<Point> &PX, int l, int r) {
2     // Si el subrango es peque, fuerza bruta
3     if (r - l <= 3) {
4         return bruteForce(PX, l, r);
5     }
6
7     int mid = (l + r) / 2;
8     double midX = PX[mid].x;
9
10    // bestPair se va actualizando;
11    // para tener la minima distancia, llamo recursivamente a izq y
12    der
13    double dl = closestUtil(PX, l, mid);
14    double dr = closestUtil(PX, mid+1, r);
15
16    double d = dl < dr ? dl : dr;
17    // Construir el strip (puntos que quedan en [midX - d, midX + d
18    ])
19    vector<Point> strip;
20    strip.reserve(r - l + 1);
21    for (int i = l; i <= r; i++) {
22        if (fabs(PX[i].x - midX) < d) {
23            strip.push_back(PX[i]);
24        }
25    }
26    // Ordenar la franja por Y
27    sort(strip.begin(), strip.end(), [](auto &a, auto &b){
28        return a.y < b.y;
29    });
30
31    // Revisar cada punto del strip con sus siguientes
32    // mientras la diferencia en Y sea < d
33    for (int i = 0; i < (int)strip.size(); i++) {
34        for (int j = i+1; j < (int)strip.size() && (strip[j].y
35        - strip[i].y) < d; j++) {
36            double distStrip = dist(strip[i], strip[j]);
37            if (distStrip < d) {
38                d = distStrip;
39            }
40        }
41    }
42
43    return d;
44 }
```

Divide y vencerás de “El par de puntos más cercano”

Para el análisis teórico de este algoritmo debemos estudiar la ecuación de recurrencia:  $T(n) = 2T(\frac{n}{2}) + O(n)$ . Esta ecuación se obtiene de que una llamada a la función de tamaño  $n$  hará dos llamadas recursivas con cada mitad de los puntos (izquierda y derecha),  $O(n)$  se obtiene de calcular los puntos en el strip central, que aunque pueda parecer  $O(n)$ , cada punto del strip solo se compara con un número constante de puntos, al habernos quedado previamente con una cantidad de puntos pequeñas dentro de un rango. Cuando el tamaño es menor o igual que 3 (caso base), donde aplicamos fuerza bruta, al ser un tamaño tan pequeño, no afectará a la eficiencia.

Tras resolver la recurrencia comprobamos que este código tiene una eficiencia de  $O(n \cdot \log(n))$ .

### 4.3. Tablas y Gráficos de Rendimiento



### 4.4. Análisis de los Resultados

Como se esperaba tras el análisis teórico, en las gráficas podemos observar que el algoritmo Divide y Vencerás tiene una mayor eficiencia que el de Fuerza Bruta, también podemos observar la clara forma parabólica del algoritmo Fuerza Bruta como habíamos previsto. También se puede observar la región a partir del cual el primer algoritmo, comienza a tener tiempos de ejecución más bajos que el segundo.



## 5. La envolvente convexa

### 5.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en empezar seleccionando el punto más a la izquierda, el cual sabemos que siempre va a estar en la envolvente convexa, seguidamente seleccionas otro punto y compruebas si la recta que une los dos puntos deja el resto de puntos a la izquierda, una vez encuentras ese punto, es nuevo punto actúa como el primero que seleccionaste, y haces esto hasta que encuentres todos los puntos pertenecientes a la envolvente convexa.

El código es el siguiente:

```
1 // Estructura para representar un punto
2 struct Point {
3     float x, y;
4 };
5
6 typedef pair<int, int> ParPuntos;
7
8 // Funcion que calcula el convex hull por fuerza bruta
9 set<ParPuntos> convexHullBruteForce(const vector<Point>& points) {
10     int n = points.size();
11     set<ParPuntos> hull;
12
13     // Recorremos cada par de puntos
14     for (int i = 0; i < n; i++) {
15         for (int j = i + 1; j < n; j++) {
16             int left = 0, right = 0;
17
18             // Verificamos la posicion de cada otro punto respecto a la
19             // linea formada por points[i] y points[j]
20             for (int k = 0; k < n; k++) {
21                 if (k == i || k == j)
22                     continue;
23
24                 float position = (points[j].x - points[i].x) * (points[
25                     k].y - points[i].y) -
26                     (points[j].y - points[i].y) * (points[k].x - points[i].
27                     x);
28
29                 if (position > 0)
30                     left++;
31                 else if (position < 0)
32                     right++;
33             }
34
35             // Si todos los puntos se encuentran de un mismo lado (o son
36             // colineales), se adjuntan los puntos de la arista
37             if (left == 0 || right == 0) {
```

```

34         hull.insert({points[i].x, points[i].y});
35         hull.insert({points[j].x, points[j].y});
36     }
37 }
38 }
39
40 return hull;
41 }

```

### Brute Force de “La envoltente convexa”

Tras el análisis teórico comprobamos que este código tiene una eficiencia de  $O(n^3)$

## 5.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en ir dividiendo los puntos del plano, previamente ordenados de derecha a izquierda, en dos partes (izquierda y derecha) hasta llegar a un tamaño pequeño donde se calculen las envoltentes convexas, tanto de izquierda como de derecha que se juntarán usando la función merge. Esta función comienza buscando de entre los puntos de la envoltente de la derecha el punto más a la izquierda, actuando analogamente con los de la izquierda, tras esto buscamos la tangente superior que cuyos puntos formarán parte de la envoltente. Para ello, vamos buscando con el punto más a la derecha de la izquierda fijo, y rotando el punto más a la izquierda de los derecha, hacia la derecha hasta encontrar uno que deje el resto de puntos a la derecha, una vez lo encontramos fijamos ese punto y rotamos hacia la izquierda los puntos de la izquierda hasta encontrar uno que deje todos los puntos de la izquierda a la izquierda. Repetimos para la tangente inferior rotando en sentido contrario, finalizamos guardando todos los puntos resultantes en el vector a devolver.

El código es el siguiente:

```

1  vector<Point> convexHullDC(vector<Point> &points, int l, int r) {
2      // Base case: one point
3      if (l == r) {
4          return {points[l]};
5      }
6      // Base case: two points
7      if (r - l == 1) {
8          if (points[l].x == points[r].x && points[l].y == points[r].y)
9              return {points[l]};
10         return {points[l], points[r]};
11     }
12     int mid = (l + r) / 2;
13     vector<Point> leftHull = convexHullDC(points, l, mid);
14     vector<Point> rightHull = convexHullDC(points, mid + 1, r);
15     return mergeHulls(leftHull, rightHull);
16 }

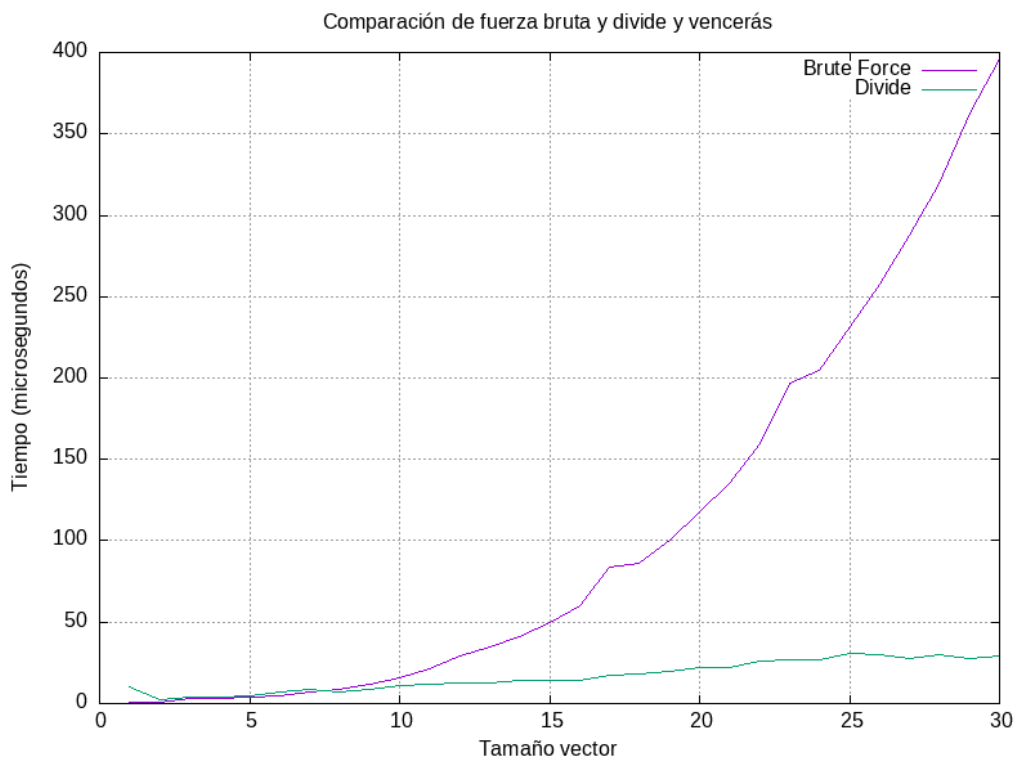
```

### Divide y vencerás de “La envoltente convexa”

Para el análisis teórico de este algoritmo debemos estudiar la ecuación de recurrencia:  $T(n) = 2T(\frac{n}{2}) + O(n)$ . Esta ecuación se obtiene de que una llamada a la función de tamaño  $n$  hará dos llamadas recursivas con cada mitad de los puntos (izquierda y derecha),  $O(n)$  se obtiene de la llamada a la función `merger` que junta las dos envolventes con una eficiencia de  $O(n)$ , ya que hay se recorren varios bucles de la mitad del tamaño en cada iteración ( $\frac{n}{2}$ ) obviando la constante obtenemos  $O(n)$ . El caso base usa la fuerza bruta en un conjunto de 5 puntos, su eficiencia es presecindible.

Tras resolver la recurrencia comprobamos que este código tiene una eficiencia de  $O(n \cdot \log(n))$ .

### 5.3. Tablas y Gráficos de Rendimiento



### 5.4. Análisis de los Resultados

Como se esperaba tras el análisis teórico, en las gráficas podemos observar que el algoritmo Divide y Vencerás tiene una mayor eficiencia que el de Fuerza Bruta, también podemos observar la clara forma cúbica del algoritmo Fuerza Bruta como habíamos previsto. También se puede observar la región a partir del cual el primer algoritmo, comienza a tener tiempos de ejecución más bajos que el segundo.

En esta sección se describen los diferentes problemas que se abordarán.

## 6. Conclusiones

Tras esta práctica hemos descubierto que la estrategia de desarrollo de algoritmos Divide y Vencerás es útil en un amplio repertorio de casos pero esto no quiere decir que siempre sea

mejor que un algoritmo fuerza bruta, esto lo podemos comprobar en 3. Aún así en la gran mayoría de casos aplicar Divide y Vencerás supone una mejoría en la eficiencia del código.