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

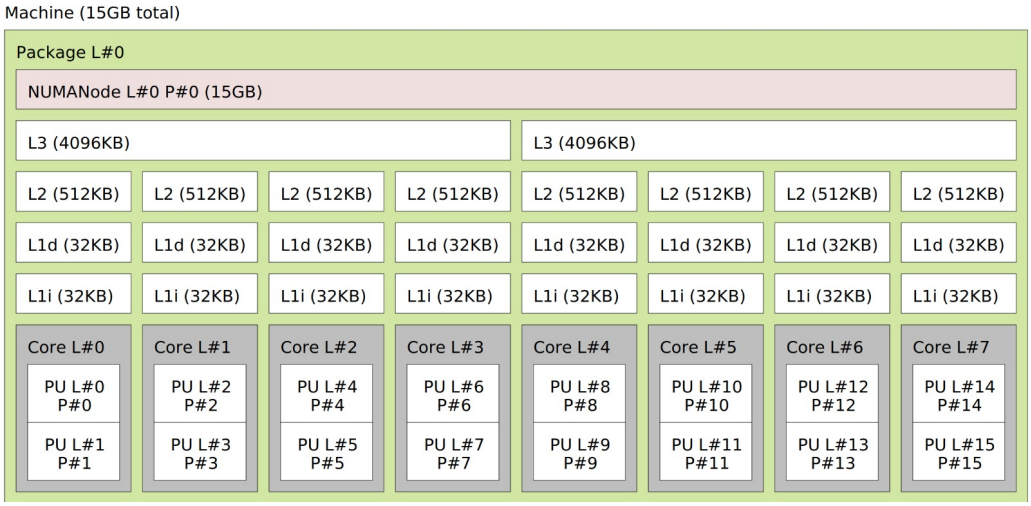
1. Introducción

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

- Menor número de k cifras
- Menor distancia entre puntos
- Envolvente convexa de puntos

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 10 veces, nos quedaremos con una media de todas las ejecuciones y podremos comparar la eficiencia de ambas soluciones, viendo a partir de qué 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 imágen:



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. Cómo ejecutar

Para compilar todos los programas, basta con abrir una terminal en el directorio `ficheros_cpp` y ejecutar el siguiente comando:

```
$ make
```

El archivo `Makefile` se encargará automáticamente de compilar todos los ficheros fuente y generar los ejecutables correspondientes. Una vez compilados, los programas podrán ejecutarse directamente desde la terminal. Se ejecutan fácilmente proporcionando un archivo de salida, un número que actúe como semilla para los generadores de números aleatorios, y un entero n que indique el tamaño de la prueba (número de puntos):

```
$ ./programa [archivo_salida] [semilla] [tamaño problema]
```

4. El número más pequeño

4.1. 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:

```
int getSmallestNumber(std::vector<int> &v, int k) {

    int N = v.size();

    vector<int> copy(v);
    auto pos = copy.begin();
    string smallest_str = "";

    while (smallest_str.size() < k){
        int smallest = std::numeric_limits<int>::max();
        for (auto i = copy.begin(); i != copy.end(); ++i){ //
           encontramos minimo
            if (*i < smallest){
                smallest = *i;
                pos = i;
            }
        }
        smallest_str.append(to_string(smallest)); // metemos el minimo
        copy.erase(pos);
    }

    return (stoi(smallest_str));
```

```
}
```

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 nosostros de que sea menor que 9. Sin embargo, en un caso más general, si pudieramos escoger un número de más y más cifras, la k crecería y la eficiencia se acercaría a $O(n^2)$.

4.2. Análisis de Divide y Vencerás

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

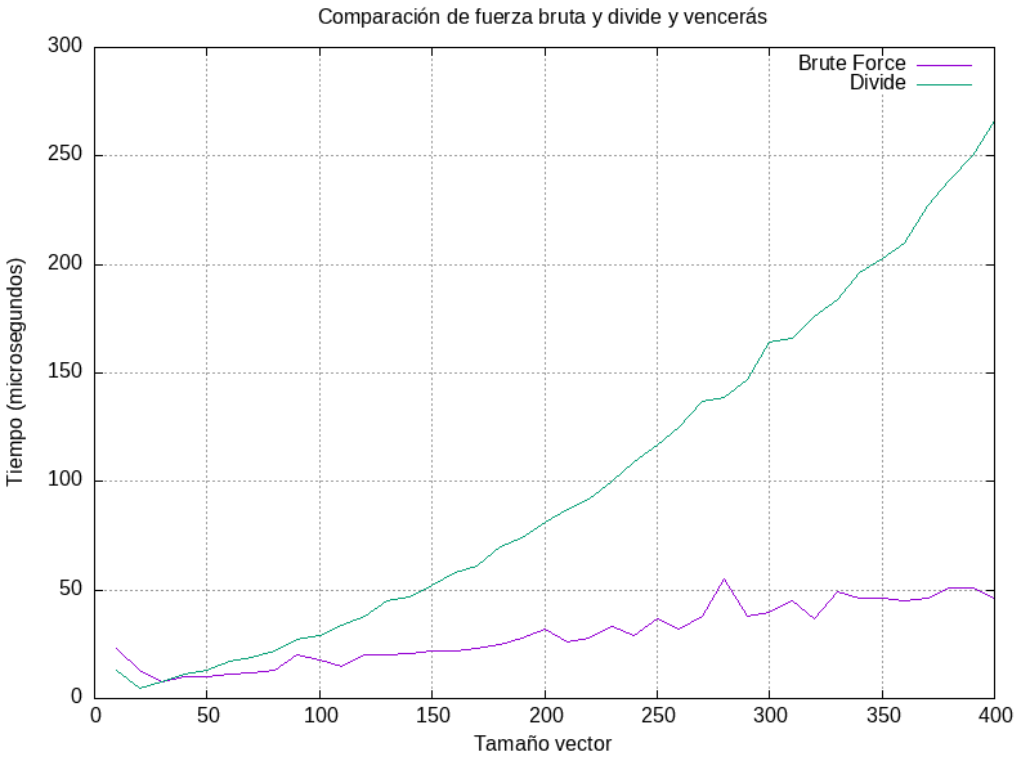
El código es el siguiente:

```
int getSmallestNumber(vector<int> v, int k)
{
        quicksort(v, 0, v.size() - 1);
        string result = "";
        for (int i = 0; i < k; ++i)
        {
                result += to_string(v[i]);
        }
        return stoi(result);
}
```

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 conociamos de la práctica anterior, aunque debido a la cantidad de dígitos repetidos su eficiencia decae hasta casi $O(n^2)$.

4.3. Tablas y Gráficos de Rendimiento



4.4. Análisis de los Resultados

Tras el analisis 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 algortimo depende de una constante k que consideramos siempre menor que n, acotada entre 4 y 9, 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 algortimo de ordenación.

5. El par de puntos más cercano

5.1. 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++) {
```

```
                        double d = dist(P[i], P[j]);
                        if (d < minDist) {
                                minDist = d;
                        }
                }
        }
        return minDist;
}
```

Brute Force de **"El par de puntos más cercano"**

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

5.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en hacer divisiones de un array de puntos previamente ordenado por la coordenada x (de derecha a izquierda).

A partir de ahí, se divide el vector por la mitad de forma recursiva hasta que se obtienen subconjuntos de tres puntos o menos, en los cuales se resuelve el problema mediante fuerza bruta.

De esta forma, se calcula la mínima distancia en cada franja. Luego, se toma la mínima distancia entre la franja izquierda y la derecha, y se utiliza para estudiar una franja intermedia de ancho igual a esa distancia mínima.

Esta franja intermedia se construye con los puntos que se encuentran a una distancia menor que la mínima previamente calculada con respecto al punto central.

Una vez construido el *strip*, se comparan los puntos según su coordenada y. Esta comparación se limita a un número constante de puntos, ya que se filtran usando la distancia mínima.

Esto se hace para evitar comparar únicamente puntos cercanos en la coordenada x pero alejados en y, lo cual no sería eficiente. Se busca siempre encontrar una distancia menor que la previamente conocida.

Así se obtiene la mínima distancia considerando la franja izquierda, la derecha y la central. Finalmente, resolviendo de manera recursiva, se obtiene la mínima distancia entre todos los puntos del conjunto.

El código es el siguiente:

```
double closestUtil(vector<Point> &PX, int l, int r) {
        // Si el subrango es peque, fuerza bruta
        if (r - l <= 3) {
                return bruteForce(PX, l, r);
        }

        int mid = (l + r) / 2;
        double midX = PX[mid].x;

        // bestPair se va actualizando;
        // para tener la minima distancia, llamo recursivamente a izq y
            der
```

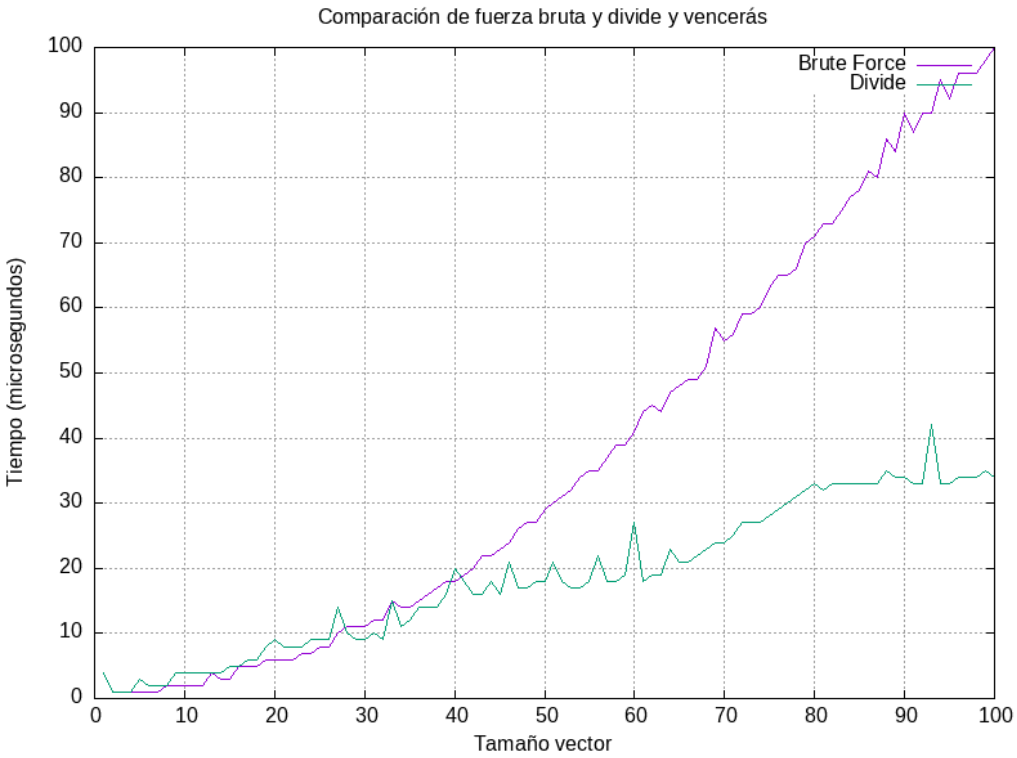
```
        double dl = closestUtil(PX, l, mid);
        double dr = closestUtil(PX, mid+1, r);

        double d = dl < dr ? dl : dr;
        // Construir el strip (puntos que quedan en [midX - d, midX + d])
        vector<Point> strip;
        strip.reserve(r - l + 1);
        for (int i = l; i <= r; i++) {
                if (fabs(PX[i].x - midX) < d) {
                        strip.push_back(PX[i]);
                }
        }
        // Ordenar la franja por Y
        sort(strip.begin(), strip.end(), [](auto &a, auto &b){
                return a.y < b.y;
        });

        // Revisar cada punto del strip con sus siguientes
        // mientras la diferencia en Y sea < d
        for (int i = 0; i < (int)strip.size(); i++) {
                for (int j = i+1; j < (int)strip.size() && (strip[j].y - strip[i].y) < d; j++) {
                        double distStrip = dist(strip[i], strip[j]);
                        if (distStrip < d) {
                                d = distStrip;
                        }
                }
        }

        return d;
}
```

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 auqnue 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 $O(n \cdot \log(n))$.

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 parábolica del algoritmo Fuerza Bruta como habiamos previsto. También se puede observar la región a partir del cual el primer algortimo, comienza a tener tiempos de ejecución más bajos que el segundo.

6. La envolvente convexa

6.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en comenzar seleccionando el punto más a la izquierda, ya que sabemos que este siempre formará parte de la envolvente convexa.

A continuación, se selecciona otro punto y se comprueba si la recta que une ambos deja al resto de los puntos a su izquierda.

Una vez encontrado ese punto, pasa a desempeñar el mismo papel que el primero, y repetimos el proceso hasta identificar todos los puntos que pertenecen 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
              linea formada por points[i] y points[j]
19         for (int k = 0; k < n; k++) {
20                 if (k == i || k == j)
21                         continue;
22
23                 float position = (points[j].x - points[i].x) * (points[
                      k].y - points[i].y) -
24                 (points[j].y - points[i].y) * (points[k].x - points[i].
                      x);
25
26                 if (position > 0)
27                         left++;
28                 else if (position < 0)
29                         right++;
30         }
31
32         // Si todos los puntos se encuentran de un mismo lado (o son
              colineales), se adjuntan los puntos de la arista
33         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 envolvente convexa"**

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

6.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en dividir progresivamente los puntos del plano (previamente ordenados de derecha a izquierda) en dos partes: izquierda y derecha.

Esta división continúa hasta que se alcanza un tamaño pequeño, donde se calculan las envolventes convexas de cada parte por separado. Posteriormente, ambas envolventes se combinan utilizando la función *merge*.

Esta función comienza buscando, entre los puntos de la envolvente derecha, el que esté más a la izquierda, y de manera análoga en la envolvente izquierda.

A continuación, se determina la tangente superior, cuyos puntos formarán parte de la envolvente final. Para ello, se fija el punto más a la derecha de la envolvente izquierda y se rota el punto más a la izquierda de la envolvente derecha hacia la derecha hasta encontrar uno que deje al resto de puntos a su derecha.

Una vez hallado, se fija dicho punto y se rota hacia la izquierda la envolvente izquierda hasta encontrar un punto que deje a todos sus compañeros a la izquierda.

Este mismo proceso se repite para hallar la tangente inferior, rotando en sentido contrario.

Finalmente, todos los puntos resultantes se almacenan en el vector que se devolverá.

El código es el siguiente:

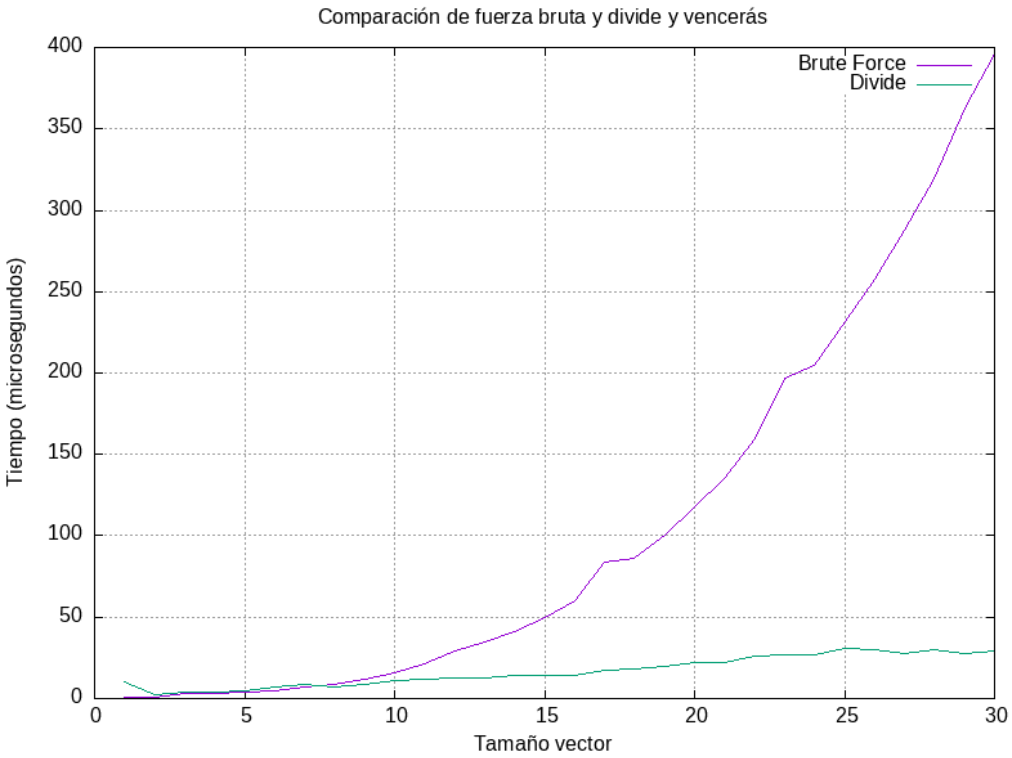
```
vector<Point> convexHullDC(vector<Point> &points, int l, int r) {
        // Base case: one point
        if (l == r) {
                return {points[l]};
        }
        // Base case: two points
        if (r - l == 1) {
                if (points[l].x == points[r].x && points[l].y == points
                   [r].y)
                        return {points[l]};
                return {points[l], points[r]};
        }
        int mid = (l + r) / 2;
        vector<Point> leftHull = convexHullDC(points, l, mid);
        vector<Point> rightHull = convexHullDC(points, mid + 1, r);
        return mergeHulls(leftHull, rightHull);
}
```

Divide y vencerás de **"La envolvente 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 $O(n \cdot \log(n))$.

6.3. Tablas y Gráficos de Rendimiento



6.4. Análisis de los Resultados

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

7. 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 4. Aún así en la gran mayoría de casos aplicar Divide y Vencerás supone una mejoría en la eficiencia del código.