Grado en Ingeniería Informática (Plan 2015)

UNIVERSIDAD DE ALMERÍA

Estructura de Datos y Algoritmos II

Curso 2024-2025

Práctica 1 – Divide and Conquer - Controlando el espacio aéreo

---

Alumnos:

Alejandro Doncel Delgado

Jorge Godoy Beltrán

Índice de contenidos

1. OBJETIVO

1.1. Justificación de la aplicación del método Divide and Conquer

En esta primera práctica de la asignatura se considera el método Divide and Conquer (divide y vencerás) debido a sus ventajas para resolver problemas de forma eficiente cuando es posible descomponerlos en subproblemas de menor tamaño. En el contexto de la búsqueda del par de puntos más cercano, esta técnica permite:

- Dividir el conjunto de datos en secciones manejables y resolver subproblemas más pequeños.
- Combinar los resultados parciales para obtener la solución global.
- Reducir el tiempo de ejecución de manera significativa frente a algoritmos iterativos exhaustivos, especialmente cuando se manejan grandes volúmenes de datos.

La elección de este enfoque obedece a la necesidad de optimizar la búsqueda del par de puntos que se encuentran a menor distancia entre sí, evitando el alto coste computacional de métodos puramente iterativos que requieren revisar todas las parejas de puntos.

1.2. Descripción general del problema del "par de puntos más cercano"

El problema consiste en identificar, dentro de un conjunto de puntos en el plano 2D, aquellos dos cuya distancia sea la mínima posible. Para ello, cada punto se representa por sus coordenadas (x,y). El principal reto radica en gestionar eficientemente el crecimiento del número de comparaciones, puesto que, con métodos de fuerza bruta, el total de parejas a considerar aumenta cuadráticamente con el número de puntos.

De forma específica, se busca:

- Calcular la distancia entre puntos siguiendo la métrica euclídea.
- Emplear distintas estrategias de mejora —desde un enfoque iterativo simple hasta variantes del método Divide and Conquer— con el fin de comparar rendimientos y complejidades.
- Validar la solución en escenarios reales, donde el volumen de datos puede ser elevado y la eficiencia del algoritmo resulta prioritaria.

2. TRABAJO EN EQUIPO

2.1. Organización y roles

Con el fin de llevar a cabo la práctica de manera ordenada y eficiente, se ha optado por establecer un rol de liderazgo y roles específicos para cada integrante del equipo. El propósito es asegurar que cada persona tenga una responsabilidad clara y una contribución definida al proyecto, de la siguiente forma:

- **Alejandro Doncel [AD]**: Asume la coordinación general, planifica la estructura del proyecto en el repositorio, métodos dependientes, documenta las funciones y colabora en la depuración de los algoritmos.
- **Jorge Godoy [JG]**: Desarrolla la parte principal del código (algoritmos iterativos y recursivos).

2.2. Tabla de distribución de tareas

Tarea	Responsable(s)	Fecha de Finalización
Configuración del repositorio y estructura de carpetas	[AD]	6 de marzo 2025
Implementación de algoritmo iterativo (fuerza bruta)	[AD], [JG]	8 de marzo 2025
Implementación de algoritmo iterativo mejorado	[JG]	10 de marzo 2025
Desarrollo del algoritmo DyV (versión inicial)	[JG]	10 de marzo 2025
Optimización y combinación (regla de los 7, franja)	[AD]	17 de marzo 2025
Estudio experimental (pruebas y gráficos)	[AD]	19 de marzo 2025
Análisis teórico y validación de resultados	[JG]	17 de marzo 2025
Documentación y memoria (redacción final)	[AD], [JG]	25 de marzo 2025
Revisión y entrega	[AD], [JG]	29 de marzo 2025

3. ANTECEDENTES

El presente trabajo tiene como objetivo aplicar el método de Divide y Vencerás a la resolución del problema del par de puntos más cercano en un conjunto de datos, con aplicaciones concretas en el control del tráfico aéreo, análisis de datos espaciales y optimización de rutas.

Para abordar este problema, se han evaluado tanto soluciones iterativas como recursivas, analizando sus ventajas y limitaciones. La comparación entre ambos enfoques representa una oportunidad para profundizar en conceptos clave de diseño y análisis de algoritmos, permitiendo observar cómo pequeñas optimizaciones pueden traducirse en una mejora significativa del rendimiento computacional.

A lo largo de la práctica, se han explorado distintos enfoques algorítmicos, evaluando sus alternativas con el propósito de encontrar un equilibrio entre precisión y eficiencia computacional. Las principales soluciones consideradas han sido:

3.1. Alternativas contempladas y solución final adoptada

Enfoque de Fuerza Bruta (iterativo)

Se consideró este método como punto de partida por su sencillez conceptual y facilidad de implementación. La complejidad, que crece cuadráticamente con el número de puntos, se vuelve poco manejable a partir de tamaños de entrada elevados.

Resulta adecuado para validar resultados en casos pequeños o en tareas de depuración, pero no para volúmenes grandes de datos.

Métodos Iterativos Mejorados

Se introdujo la reducción del número de comparaciones prescindiendo de las repeticiones innecesarias. Aunque mejora la implementación inicial, el crecimiento de la complejidad sigue siendo $O(n^2)$, resultando todavía costoso para grandes conjuntos.

Enfoque Divide and Conquer

Se examinaron varias versiones del método Divide and Conquer, comenzando con la partición del conjunto de puntos en dos mitades y la combinación de resultados.

Se valoró su eficacia para disminuir significativamente el número de comparaciones, logrando complejidades inferiores a la aproximación iterativa pura.

Optimizaciones en la Etapa de Combinación

Se consideraron optimizaciones como la franja central, la regla de los 7 y el reordenamiento de puntos según las coordenadas x e y, con el fin de acotar la búsqueda de candidatos de manera más selectiva.

Estas mejoras reducen la necesidad de examinar todos los pares entre ambas mitades, manteniendo la corrección del resultado.

Solución final adoptada

La propuesta final integra la versión más avanzada del método Divide and Conquer, que contempla:

- Ordenar inicialmente los puntos por coordenada x (y posteriormente también por coordenada y).
- Dividir el conjunto en dos subproblemas equilibrados.
- Combinar los resultados mediante criterios de optimización (franja central, regla de los 7).
- Comparar cada nuevo punto únicamente con un número limitado de vecinos en la franja, lo cual reduce el coste computacional de la etapa de combinación.

De este modo, se logra un algoritmo con una complejidad cercana a O (n log n) en la práctica, ofreciendo un rendimiento muy superior al de las variantes iterativas para tamaños de problema elevados y garantizando, a la vez, una correcta localización de la pareja de puntos más próximos.

4. ESTUDIO TEÓRICO Y DE IMPLEMENTACIÓN

4.1. Introducción

Tras haber introducido el proceso de desarrollo e implementación del algoritmo para encontrar el par de puntos más cercanos mediante el método **Divide and Conquer**, en este apartado se han llevado a cabo diversos análisis de cada uno de los algoritmos requeridos para la práctica. Cada estudio seguirá una estructura definida y, al final, se compararán los resultados obtenidos. La estructura para seguir es la siguiente:

1. **Estudio de la implementación:** en esta fase se explican los detalles más importantes de las implementaciones
2. **Estudio teórico:** estudiar los tiempos de ejecución de los algoritmos implementados, determinando la complejidad temporal y espacial.
3. **Estudio experimental:** en este aparatado se contractarán los resultados teóricos y los experimentales, comprobando si los experimentales confirman los teóricos previamente analizados.
4. **Anexo:** Este apartado es fundamental dentro de la sección, ya que incluirá enlaces a contenidos clave relacionados con los algoritmos, como los pseudocódigos de cada uno de ellos.

Se trata de una sección extensa e introductoria de cada uno de los algoritmos, más adelante se compararán unos con otros, agrupándolos en iterativos o recursivo.

4.2. Algoritmo Iterativo (fuerza bruta)

Se trata de un **algoritmo iterativo**, utilizando un recorrido exhaustivo (fuerza bruta) sobre el array *P* calculando la distancia de cada uno de los pares de puntos (uno a uno) y devolviendo aquél que tenga la menor distancia. Este algoritmo genera un total de n^2 posibles pares y se escogerá de ellos el que tenga la menor distancia.

Estudio de implementación - Funcionamiento del algoritmo

Recorre todos los pares posibles del conjunto haciendo uso de dos bucles anidados

- **Primer bucle:** recorre desde el primer punto hasta el penúltimo
- **Segundo bucle:** permite recorrer todos los puntos y poder comparar distancias

Dentro del segundo bucle debe existir un par de condiciones, de manera que i == j, se salta la comparación pues no se compara consigo mismo, y, por otro lado, calcular la distancia entre cada par.

Una vez finalizado ambos bucles el mejor par guardado (el que tiene menor distancia euclídea) será devuelto por el método, y obteniéndose el resultado requerido.

Estudio Teórico - Análisis de la complejidad del algoritmo

1. El primer bucle (i) se ejecuta n-1 veces, eso implicado que ejecuta prácticamente todo el array, teniendo una complejidad de → O(n)
2. El segundo bucle se ejecuta n veces para cada i. → O(n)
3. Posteriormente dentro del bucle anidado hay operaciones de comparación y cálculo de distancia, que son de coste constante **O(1)**.

Por lo tanto, al tratarse de dos bucles anidados, el orden de complejidad sería de **O(n^2)**

Análisis de eficiencia

- **Ventajas:** implementación sencilla y directa, funcionamiento con eficiencia en conjuntos de puntos pequeños n < 1000.
- **Desventajas:** la complejidad cuadrática lo hace ineficiente para conjuntos de datos grandes, al igual que se hacen cálculos redundantes ya que se calcula la distancia de (i, j) y (j, i), ya que la distancia es simétrica. Tanto una desventaja como la otra se mejorará en los algoritmos siguientes, empezando por eliminar la redundancia con el **Algoritmo Iterativo Mejorado.**

4.3. Algoritmo Iterativo Mejorado (fuerza bruta)

Este algoritmo es una mejora sobre la versión de fuerza bruta simple, siendo su objetivo reducir el número de comparaciones eliminando redundancias en el cálculo de distancias. En lugar de comparar todos los pares de puntos dos veces, ahora solo se considera cada par una única vez.

El algoritmo genera un total de (n(n-1) /2) posibles pares en lugar de n^2, lo que reduce a la mitad el número de cálculos y mejora ligeramente la eficiencia.

Estudio de implementación - Funcionamiento del algoritmo

La parte fundamental de este algoritmo vuelven a ser los bucles

1. **Primer bucle:** se recorre desde el primer punto hasta el penúltimo, siendo igual que el algoritmo iterativo básico, un orden de complejidad de **O(n)**
2. **Segundo bucle:** a diferencia del anterior, este bucle anidado comienza en el siguiente punto int j = i+1, asegurando que cada par se compare solo una vez, y así evitar redundancia

Dentro de segundo bucle se calcula la distancia euclídea entre los puntos[i] y punto[j], y en caso de que la distancia sea menor que la mínima encontrada hasta el momento, se actualiza el mejor par.

Una vez que finalizan ambos bucles, el par con menor distancia es devuelto como resultado.

Estudio Teórico - Análisis de la complejidad del algoritmo

1. **Primer bucle:** se ejecuta n-1 veces, dando una complejidad de O(n).
2. **Segundo bucle:** se ejecuta n-i-1 veces, lo que da una suma total de ejecuciones de:

$$\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}$$

Esto resulta en una complejidad de $O(n^2)$ en el peor de los casos.

3. Operaciones dentro del bucle como el cálculo de distancia y comparación es 0(1), lo que no afecta el orden de complejidad global.

Aunque la complejidad sigue siendo $O(n^2)$, el número de comparaciones se reduce aproximadamente a la mitad en comparación con la versión anterior, lo que mejora de manera significante el rendimiento.

Análisis de Eficiencia

- **Ventajas:** sigue siendo una implementación sencilla y fácil de entender, reduciendo el número de comparaciones en un factor de 2 con respecto al algoritmo de Algoritmo Iterativo Básico.
- **Desventajas:** se trata de una versión más eficiente que la anterior, pero sigue teniendo una complejidad $O(n^2)$. Más adelante se verán algoritmo de Divide and Conquer que permitirán obtener una menor complejidad.

4.4. Algoritmo Divide y Vencerás Simple

Este es el primero de cuatro algoritmos recursivos analizados en esta práctica. Implementa la estrategia de "Divide y Vencerás" para encontrar el par de puntos más cercanos dentro de un conjunto. A diferencia de los enfoques anteriores, este método descompone el problema en subproblemas más pequeños hasta alcanzar un caso base, resolviéndolos de forma recursiva y combinando los resultados para obtener la solución final.

Más adelante, veremos que esta es la versión básica del algoritmo de Divide y Vencerás, y que los siguientes optimizarán su eficiencia y reducirán su complejidad. Aunque este enfoque es recursivo en lugar de iterativo, aún mantiene una complejidad de **$O(n^2)$**.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo consta de dos métodos

- **`Par divideVencerasSimple(Punto[] puntos)`:** Encargado de ordenar los putos por la coordinada X e Y mediante el uso de la clase Arrays.sort() y el comparador. Por otro lado, es el encargado de verificar que no existen puntos repetidos, ya que en el caso de que sí los haya ese será el punto solución. Y finalmente llama al método privado que está explicado a continuación.

- **`Par divideVencerasSimple(Punto[] puntos, int inicio, int fin)`:** método que cumple con el esquema general de Divide y Vencerás (DyV). Por un lado, está un primer fragmento que determina cual es el caso base del problema, que este caso y cuando el subproblema tiene igual o menos de 3 puntos.

  La segunda parte que hay que destacar es la parte recursiva, siendo esta la más importante a la hora de calcular su orden de complejidad, y permite dividir el problema en subproblemas del mismo tipo y **disjuntos.** Y por último la parte de combinar para obtener la solución al problema original, a partir de las soluciones de los subproblemas. Devuelve la solución al problema.

Estudio Teórico - Análisis de la complejidad del algoritmo

1. **Ordenación inicial → O(n log n):** antes de acceder al método privado donde se emplea recursividad, se debe ordenar los puntos por la coordenada X e Y, al tratarse de un ordenación como Merge Sort o Quick Sort es O(n log n)

2. **Evaluación con fuerza bruta los casos bases → O(1):** cuando el subproblema tiene ≤ 3 puntos, estos se compararán entre ellos y pasando a un complejidad constante.
3. **Fragmento recursivo:** al tener dos llamadas recursivas, cada nivel de la recursión divide el conjunto en dos partes y realiza una combinación de $O(n^2)$. Ya que se recoge el array, siendo n*n

   La conclusión es que, aunque se emplee un enfoque de divide y vencerás, esta básica versión no mejora la complejidad respecto a la fuerza bruta, ya que la combinación sigue siendo costosa.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2) = O(n^2)$$

Para optimizar este algoritmo es necesario modificar la fase de combinación, que es donde aumenta la complejidad, en futuras versiones será posible alcanzar la complejidad de **O(n log n).**

Análisis de eficiencia

- **Ventajas:** aun teniendo el mismo orden de complejidad que las cosas anteriores, este algoritmo será más eficiente a la hora de emplear grandes volúmenes de datos, ya que además realiza menos comparaciones innecesarias gracias a la división de la reducción de espacio de búsqueda.
- **Desventajas:** a diferencia de los dos casos anteriores, la implementación es mayor compleja respecto a la fuerza bruta, existe el requisito de conocer el esquema general de DyV, y, por último, requiere un paso extra qué es un ordenamiento previo.

4.5. Algoritmo Divide y Vencerás Mejorado

Este es el segundo de los algoritmos recursivos analizados en esta práctica, implementa una versión optimizada de "Divide y Vencerás" para encontrar el par de puntos más cercano dentro de un conjunto, que, a diferencia de la versión básica, esta mejora la eficiencia en la fase de combinación, reduciendo la cantidad de comparaciones innecesarias.

El problema es que, aunque se ha optimizado la parte conflictiva del algoritmo, el orden de complejidad sigue siendo el mismo, es decir, **$O(n^2)$,** pero en el estudio teórico al realizar menos comparaciones, es más eficiente.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo consta de dos métodos principales:

1. `Par divideVencerasMejorado(Punto[] puntos)`: encargado de validar que el array de puntos no sea nulo y que tenga al menos dos puntos, al igual,

que es el encargado de ordenar el array original haciendo uso del Arrays.sort() y el XYcomparator. Por último, verifica si existen puntos repetidos usando **comprobarRepetidos(copiaPuntos)**.

2. **`Par divideVencerasMejorado(Punto[] puntos, int inicio, int fin):`** método que hace uso de la recursividad en la cual destacan las 4 partes del esquema general de DyV. En primer lugar, está el caso base en la cual, si el subproblema tiene 3 o menos puntos, se comparan directamente por fuerza bruta. En segundo lugar, la parte de división que son las llamadas recursivas, que dividen el conjunto en dos mitades

   Una vez que se resuelto cada mitad esta la parte combinación, que es la parte que se ha optimizado en este algoritmo, en la cual se comparan las mejores soluciones de cada mitad, limitando las comparaciones a los puntos cercanos a la franja central, y así reduciendo la redundancia.

Estudio Teórico - Análisis de la complejidad del algoritmo

1. **Ordenación inicial → O(n log n):** antes de acceder al método privado donde se emplea recursividad, se debe ordenar los puntos por la coordenada X e Y, al tratarse de un ordenación como Merge Sort o Quick Sort es O(n log n).
2. Cada nivel de la recursión divide el conjunto en dos partes y realiza una combinación **O(n),** que al tratarse de dos llamadas recursivas sería una complejidad de O($\mathbf{n^2}$).
3. **Caso base con fuerza bruta:** cuando hay 3 o menos puntos, se comparan todos entre sí en O(1).

Inicialmente, el orden de complejidad de los algoritmos propuestos no mejora, ya que todos mantienen el mismo nivel de complejidad. Sin embargo, a medida que se avanza, se observa una mejora significativa en la eficiencia, lo que representa el primer paso hacia el desarrollo de un algoritmo verdaderamente optimizado para el problema de los puntos más cercanos.

$$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + O(n^2) = O(n^2)}$$

Análisis de eficiencia

- **Ventajas:** reduce el número de comparaciones innecesarias ya que se comparan únicamente las que están dentro de la franja central. Hasta el momento es de los algoritmos más eficientes implementados, superando a la versión básica y los de fuerza bruta.

- **Desventajas:** al igual que es la más eficiente hasta ahora, también se trata de la más compleja para implementar, y además de que sigue siendo de complejidad de **O($\mathbf{n^2}$)** en el peor de los casos. El peor de caso posible es si los puntos estén alineados verticalmente.

4.6. Algoritmo Divide y Vencerás Pro – Regla de los 7 puntos

En este tercer algoritmo se implementa una versión mejorada de “Divide y Vencerás” en comparación a los dos anteriores. Esto se consigue gracias a la regla de los 7 puntos, reduciendo de manera significativa la cantidad de comparaciones necesarias en la fase de combinación.

Ya no solo eso, eso uso de esta regla permite reducir el orden de complejidad, pasando de **O(n^2)** a **O(n log n),** en el análisis de la complejidad del algoritmo se explicará de maneras más detallada como se ha llegado a esta conclusión.

Estudio de implementación - Funcionamiento del algoritmo

1. **`public static Par divideVencerasPro(Punto[] puntos):`** Al igual que en los algoritmos anteriores, este método público comienza asegurando que el array de puntos no sea nulo y contenga al menos dos elementos. Luego de esta verificación, el array se ordena por la coordenada X utilizando Arrays.sort() junto con el comparador Xcomparador.

   En tercer lugar, se identifican los puntos duplicados mediante el método **comprobarRepetidos(copiaPuntos)**, al cual se le pasa la versión ordenada del array. Finalmente, se invoca la función recursiva divideVencerasPro (puntos, 0, n – 1)

2. **`private static Par divideVencerasPro(Punto[] puntos, int inicio, int fin):`** Como regla general, lo primero que se debe implementar es el caso base, que es similar a los anteriores: si hay tres o menos puntos, se comparan directamente entre sí. Luego, se realizan dos llamadas recursivas que dividen el problema en dos mitades del mismo tipo.

   Una vez resueltos los subproblemas de forma recursiva, se aplica la regla de los 7, lo que ha permitido reducir significativamente la complejidad del algoritmo en la fase de combinación.

   - Lo primero que se hace es determinar el mejor par entre ambas mitades, de esa manera determinar una franja de puntos cercanos al eje central.
   - Una vez hallado aquellos puntos que pertenecen a la franja se ordenan de nuevo con el método Arrays.sort(), pero esta vez haciendo uso de la coordenada Y.
   - En este punto se aplica la regla de los 7 puntos, limitando las comparaciones dentro de la franja, y así encontrando el mejor par.

Estudio Teórico - Análisis de complejidad del algoritmo

1. En el método público se hace un primera ordenación por el eje X, que al emplear un algoritmo sort, ya implicando una complejidad de O**(n log n)**.

2. Ya dentro del método privado, el enfoque de Divide y Vencerás divide una y otra vez el conjunto de puntos en dos mitades, este proceso ocurre **log n** veces, ya que n se divide en 2 en cada nivel de recursividad. Implicando de nuevo un orden de complejidad de **O(log n).**

3. La evaluación por fuerza bruto de los casos bases al ser ya demasiados pequeños aumentan la complejidad del algoritmo, limitándose a ser **O(1).**

4. Antes de recorrer la franja y obtener el mejor par, primero es necesario ordenándolo de nuevo, pero esta vez por la coordenada Y, llegando a la conclusión de un orden de complejidad **O(n log n)**

5. Finalmente, en la parte de la comparación en la flanja combinatorio, en el peor de los casos hará falta recorrer todo el array, pero gracias a la regla de los 7 puntos, cada punto solo se compara con un máximo de 7 otros puntos, limitado la cantidad de comparaciones a **O(n)** en la mayoría de los casos

Finalmente se obtiene como conclusión que el orden de complejidad es el siguiente:

$$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n\,log\,n)}$$

Finalmente, en este algoritmo si se ha conseguido reducir el orden de complejidad, pues, aunque en casos anteriores se ha podido mejorar la eficiencia, todavía no se había conseguido disminuir el orden de complejidad.

Análisis de eficiencia

- **Ventajas:** Este es el primer algoritmo que ha logrado reducir el orden de complejidad, convirtiéndose en el más eficiente hasta el momento. Esto se debe a la aplicación de la regla de los 7 puntos, que impacta directamente en la cantidad de comparaciones durante la fase de combinación, la cual era el principal factor de pérdida de eficiencia en los algoritmos anteriores.

- **Desventajas:** En algunos casos muy especiales, puede seguir alcanzándose **O(n^2)**, aunque con menor probabilidad que en versiones anteriores. Uno de esos casos es cuando hay distribución en una línea vertical, es decir, que los puntos tienen unas coordenadas X muy similares pero diferenciados en la coordenada Y. Y todo esto haciendo que la franja central pueda contener muchos puntos

4.7. Algoritmo Divide y Vencerás Ultra

Esta es la opción óptima para esta práctica, ya que aplica la regla de los 7 puntos sin necesidad de ordenar la franja. El detalle de su funcionamiento se explicará más adelante en el algoritmo. Su eficiencia radica en la reducción de comparaciones innecesarias al

construir la franja de manera óptima, manteniendo dos listas ordenadas: px por coordenada X y py por coordenada Y.

Estudio de implementación - Funcionamiento del algoritmo

1. **`divideVencerasUltra(Punto[] puntos):`** Lo primero que hace el método público del algoritmo es verificar que los datos del array no sean nulos y que contengan al menos dos puntos. En el segundo paso, a diferencia de los algoritmos anteriores, se crean dos arrays: uno de ellos contiene los puntos ordenados por la coordenada X y la coordenada Y, mientras que el segundo array se inicializa con la longitud del array que se pasa como parámetro al método público.

   Una vez que se ha ordenado el primer array se verifica que en dicho array haya puntos repetidos, haciendo uso del método **comprobarRepetidos()**, para que posteriormente se pueda copiar este array a otro. Por último, se hace la llamada recursiva al método privado.

2. **`divideVencerasUltra(Punto[] px, Punto[] py, Punto[] auxiliarPuntos, int inicio, int fin):`** este es el único algoritmo donde su caso base cambia, ya que en este caso solo se considera cuando el subproblema tiene un solo punto o está vacío, y se devuelve una distancia infinita.

   Una vez verificado el caso base empieza el caso recursivo en el cual se hace la división del problema con dos llamadas recursivas. El siguiente paso es mezclar dos arrays de puntos ordenados por la coordenada Y en uno solo, para ello se ha creado el método private **static void mix(Punto[] py, Punto[] auxiliarPuntos, int inicio, int mitad, int fin).**

   Una vez que se ha realizado la ordenación del segundo array mediante el método mix(), se seleccionan los puntos relevantes de la franja. Se filtran aquellos puntos que están a una distancia delta en X respecto a la línea central, almacenando los puntos relevantes en un array auxiliar, lo que reduce el número de comparaciones.

   Finalmente, se mejora la regla de los 7 puntos, ya que cada punto en la franja se compara únicamente con los siguientes puntos en el array auxiliar. Esto mantiene una estructura ordenada y evita ordenamientos innecesarios.

Análisis de complejidad

1. **Ordenación inicial. O(n log n):** antes de la recursión se realizan dos operaciones sobre un array, uno por la coordenada X y la coordenada Y, como cada ordenación se realiza con MergeSort, su complejidad es **O(n log n).**

2. **División del problema. O(log n):** una vez dentro del método recursivo, la división del array en dos mitades, la resolución recursiva de cada mitad y la

combinación de ambas mitades da como resultado un orden de complejidad de **O(log n).**

3. **Construcción de la franja. O(n):** una vez dividido el problema de manera recursiva, se identifican los puntos que están dentro de la franja filtrando los puntos por la coordenada X y que la distancia a la que se encuentren de la línea media. Este filtrado se hace de una sola pasada por lo que tiene un orden de complejidad de **O(n)**.

4. **Comparaciones dentro de la franja:** en esta parte es donde se aplica la regla de los 7 puntos, en la cual cada punto se compara solo con los siguiente en la franja que cumplen la condición en Y de la distancia, significando que cada apunto solo se compara con un máximo de 7 puntos. Dado que en cada nivel se tiene O(n) puntos en la franja, el orden de complejidad de esta parte es **O(n)**

La conclusión es que el orden de complejidad de este algoritmo es:

$$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n\ log\ n)}$$

Pero haciendo uso del Teorema Maestro, quedaría de la siguiente manera:

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + O(n^d)}$$

a = 2 → número de llamadas recursivas

b = 2 → el problema se divide en 2 mitades

c = 1 → la fase de combinación toma **O(n).**

Calculamos log2 (2) = 1, y como d = log b(a), estamos en el caso 2 del Teorema Maestro, lo que da una complejidad final de $\mathbf{O(n\ log\ n)}$

Análisis de eficiencia

- **Ventajas:** Mantiene py ordenado sin necesidad de ordenamientos adicionales y, al mismo tiempo, reduce el número de comparaciones innecesarias en la franja. Esto se logra mediante una aplicación eficiente de la regla de los 7 puntos.

- **Desventajas:** se trata de uno d ellos algoritmos más eficientes implementados, pero también se trata del más complejo de todos, y en casos muy desfavorables podría llegar a un orden de complejidad de **$O(n^2)$,** y pequeños grupos de puntos algoritmos más sencillos como fuerza bruta o divide y vencerás de versiones anteriores es más eficiente. Es ideal en grandes conjuntos de datos.

5. ESTUDIO EXPERIMENTAL

5.1. Metodología de experimentación

Para sacar conclusiones sobre el rendimiento de la implementación de nuestros algoritmos se han realizado una serie de pruebas para después analizar los resultados de tiempos obtenidos. Como conjunto de datos usados podemos diferenciar:

Conjuntos de datos reales

Estos ficheros contienen coordenadas que representan ubicaciones reales o realistas, lo que permite comprobar el comportamiento de los algoritmos en situaciones donde los datos no siguen patrones sintéticos.
Se hace uso de los siguientes archivos

- *Spain.txt* (8112 puntos)
- *NAclpoint.txt* (9203 puntos)
- *NApppoint.txt* (24493 puntos)
- *NArrpoint.txt* (191637 puntos)
- EDAland

Conjuntos de datos generados

Se ha implementado un generador de puntos aleatorios que proporciona valores según tres tipos de distribuciones estadísticas: gaussiana, uniforme y exponencial.

Para evaluar el desempeño del generador, se han creado conjuntos con tamaños crecientes, definidos por potencias de 2, desde n=128 hasta n=262144. Esta variación en la magnitud del conjunto permite analizar cómo evoluciona el tiempo de ejecución y verificar experimentalmente si los resultados coinciden con las complejidades teóricas esperadas.

5.2. Toma de medidas y análisis

Medidas

Para el conjunto de datos reales, cada uno de los algoritmos considerados (fuerza bruta, fuerza bruta mejorada y variantes del método Divide y vencerás) ha sido ejecutado múltiples veces sobre cada conjunto de datos generado.

Por otra parte, para el conjunto generado, el propio generador esta implementado para proporcionar un archivo .csv con las medidas para cada algoritmo en cada iteración de tamaño, facilitando así la recogida de los datos

Posteriormente, se ha calculado el promedio de los tiempos de ejecución para minimizar la variabilidad derivada de factores externos, como las interferencias del sistema operativo.

Finalmente, los resultados obtenidos se han expresado generalmente en nanosegundos, representando con precisión el tiempo requerido por cada algoritmo para encontrar el par de puntos más cercano.

Procesado de datos

Para el procesado y análisis de los datos se ha utilizado MATLAB como herramienta principal. Concretamente, se ha llevado a cabo:

- **Generación de gráficas** que representan la relación entre el tamaño del conjunto (n) y el tiempo medio de ejecución registrado.
- **Cálculo de ajustes** mediante modelos cuadráticos y $n \log n$, acompañados por la determinación del coeficiente R^2. Esto ha permitido evaluar de forma cuantitativa la correspondencia entre los resultados experimentales obtenidos y las curvas teóricas esperadas.

5.3. Resultados de la ejecución e interpretación

Los resultados indicaran claramente que los métodos basados en fuerza bruta (tanto el algoritmo simple como su versión mejorada) y las versiones de Divide y vencerás hasta su versión Pro presentaron un crecimiento cuadrático en el tiempo de ejecución, provocando un aumento exponencial en el tiempo requerido al procesar grandes volúmenes de datos.

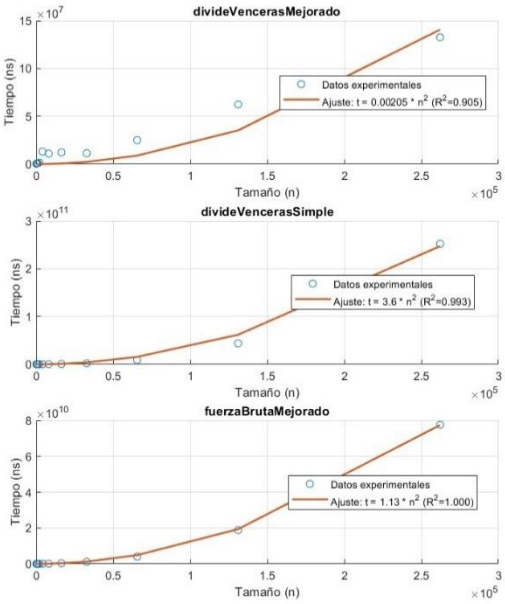
Por el contrario, las variantes de Divide y vencerás Pro y Ultra sí que demostraron un rendimiento superior, destacando especialmente la versión optimizada con franja central y la aplicación de la regla de los 7 puntos. Esta versión mostró un crecimiento temporal cercano a la complejidad teórica esperada, $O(n \log n)$

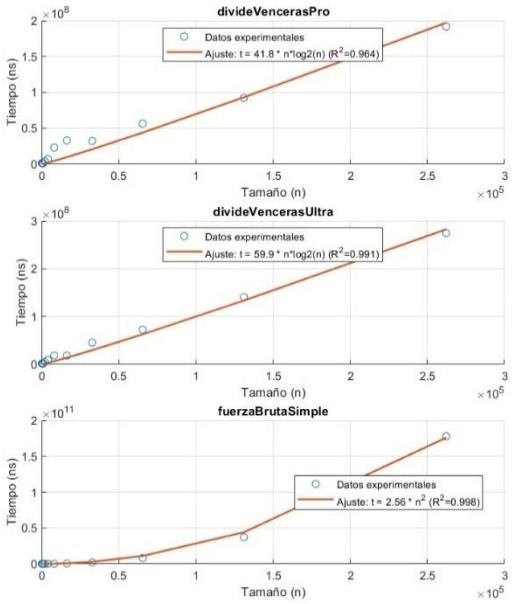
Se verificó además mediante los coeficientes de determinación $R^2 > 0.9$ que los ajustes realizados cuadrático para fuerza bruta y los Divide y vencerás teóricamente calculados, y $n \log n$ para los Divide y vencerás Pro y Ultra se adaptan satisfactoriamente a los datos experimentales obtenidos.

El conjunto EDAland, siendo de menor tamaño, ha permitido ilustrar adecuadamente un caso práctico real relacionado con el control aéreo, pero a la hora de probar el rendimiento no existe una diferencia significativa entre los algoritmos de fuerza bruta y los que aplican Divide y vencerás.

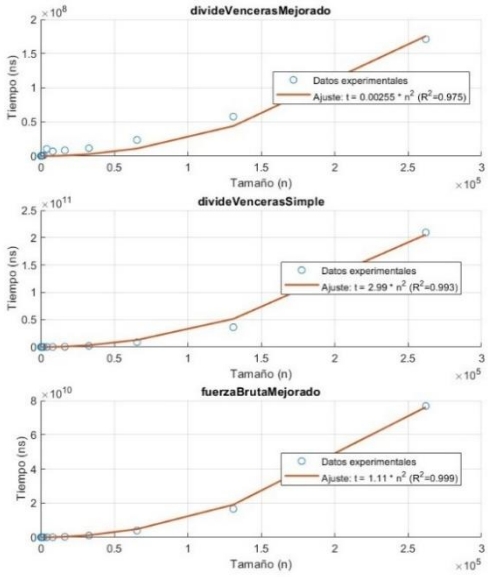
5.4. Gráficas para el análisis

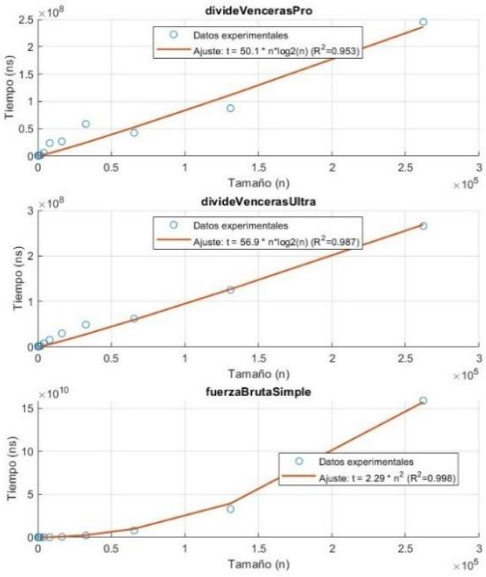
Distribución Gaussiana Generador



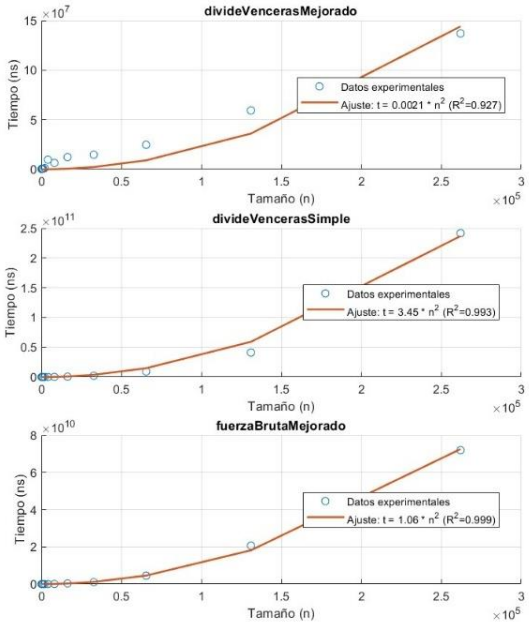


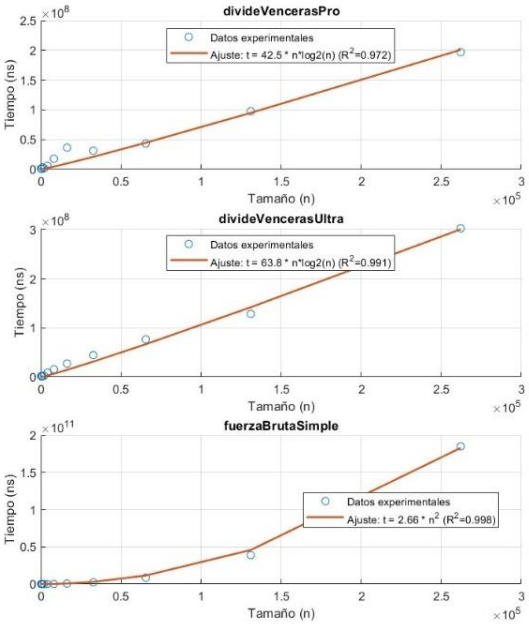
Distribución Uniforme Generador



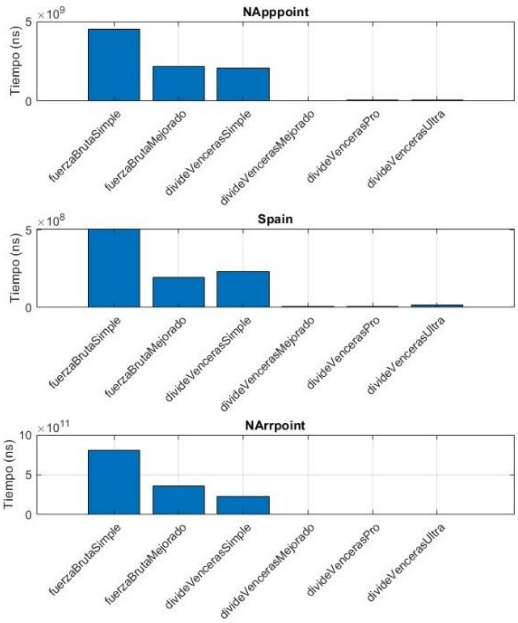


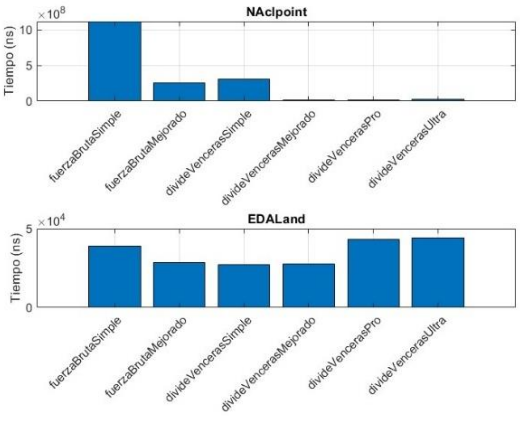
Distribución Exponencial Generador





Archivos proporcionados





6. ANEXOS

6.1. Pseudocódigo de los algoritmos

A continuación, las siguientes figuras presentan los pseudocódigos correspondientes a cada uno de los algoritmos implementados, comenzando con los algoritmos iterativos y avanzando hasta la última versión del enfoque de divide y vencerás. Estas representaciones muestran los diferentes pasos que han ido siguiendo.

Algoritmo de Fuerza Bruta Simple

Algorithm 1 Fuerza Bruta Simple

```
Entrada: array de puntos
Salida: par de puntos con la menor distancia
if puntos es nulo then
    Lanzar Excepción
end if
if longitud de puntos < 2 then
    Lanzar Excepción
end if
mejor ← Par vacío (con distancia infinita)
for i ← 0 hasta n − 1 do
    for j ← 0 hasta n − 1 do
        if i = j then
            Continuar
        end if
        d ← distancia entre puntos[i] y puntos[j]
        if d < mejor.distancia then
            mejor ← nuevo Par(puntos[i], puntos[j], d)
        end if
    end for
end for
return mejor
```

Algoritmo Fuerza Bruta Mejorado

Algorithm 1 Fuerza Bruta Mejorado

```
Entrada: array de puntos
Salida: par de puntos con la menor distancia
if puntos es nulo then
    Lanzar Excepción
end if
if longitud de puntos < 2 then
    Lanzar Excepción
end if
mejor ← Par vacío (con distancia infinita)
for i ← 0 hasta n − 2 do
    for j ← i + 1 hasta n − 1 do
        d ← distancia entre puntos[i] y puntos[j]
        if d < mejor.distancia then
            mejor ← nuevo Par(puntos[i], puntos[j], d)
        end if
    end for
end for
return mejor
```

Algoritmo Divide y Vencerás Simple

Algorithm 1 Divide y Vencerás Simple - Método Público

```
1: Entrada: array de puntos
2: Salida: par de puntos con la menor distancia
3: if puntos es nulo then
4:     Lanzar Excepción
5: end if
6: if longitud de puntos < 2 then
7:     Lanzar Excepción
8: end if
9: copiaPuntos ← copia ordenada de puntos por coordenadas (x, y)
10: mejor ← comprobarRepetidos(copiaPuntos)
11: if mejor ≠ nulo then
12:     return mejor
13: end if
14: return divideVencerasSimple(copiaPuntos, 0, longitud de copiaPuntos
    - 1)
```

Algorithm 2 Divide y Vencerás Simple - Método Privado

```
1: Entrada: array de puntos, índice de inicio, índice de fin
2: Salida: par de puntos con la menor distancia
3: mejor ← Par vacío (con distancia infinita)
4: if fin - inicio ≤ 3 then
5:     for i ← inicio hasta fin - 1 do
6:         for j ← i + 1 hasta fin do
7:             d ← distancia entre puntos[i] y puntos[j]
8:             if d < mejor.distancia then
9:                 mejor ← nuevo Par(puntos[i], puntos[j], d)
10:            end if
11:        end for
12:    end for
13:    return mejor
14: end if
15: mitad ← (inicio + fin) / 2
16: izquierda ← divideVencerasSimple(puntos, inicio, mitad)
17: derecha ← divideVencerasSimple(puntos, mitad + 1, fin)
18: mejor ← el menor entre izquierda y derecha
19: for i ← inicio hasta mitad do
20:     for j ← mitad + 1 hasta fin do
21:         d ← distancia entre puntos[i] y puntos[j]
22:         if d < mejor.distancia then
23:             mejor ← nuevo Par(puntos[i], puntos[j], d)
24:         end if
25:     end for
26: end for
27: return mejor
```

Algoritmo Divide y Vencerás Mejorado

Algorithm 1 Divide y Vencerás Mejorado - Método Público

```
1: Entrada: array de puntos
2: Salida: par de puntos con la menor distancia
3: if puntos es nulo then
4:     Lanzar Excepción
5: end if
6: if longitud de puntos < 2 then
7:     Lanzar Excepción
8: end if
9: copiaPuntos ← copia ordenada de puntos por coordenadas (x, y)
10: mejor ← comprobarRepetidos(copiaPuntos)
11: if mejor ≠ nulo then
12:     return mejor
13: end if
14: return divideVencerasMejorado(copiaPuntos, 0, longitud de copiaPuntos - 1)
```

Algorithm 2 Divide y Vencerás Mejorado Recursivo - Método Privado

```
1: Entrada: array de puntos, índice de inicio, índice de fin
2: Salida: par de puntos con la menor distancia
3: mejor ← Par vacío (con distancia infinita)
4: if fin - inicio ≤ 3 then
5:     for i ← inicio hasta fin - 1 do
6:         for j ← i + 1 hasta fin do
7:             d ← distancia entre puntos[i] y puntos[j]
8:             if d < mejor.distancia then
9:                 mejor ← nuevo Par(puntos[i], puntos[j], d)
10:            end if
11:        end for
12:    end for
13:    return mejor
14: end if
15: mitad ← (inicio + fin) / 2
16: izquierda ← divideVencerasMejorado(puntos, inicio, mitad)
17: derecha ← divideVencerasMejorado(puntos, mitad + 1, fin)
18: mejor ← el menor entre izquierda y derecha
19: for i ← mitad hasta inicio y puntos[i].x > puntos[mitad].x − mejor.distancia do
20:     for j ← mitad + 1 hasta fin y puntos[j].x < puntos[mitad].x + mejor.distancia do
21:         d ← distancia entre puntos[i] y puntos[j]
22:         if d < mejor.distancia then
23:             mejor ← nuevo Par(puntos[i], puntos[j], d)
24:         end if
25:     end for
26: end for
27: return mejor
```

Algoritmo Divide y Vencerás Pro

Algorithm 1 Divide y Vencerás Pro - Método Público

```
1: Entrada: array de puntos
2: Salida: par de puntos con la menor distancia
3: if puntos es nulo then
4:     Lanzar Excepción
5: end if
6: if longitud de puntos < 2 then
7:     Lanzar Excepción
8: end if
9: copiaPuntos ← copia ordenada de puntos por coordenadas (x, y)
10: mejor ← comprobarRepetidos(copiaPuntos)
11: if mejor ≠ nulo then
12:     return mejor
13: end if
14: return divideVencerasMejorado(copiaPuntos, 0, longitud de copiaPuntos - 1)
```

Algorithm 2 Divide y Vencerás Pro - Método Privado

```
1: Entrada: array de puntos, índice de inicio, índice de fin
2: Salida: par de puntos con la menor distancia
3: mejor ← Par vacío (con distancia infinita)
4: if fin - inicio ≤ 3 then
5:     for i ← inicio hasta fin - 1 do
6:         for j ← i + 1 hasta fin do
7:             d ← distancia entre puntos[i] y puntos[j]
8:             if d < mejor.distancia then
9:                 mejor ← nuevo Par(puntos[i], puntos[j], d)
10:            end if
11:        end for
12:    end for
13:    return mejor
14: end if
15: mitad ← (inicio + fin) / 2
16: izquierda ← divideVencerasMejorado(puntos, inicio, mitad)
17: derecha ← divideVencerasMejorado(puntos, mitad + 1, fin)
18: mejor ← el menor entre izquierda y derecha
19: for i ← mitad hasta inicio y puntos[i].x > puntos[mitad].x − mejor.distancia do
20:     for j ← mitad + 1 hasta fin y puntos[j].x < puntos[mitad].x + mejor.distancia do
21:         d ← distancia entre puntos[i] y puntos[j]
22:         if d < mejor.distancia then
23:             mejor ← nuevo Par(puntos[i], puntos[j], d)
24:         end if
25:     end for
26: end for
27: return mejor
```

Algoritmo Divide y Vencerás Ultra

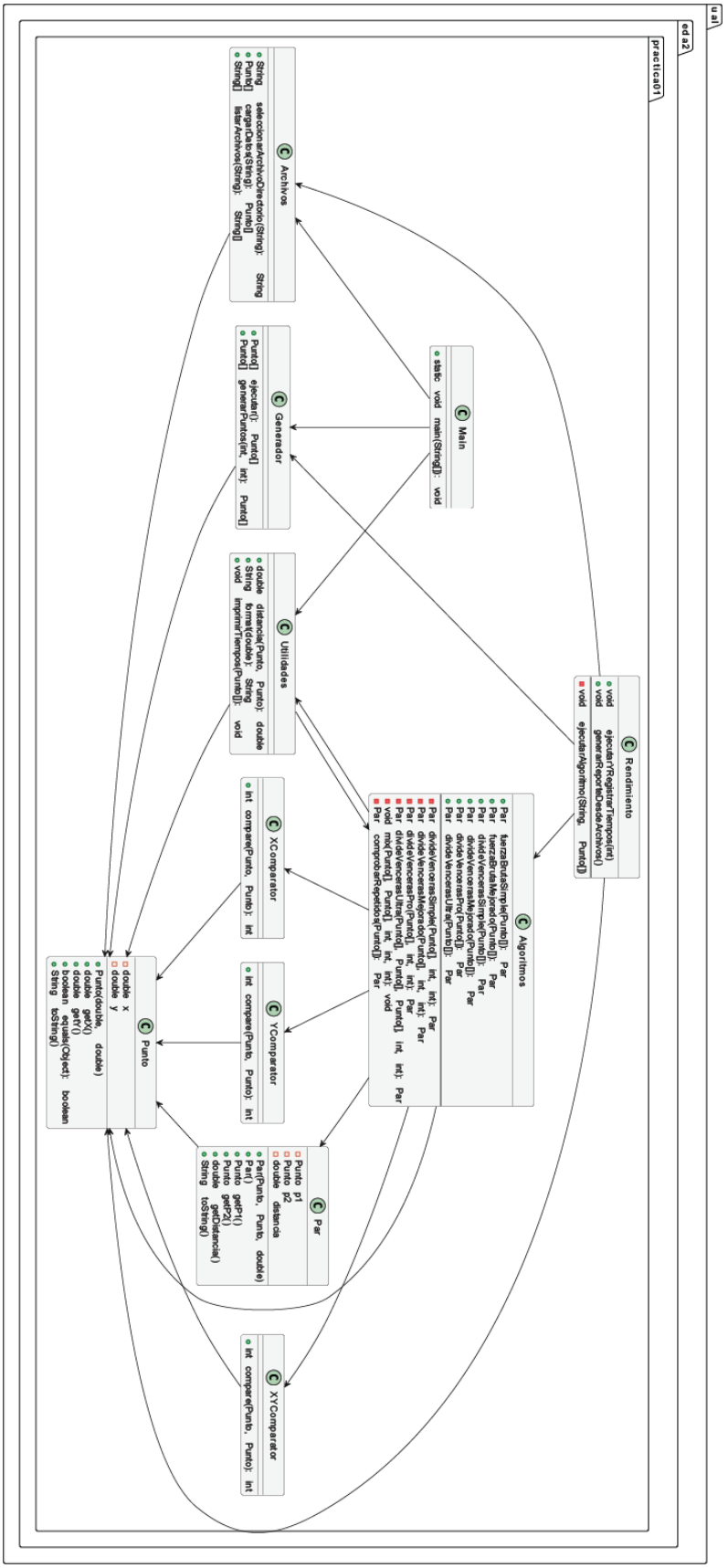
Algorithm 1 Divide y Vencerás Ultra - Método Público

```
1: Entrada: array de puntos
2: Salida: par de puntos con la menor distancia
3: if puntos es nulo then
4:     Lanzar Excepción
5: end if
6: if longitud de puntos < 2 then
7:     Lanzar Excepción
8: end if
9: px ← copia de puntos ordenado por x
10: py ← copia de puntos ordenado por y
11: mejor ← comprobarRepetidos(px)
12: if mejor ≠ nulo then
13:     return mejor
14: end if
15: return divideVencerasUltra(px, py, auxiliarPuntos, 0, longitud de px -
    1)
```

Algorithm 2 Divide y Vencerás Ultra - Método Privado

```
1: Entrada: arrays de puntos px, py, auxiliarPuntos, índices inicio, fin
2: Salida: par de puntos con la menor distancia
3: mejor ← Par vacío (con distancia infinita)
4: if inicio ≥ fin then
5:     return mejor
6: end if
7: mitad ← (inicio + fin)/2
8: izquierda ← divideVencerasUltra(px, py, auxiliarPuntos, inicio, mitad)
9: derecha ← divideVencerasUltra(px, py, auxiliarPuntos, mitad + 1, fin)
10: mejor ← el menor entre izquierda y derecha
11: mezclar(py, auxiliarPuntos, inicio, mitad, fin)
12: k ← 0
13: for i ← inicio hasta fin do
14:     if |py[i].getX() − px[mitad].getX()| ¡ mejor.distancia then
15:         auxiliarPuntos[k] ← py[i]
16:         k + +
17:     end if
18: end for
19: for i ← 0 hasta k − 2 do
20:     for j ← i + 1 hasta k − 1 y py[j].getY() − py[i].getY() < mejor.distancia
    do
21:         d ← distancia entre auxiliarPuntos[i] y auxiliarPuntos[j]
22:         if d < mejor.distancia then
23:             mejor ← nuevo Par(auxiliarPuntos[i], auxiliarPuntos[j], d)
24:         end if
25:     end for
26: end for
27: return mejor
```

6.2. Diagramas de clases



7. BIBLIOGRAFÍA Y REFERENCIAS

Frias, S. (2021, abril 4). *Significado del algoritmo divide y vencerás: Explicado con ejemplos*. freecodecamp.org. https://www.freecodecamp.org/espanol/news/significado-del-algoritmo-divide-y-venceras/

Improve, K. F. (2012, noviembre 28). *Closest Pair of Points using Divide and Conquer algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/

Wu, S.-Y. (fiona). (2021, julio 22). *Algorithms StudyNote — 4: Divide and conquer — closest pair*. Medium. https://medium.com/@shihyu-wu/algorithms-studynote-4-divide-and-conquer-closest-pair-49ba679ce3c7