Práctica 1: Estrategias Algorítmicas

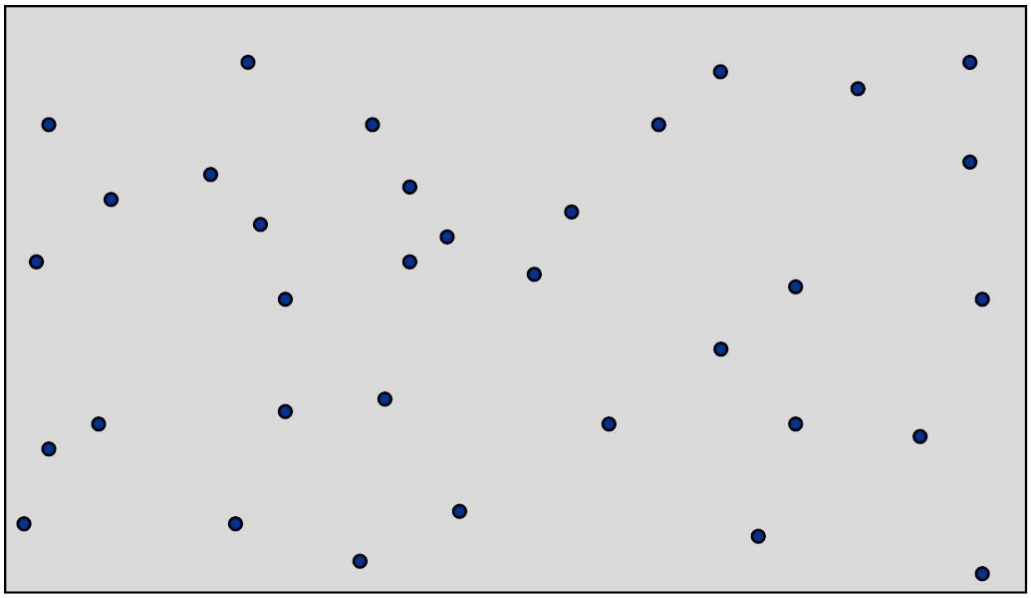
Objetivo.

El objetivo de esta primera parte de la práctica 1 es desarrollar algoritmos para resolver un problema clásico de búsqueda sobre conjuntos de puntos: la búsqueda del punto más cercano a otro. Para ello se plantearán estrategias de búsqueda exhaustiva, búsqueda con poda y Divide y Vencerás.

Parte 1 Análisis de algoritmos exhaustivos y Divide y Vencerás

El Problema del punto más cercano a otro.

Dado un conjunto de puntos situados en un plano $P = \{ (x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n) \}$ este problema consiste en encontrar (teniendo en cuenta que la distancia entre dos puntos i y j es $sqrt[(x_i-x_j)^2+(y_i-y_j)^2]$) el **punto** $p_i = (x_i,y_i)$ **más cercano a otro** $p_j = (x_j,y_j)$ tal que distancia ((x_i,y_i), (x_j,y_j)) sea mínima, es decir, se debe encontrar de entre todos los puntos posibles cual es el que está más cerca de otro punto (encontrar la pareja de puntos con la menor distancia euclídea entre ellos).



Primera Aproximación: Estrategia de búsqueda exhaustiva

Una primera solución es realizar una búsqueda exhaustiva analizando todos los pares de puntos posibles y quedarse con el más pequeño. Para n puntos, existen n·(n-1)/2 pares de puntos posibles, por lo que el tiempo de ejecución es de **$O(n^2)$**.

El programa resultante es corto y muy rápido para casos pequeños, pero a medida que aumenta el tamaño del conjunto de puntos el tiempo de ejecución va creciendo de forma exponencial, por lo que se hace necesario encontrar un algoritmo más rápido. ¿Cuál?

Segunda Aproximación: Estrategia de búsqueda con poda

En la búsqueda exhaustiva, se prueban combinaciones de puntos que podemos descartar por lo lejos que están unos de otros. Por ello, en una segunda aproximación lo que haremos es ordenar los puntos por una determinada coordenada (x o y) de manera que volveremos a realizar una búsqueda exhaustiva con poda, de manera que cada iteración la abortaremos en el momento en el que la distancia respecto a la coordenada por la que hemos ordenado sea mayor que la distancia mínima que tenemos calculada.

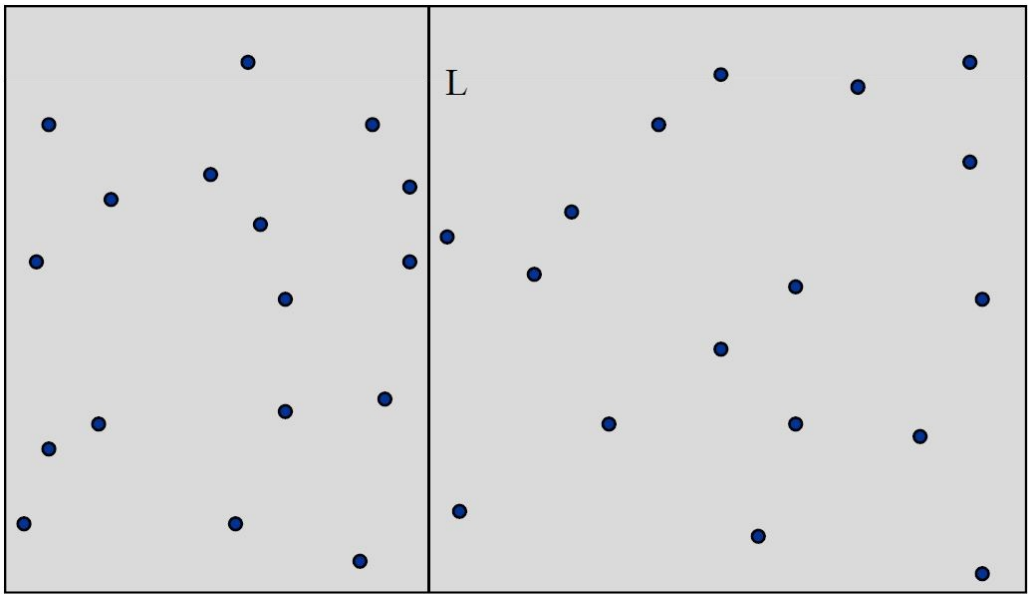
Supongamos que ordenamos los puntos según la coordenada x; esto supondría un tiempo de **O(n·log n)** si empleamos un algoritmo de ordenación rápida, por lo que el algoritmo tarda como mínimo eso (es una cota inferior para el algoritmo completo). Ahora que se tiene el conjunto ordenado, se puede modificar el algoritmo exhaustivo anterior de manera que seguiremos iterando mientras que la distancia entre los ejes x de los puntos que estamos comparando sea menor que la distancia mínima calculada hasta ese momento.

Dados n puntos, en cada iteración i compararemos el **punto** $p_i = (x_i,y_i)$) $1 < i \leq n$ con todos los demás **puntos** $p_j = (x_j,y_j)$ $i < j \leq n$ mientras que la distancia ($(x_i,0)$, $(x_j,0)$) sea menor a la distancia mínima calculada hasta ese momento. En el momento que encontremos un punto $p_j = (x_j,y_j)$ $i < j \leq n$ que no cumpla la condición anterior podremos descartar todos los puntos p_k $k > j$ con el consiguiente ahorro de comparaciones.

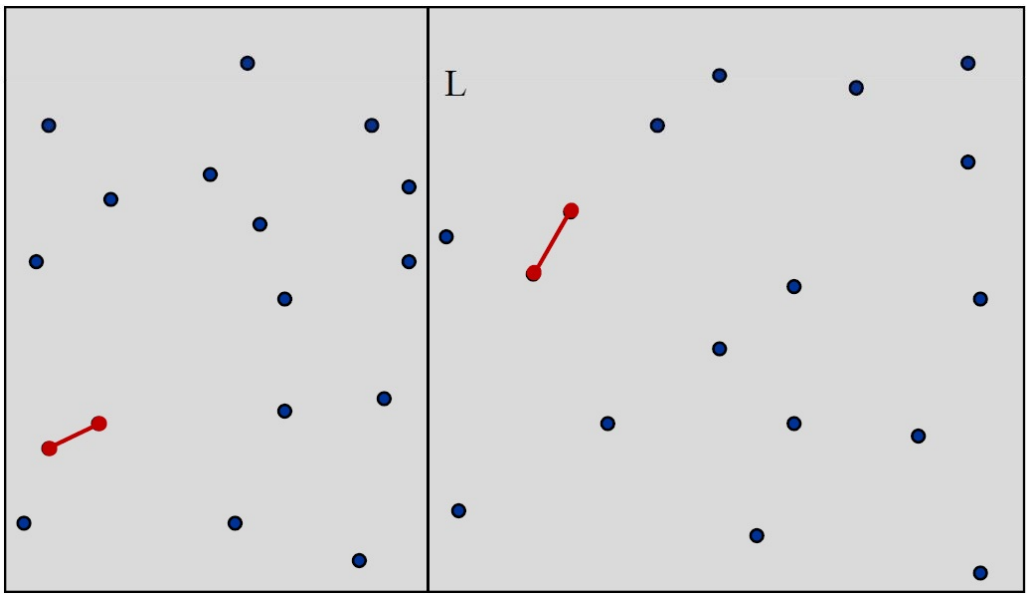
Tercera Aproximación: Estrategia de búsqueda con técnica Divide y Vencerás

En esta tercera aproximación aplicaremos la técnica **Divide y Vencerás** para lo cual es necesario, al igual que en la segunda aproximación ordenar los puntos por una determinada coordenada (x o y), por lo que el algoritmo tarda como mínimo un tiempo de **O(n·log n)** al igual que antes (es una cota inferior para el algoritmo completo).

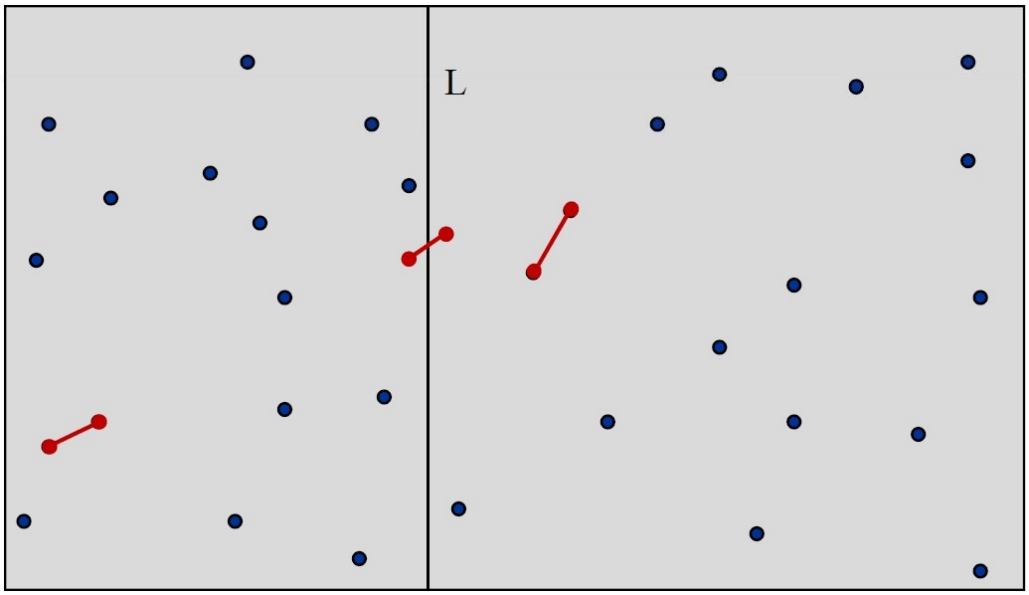
Supongamos que lo ordenamos por la coordenada x. Ahora que se tiene el conjunto P ordenado, se puede trazar una línea vertical, $x = x_m$, con n/2 puntos a cada lado: P_i y P_d.



Ahora, o el par de puntos más cercanos p_s está en P_i, o está en P_d, …

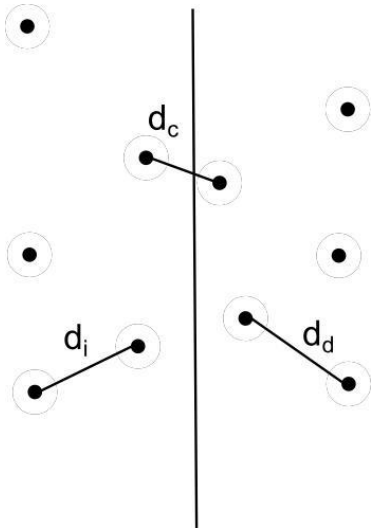


…pero puede que uno de los puntos del par está en P_i y el otro en P_d.



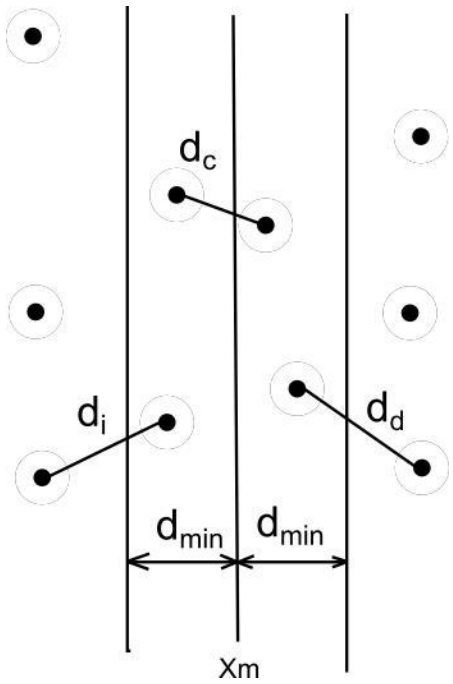
Si los dos estuvieran en P_i o en P_d, se hallaría recursivamente, subdividiendo más el problema. El caso más complejo, por tanto, se reduce al tercer caso, cuando el par mínimo tiene un punto en cada zona.

La siguiente figura ilustra los 3 casos antes comentados:



El par de puntos más cercano puede estar en la zona izquierda (zona P_i), en la derecha (zona P_d), o tener un punto en cada zona (uno en la zona P_i y el otro en la zona P_d).

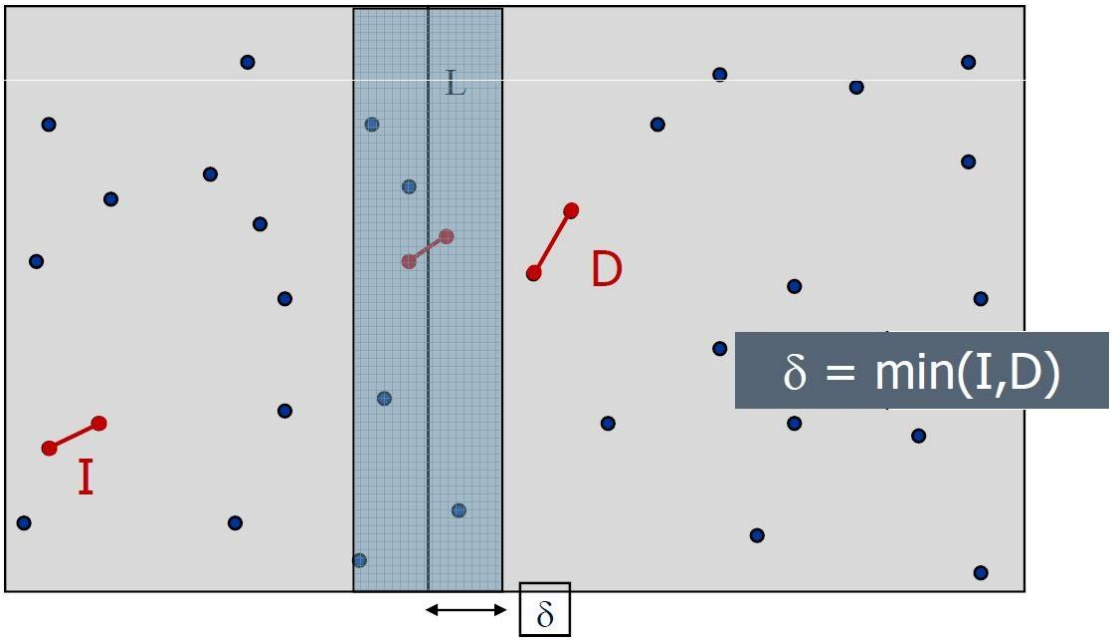
Llamemos d_i, d_d y d_c a las mínimas distancias en el primer caso, en el segundo, y en el tercero, respectivamente, y d_{min} al menor de d_i y d_d. Para resolver el tercer caso, sólo hace falta mirar los puntos cuya coordenada x esté entre x_m-d_{min} y x_m+d_{min}. Para grandes conjuntos de puntos distribuidos uniformemente, el número de puntos que caen en esa franja es sqrt(n), así que con una búsqueda exhaustiva el tiempo de ejecución sería de **O(n)**, y tendríamos el problema resuelto. El tiempo de ejecución sería, según lo dicho en el otro apartado, **O(n·log n)**.



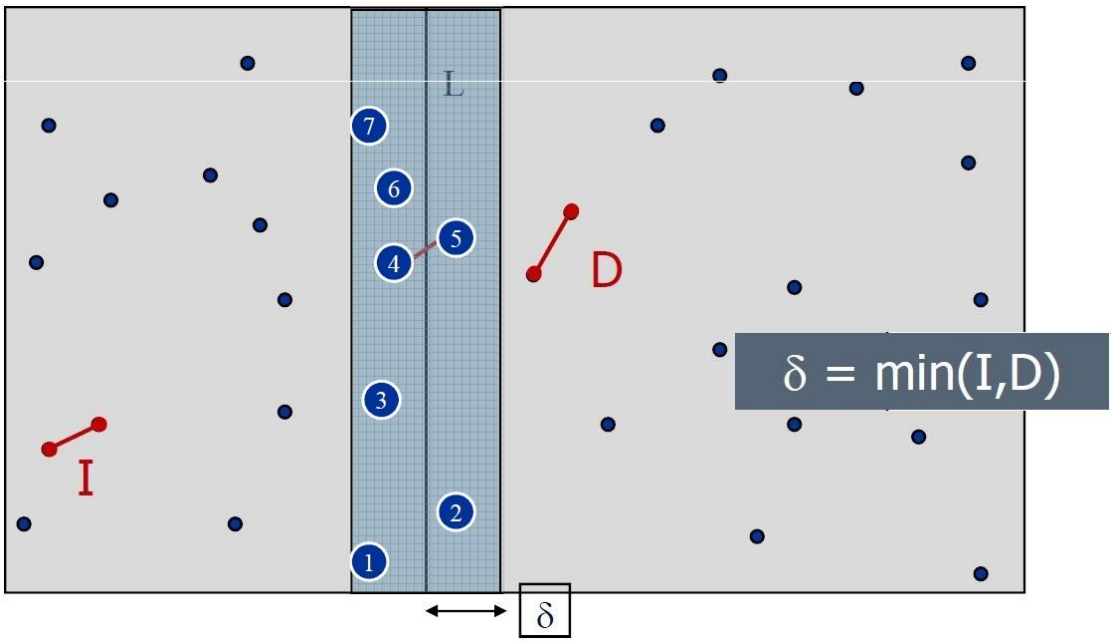
Cuarta Aproximación: Estrategia de búsqueda con técnica Divide y Vencerás con mejora en la búsqueda en la zona intermedia

Si los puntos no están uniformemente distribuidos, el algoritmo Divide y Vencerás anterior deja de ser tan eficiente, ya que en el peor de los casos (todos los puntos tienen la misma coordenada x, es decir, están en la misma vertical), todos los puntos quedan dentro de la franja, por lo que el realizar una búsqueda exhaustiva dentro de la franja equivale a la primera aproximación (ya que todos los puntos quedan dentro de ella), lo que supone un tiempo de ejecución de **O(n^2)**.

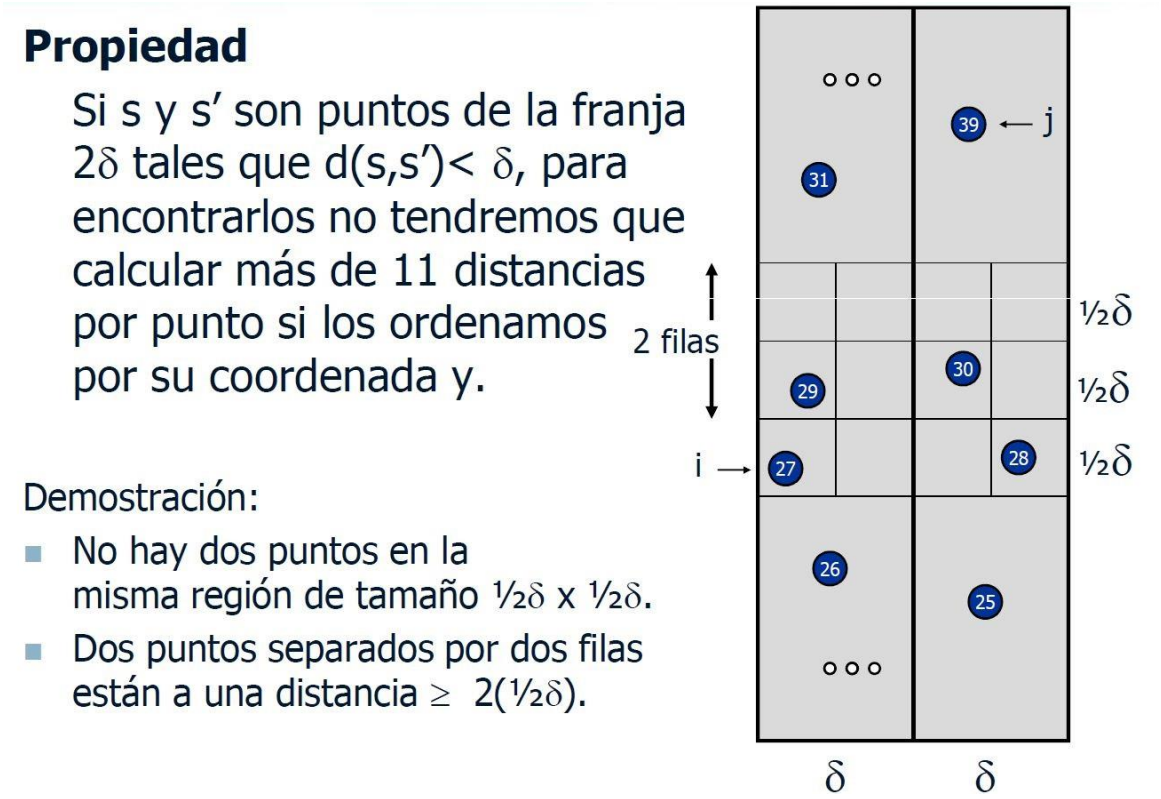
Para resolver este (peor) caso, en esta cuarta aproximación aplicaremos la técnica **Divide y Vencerás** anterior, pero ordenando los puntos de la franja por la coordenada y (en lugar de la x), lo que supone un tiempo de ejecución de **O(n·log n)**.



Una vez tenemos delimitada la franja 2δ, ordenamos los elementos de la franja por la coordenada y (en lugar de la x) y sólo comprobamos aquéllos que están a menos de 12 posiciones en la lista ordenada por la coordenada y



La razón por la que sólo debemos comprobar aquéllos que están a menos de 12 posiciones en la lista ordenada en función de y se ilustra en la siguiente figura:



Al ser la distancia mínima entre los puntos igual a δ matemáticamente sabemos que no puede haber 2 puntos en el mismo cuadrado de tamaño ½δ x ½δ (verticalmente y horizontalmente hay una distancia ½δ y la máxima distancia en el cuadrado es su diagonal que por Pitágoras es $((½\delta)^2 + (½\delta)^2)^{½} = 0{,}70\ \delta$).

2 puntos separados por 2 filas están a una distancia ≥ 2(½δ), es decir, a una distancia ≥ δ (debe haber una separación de 2 filas entre ambos puntos, ya que uno de los puntos puede estar en la frontera superior de la fila i y el otro puede estar en la frontera inferior de la fila i+3: de la frontera superior de la fila i a la inferior de la fila i+3 hay como mínimo 2(½δ) = δ, que es la anchura (altura) de 2 cuadrados de tamaño ½δ x ½δ.

En la figura de ejemplo, el punto P_{27} sólo puede tener como máximo 11 puntos con una coordenada y mayor que el suyo (los puntos P_{26} y P_{25} tienen una coordenada y inferior, los puntos P_{31} y P_{39} tienen una coordenada y ≥ y+2(½δ), es decir, una coordenada y ≥ y + δ con lo cual matemáticamente se pueden descartar). Esos 11 puntos sólo pueden encontrarse en los 11 cuadrados de tamaño ½δ x ½δ que hay a su alrededor. No puede haber más de 11 puntos.

Esto se cumple para cualquier punto P_i de la franja.

Ficheros de entrada

Para ejecutar los algoritmos propuestos se podrá utilizar tanto grupos de puntos aleatorios como grupos de puntos leídos de un fichero externo.

El programa deberá permitir cargar cualquier fichero de la biblioteca TSPLIB (http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/) del repositorio TSP para problemas simétricos: *Symmetric traveling salesman problem* (TSP) (http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/), los cuales presentan el mismo formato, una lista con dos valores para cada ciudad que representan sus coordenadas en el plano.

A continuación, se ofrece un ejemplo del formato de estos ficheros (en concreto, el fichero *burma14.tsp*).

```
NAME: burma14
TYPE: TSP
COMMENT: 14-Staedte in Burma (Zaw Win)
DIMENSION: 14
EDGE_WEIGHT_TYPE: GEO
EDGE_WEIGHT_FORMAT: FUNCTION
DISPLAY_DATA_TYPE: COORD_DISPLAY
NODE_COORD_SECTION
   1  16.47       96.10
   2  16.47       94.44
   3  20.09       92.54
   4  22.39       93.37
   5  25.23       97.24
   6  22.00       96.05
   7  20.47       97.02
   8  17.20       96.29
   9  16.30       97.38
  10  14.05       98.12
  11  16.53       97.38
  12  21.52       95.59
  13  19.41       97.13
  14  20.09       94.55
EOF
```

Nota:
Las líneas importantes de este fichero son aquella que empieza por la cadena DIMENSION (ya que dicha línea termina con un entero que indica cuantas coordenadas de puntos hay almacenadas) y los tríos de números (int double double) que aparecen tras la línea NODE_COORD_SECTION.
Por tanto, el método que lee el fichero debe coger el entero que hay al final de la línea DIMENSION y los tríos de números (tantos como indique el entero anterior) que hay tras la línea NODE_COORD_SECTION

Desarrollo de la práctica

El alumno deberá desarrollar un proyecto en lenguaje JAVA que **resuelva el problema de encontrar la ciudad más cercana a otra entre un conjunto de ciudades, dadas sus coordenadas en el mapa** (las 2 ciudades más cercanas entre sí) **mediante los algoritmos de búsqueda explicados anteriormente.**

El programa desarrollado deberá permitir seleccionar como entrada un conjunto de datos generados de forma aleatoria, o bien un conjunto de datos importados mediante la carga de cualquier fichero de la biblioteca TSPLIB. Para ello el programa debe proporcionar una opción de menú o una ventana de diálogo que permita al usuario indicar la ruta del fichero que desea abrir.

El resultado del proceso de búsqueda se debe mostrar en modo texto (dos campos que indiquen cuales son los puntos encontrados: el par de puntos más cercanos) **y en modo gráfico** (por medio de un panel en el que se dibujen los puntos del dataset y la línea que delimita la distancia de los 2 puntos más cercanos entre sí).

Para desarrollar el algoritmo de ordenación de los puntos se deberá implementar o bien el algoritmo ***HeapSort o*** el algoritmo ***QuickSort*****.**

El alumno deberá entregar un estudio teórico realizado sobre el pseudocódigo simplificado de dichos algoritmos donde se calcule su complejidad (tiempo de ejecución calculando el número de operaciones elementales o mediante la elección de una operación básica) y su orden para el mejor, caso medio y peor caso de manera razonada. Además, se debe mostrar gráficamente algún ejemplo límite (mejor y peor caso) y explicar de manera razonada qué estrategia funcionará mejor.

Finalmente se pide comparar los resultados del estudio experimental de forma gráfica (además de presentarla de forma tabular, en una tabla). Para ello se elegirán grupos aleatorios (el mismo grupo para cada algoritmo) de (como mínimo) los siguientes tamaños: 200, 500, 1500, 5000. Se comparará la ejecución de estos tamaños en las 4 estrategias (exhaustiva, exhaustiva con poda, divide y vencerás, divide y vencerás mejorado). Si se considera útil para la representación gráfica se pueden representar tamaños adicionales o bien otros tamaños diferentes (1000, 2000, …, 5000 por ejemplo). Las representaciones gráficas del estudio experimental se superpondrán en una sola gráfica de líneas (4 líneas, una por cada estrategia).

Nota: El ejecutable de la práctica debe realizar el estudio experimental y mostrar los resultados en pantalla en modo texto. La representación gráfica de dicho estudio experimental en el ejecutable es opcional (sirve para subir nota), pero en la documentación es OBLIGATORIA. Aquellos alumnos que no implemente esta funcionalidad en el ejecutable pueden utilizar cualquier programa ajeno (Excel, gnuplot, etc.) para generar la gráfica y copiarla en la documentación.

Especificaciones técnicas del programa a realizar:

El programa debe contener como mínimo las siguientes funcionalidades mediante opciones de menú (en caso de que se haga en modo texto) o mediante botones de comandos (en caso de que se haga en modo gráfico):

1) Crear un fichero .tsp aleatorio

   Permite crear un fichero en formato .tsp con datos aleatorios. Debe preguntar cuántos datos generar y guardarlo en disco con el nombre datasetN.tsp, siendo N el tamaño del conjunto de datos creado. **Las coordenadas x, y generadas deben guardarse con una precisión de 10 cifras decimales.**
   Ejemplo: si creamos un fichero aleatorio de 567 puntos el fichero debe llamarse dataset567.tsp

2) Cargar un dataset en memoria

   Esta opción debe permitir cargar el dataset con un conjunto de datos generados de forma aleatoria (en ese caso el dataset creado debe guardarse en disco en formato .tsp con el nombre indicado en el apartado 1 anterior) o con los datos leídos de un fichero .tsp ya existente.

3) Comprobar Estrategias

   Esta opción debe mostrar en pantalla el resultado de aplicar las 4 estrategias sobre el dataset cargado en memoria, con el siguiente formato (se muestra los datos para el dataset ch130.tsp):

ch130.tsp

Estrategia	Punto1	Punto2	distancia	calculadas	tiempo(mseg)
Exhaustivo	12 (252.749, 535.743)	87 (252.429, 535.166)	0.66018099	8386	0.1679
ExhaustivoPoda	87 (252.429, 535.166)	12 (252.749, 535.743)	0.66018099	49	0.0317
DivideVenceras	87 (252.429, 535.166)	12 (252.749, 535.743)	0.66018099	657	0.0675
DyV Mejorado	87 (252.429, 535.166)	12 (252.749, 535.743)	0.66018099	262	0.0669

   La distancia mínima debe mostrarse con 8 cifras decimales, el tiempo con 4 y los puntos con el formato: id del punto (el que tiene en el fichero .tsp) y las coordenadas entre paréntesis.
   La columna distancia contabiliza el número de distancias euclídeas que cada algoritmo ha tenido que calcular para hallar la solución (para calcular los puntos de la franja intermedia en Divide y Vencerás y para descartar puntos en la exhaustiva con poda no hay que calcular distancias euclídeas sino sólo restar coordenadas x entre sí. Por tanto, esas no se contabilizan).

   El programa debe generar en disco, para cada estrategia, un fichero .tsp con el contenido del dataset resultante una vez terminada de aplicar la estrategia correspondiente (el nombre del fichero debe ser el nombre de la estrategia aplicada). Al aplicar 4 estrategias el programa debe crear 4 ficheros llamados Exhaustivo.tsp, ExhaustivoPoda.tsp, DivideVenceras.tsp y DyVMejorado.tsp (el primero debe ser idéntico al .tsp original y los otros 3 deben ser iguales entre sí y con los Puntos ordenados por el eje x).

   Importante:
   Las 4 estrategias deben aplicarse sobre el mismo dataset original.
   Debe tenerse en cuenta que, a excepción de la primera estrategia (exahustivo), el resto de ellas (exhaustivo con poda, divide y vencerás, divide y vencerás mejorado) requiere ordenar el conjunto de datos por lo que, tras aplicar cualquiera de ellas, el dataset ya no es el original sino el ordenado, con lo que las “condiciones iniciales” no son las mismas para cada experimento.

   Para garantizar las mismas “condiciones iniciales” debemos realizar cada experimento con una copia del dataset original (para la última de las estrategias podemos usar el dataset original y así evitamos tener que hacer una última copia)

4) Comparar todas las estrategias

Esta opción debe permitir comparar el rendimiento de cada algoritmo con diferentes tamaños de datos. El programa debe, para cada tamaño de datos correspondiente, generar un mínimo de 10 dataset diferentes (experimentos) y calcular el tiempo medio que tarda en calcular la solución con cada estrategia.

Ejemplo: salida generada para tamaño de datos entre 1000 y 5000 con incrementos de 1000

	Exhaustivo	ExhaustivoPoda	DivideVenceras	DyV Mejorado
Talla	Tiempo(mseg)	Tiempo(mseg)	Tiempo(mseg)	Tiempo(mseg)
1000	9.237816	0.244029	0.543403	0.666994
2000	38.679922	0.569996	1.384236	1.532789
3000	84.317963	0.889868	2.109434	2.188130
4000	151.332264	1.224417	3.077690	3.263193
5000	236.353400	1.531153	4.071083	4.110759

Importante:
Al igual que en apartado anterior, las 4 estrategias deben aplicarse sobre los mismos dataset.
Como por cada talla vamos a generar varios dataset diferentes (experimentos) debemos garantizar que cada estrategia trabaja con los mismos datos (si cada estrategia usa un dataset generado de forma aleatoria diferente los datos obtenidos no serían válidos ya que para comparar un algoritmo con otro lo debemos hacer con los mismos datos, esto es, mismas "condiciones iniciales").
Por tanto, el algoritmo que muestra los resultados en pantallas debe seguir el siguiente esquema:

```
para cada talla T hacer
   para cada experimento EXP hacer
      generar un dataset aleatorio de tamaño talla
      para cada estrategia E hacer
         generar una copia del dataset
         tiempo[E] = tiempo [E] + tiempo que tarda estrategia E con esa copia
      fin para
   fin para
   mostrar en pantalla los tiempos medios de cada estrategia para esa talla concreta
fin para
```

5) Comparar 2 estrategias

Esta opción debe permitir comparar el rendimiento de 2 algoritmos entre sí, con diferentes tamaños de datos. Es idéntica a la funcionalidad anterior con la diferencia de que además del tiempo debe mostrar las distancias calculadas.

Ejemplo: salida generada para tamaño de datos entre 1000 y 5000 (Exhaustivo vs DivideVenceras)

	Exhaustivo	DivideVenceras	Exhaustivo	DivideVenceras
Talla	Tiempo(mseg)	Tiempo(mseg)	Distancias	Distancias
1000	9.191101	0.538533	499501	11200
2000	37.696322	1.334017	1999001	32640
3000	85.453171	2.133985	4498501	53380
4000	152.218202	3.184242	7998001	82718
5000	238.635871	4.125989	12497501	106022

Consideraciones a tener en cuenta:

El rendimiento de las diferentes estrategias depende de la particularidad de los datos utilizados. En condiciones normales (caso medio) los datos aleatorios generados deben estar distribuidos a lo largo del eje x e y, y el algoritmo con mejor rendimiento debe ser el exhaustivo con poda (ya que es tan bueno como los divide y vencerás y computacionalmente tiene que realizar menos cálculos).

Pero en el caso hipotético de que los datos estén generados de forma que todos los puntos estén sobre la misma vertical (peor caso) el rendimiento de las diferentes estrategias varía considerablemente, resultando ser el mejor algoritmo el divide y vencerás mejorado y los peores el exhaustivo con poda y el divide y vencerás (el exhaustivo llega a ser, sorprendentemente, el segundo mejor algoritmo).

Al estar todos los puntos en la misma vertical (todos tienen el mismo valor x) el exhaustivo con poda no puede descartar ningún punto y en el divide y vencerás todos los puntos quedan en la franja intermedia: no se ahorra ningún cálculo y se pierde tiempo calculando quien se puede descartar (y no se descarta ninguno).

Por ello es importante mostrar cuantas distancias totales calcula cada algoritmo para demostrar que en el peor caso no hay ahorro de cálculos, sino todo lo contrario y para demostrarlo empíricamente hay que generar dataset aleatorios con el mismo valor x (solo generar datos aleatorios para el eje y).

Por tanto, el programa debería tener una opción que permitiera generar los datos aleatorios para el peor caso (misma x) y para el caso general.

Ejemplo: salida para tamaño de datos entre 1000 y 5000 (Exhaustivo vs DivideVenceras) PEOR CASO

Talla	Exhaustivo Tiempo(mseg)	DivideVenceras Tiempo(mseg)	Exhaustivo Distancias	DivideVenceras Distancias
1000	9.125975	18.651482	499501	995019
2000	37.807271	75.714134	1999001	3989039
3000	96.479107	159.060151	4498501	8981267
4000	154.257473	309.515916	7998001	15976079
5000	246.104394	448.213847	12497501	24968387

Cuando se generan una gran cantidad de datos aleatorios hay muchas probabilidades de que algunos números aparezcan repetidos (aunque se generen mediante divisiones que dan decimales). En el caso de querer forzar el peor caso (misma x y datos aleatorios para la y) corremos el riesgo de que en el dataset generado haya puntos idénticos. Para evitar eso debemos utilizar un mecanismo de generación de aleatorios que garantice que no hay repetidos.

A continuación, se muestra (en C++) un método que genera aleatorios sin repetición (debéis utilizad éste para garantizar que no hay puntos repetidos).

```
void Punto::rellenarPuntos(Punto *p, int n, bool mismax) {
  int num,den;
  double x, y, aux1;
  srand (time(NULL));   // initialize random seed:
  if (mismax) { //PEOR CASO
    for(int i=0; i<n; i++) {
      aux1=rand()%1000+7; //7 y 1007
      y=aux1/((double)i+1+i*0.100); //aux2; //+(i/3.0);
      num=rand()%3;
      y+=((i%500)-num*(rand()%100));
      x=1;
      p[i]=Punto(x,y,i+1);
    }
  }
  else { //CASO MEDIO
    for(int i=0; i<n; i++) {
      num=rand()%4000+1; //genera un número aleatorio entre 1 y 4000
      den=rand()%11+7; //genera un aleatorio entre 7 y 17
      x=num/((double)den+0.37); //division con decimales
      y=(rand()%4000+1)/((double)(rand()%11+7)+0.37);
      p[i]=Punto(x,y,i+1);
    }
  }
}
```

Programa prototipo de ejemplo

Se ha dejado colgado en Moodle un ejecutable de un programa prototipo de ejemplo que implementa (en modo texto) todas las funcionalidades pedidas en la práctica.

El programa implementa algunas funcionalidades adicionales (como la de mostrar la solución de los dataset de ejemplo) y lo podéis usar como prototipo y/o para verificar que los resultados que da vuestro programa son correctos o no.

Nota de la práctica

Para que la práctica sea considerada APTA se deberá implementar en modo texto todas las funcionalidades indicadas. Se obtendrá una mayor nota si el programa se realiza en modo gráfico (mediante un formulario con botones de comando, etc.). Se obtendrán puntos adicionales por mostrar gráficamente la solución y otros adicionales por representar los resultados de comparar los tiempos de ejecución de las estrategias con diferentes tallas en una gráfica.