Objetivos

- Repaso al paradigma recursivo de resolución de algoritmos.
- Estudio de un caso concreto de recursividad: Divide y Vencerás. Beneficios y desventajas.

Conceptos

El término Divide y Vencerás, en su acepción más amplia, es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una **filosofía general para resolver problemas** y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.

En nuestro contexto, Divide y Vencerás es una **técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño**. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Es por ello que **lo más común es utilizar la recursividad para implementar estas técnicas**. Finalmente, las soluciones a los subproblemas elementales serán combinadas para dar una solución final al problema original.

Veamos cómo funciona un algoritmo de Divide y Vencerás mostrando el siguiente esquema:

```
TipoSalida DivideyVenceras(x:TipoEntrada){
    int i,k;
    TipoSalida s
    TipoEntrada subproblemas[N];
    TipoSalida subsoluciones[N];
    If (EsCasoBase(x)) then
          S=ResuelveCasoBase(x);
    else{
       k=Divide(x,subproblemas);
       For(i=0;i<k;i++)
           subsoluciones[i]=DivideyVenceras(subproblemas[i]);
       s=Combina(subsoluciones);
    }
    return s;
}
```

El proceso está compuesto por varias fases:

1. **División**: consiste en **dividir el problema original en k problemas del mismo tipo, pero de menor tamaño**. De esta forma nos aseguramos el hecho de **converger siempre hacia el caso base**.

2. **Resolución de los subproblemas**: directamente si se ajustan al caso base o bien de forma recursiva, aplicando las correspondientes subdivisiones hasta **encontrar subproblemas lo suficientemente sencillos para ser resueltos** mediante el caso base.

3. **Combinación**: combinar las soluciones obtenidas en el paso anterior para **construir la solución al problema original**.

Si nos fijamos, en los casos en los que k=1 tendremos lo que se llama algoritmos de simplificación, y **se parecen mucho a problemas que ya hemos resuelto mediante recursividad**. Por ejemplo, la implementación recursiva del factorial de un número. En este caso, sencillamente se reduce un problema a otro subproblema del mismo tipo, pero de menor tamaño. Otro ejemplo de algoritmo de simplificación es la búsqueda binaria en un vector. La ventaja de estos algoritmos es que sus **tiempos de ejecución suelen ser muy buenos** y además **permiten ser transformados a una implementación iterativa de manera sencilla**.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante tener en cuenta los siguientes factores:

1. **Independencia entre los subproblemas**: Es decir, **que no exista solapamiento entre ellos**. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Como ejemplo pensemos en el cálculo de la sucesión de **Fibonacci**: cada problema se divide en dos subproblemas cuya resolución está relacionada (Fib(5)=Fib(4) + Fib(3) --> Fib(5)= Fib (3) + Fib(2) + Fib(3)), implicando repetir **cálculos ya realizados con anterioridad**.

2. **Número de divisiones y el tamaño de las mismas**: La relación entre estas dos variables va a ser crucial para determinar la complejidad de estos tipos de algoritmos en las ecuaciones de recurrencia.

3. El **reparto de la carga entre los subproblemas**: Es importante que la división en subproblemas se haga de la forma más equilibrada posible. En caso contrario nos podemos encontrar con "anomalías de funcionamiento" como le ocurre al algoritmo de ordenación Quicksort. Éste es un representante claro de los algoritmos Divide y Vencerás, y su caso **peor aparece cuando existe un desequilibrio total en los subproblemas al descomponer el vector original en dos subvectores de tamaño 0 y n–1**. En este caso su orden es **$O(n^2)$**, frente a la mejor complejidad, **$O(n \log n)$**, que consigue cuando descompone el vector en **dos subvectores de igual tamaño**.

Finalmente comentar que, además de ser una forma natural de diseñar técnicas eficientes, **este tipo de algoritmos se adapta perfectamente a la ejecución en entornos multiprocesador**, especialmente en sistemas de memoria compartida donde la comunicación de datos entre los procesadores no necesita ser planeada por adelantado, por lo que subproblemas distintos se pueden ejecutar en procesadores distintos.

Experimentos

E1. (60 min.). Ordenación rápida, conocida como **Quick sort.** El funcionamiento de este algoritmo de ordenación se puede resumir del siguiente modo:

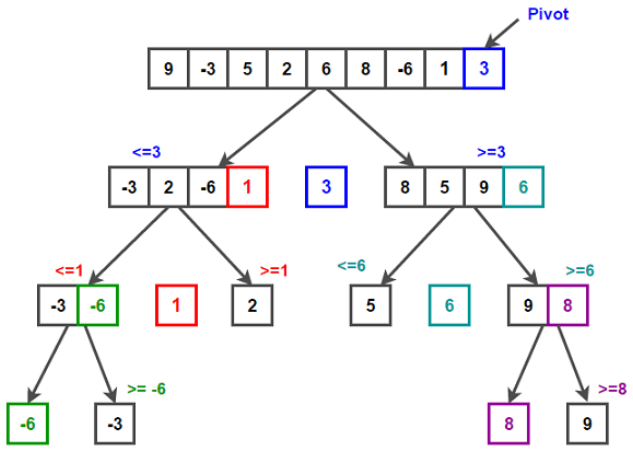
- **Elegir** un elemento del conjunto de elementos a ordenar, al que llamaremos **pivote**.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que **a un lado queden todos los menores que él, y al otro los mayores**. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- **La lista queda separada en dos sublistas**, una formada por los elementos **a la izquierda del pivote, y otra por los elementos a su derecha**.
- **Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento**. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la **eficiencia** del algoritmo **depende de la posición en la que termine el pivote elegido**.

- En el **mejor caso**, el pivote termina en el **centro de la lista**, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es **$O(n \cdot \log n)$**.
- En el **peor caso**, el **pivote termina en un extremo de la lista**. El orden de complejidad del algoritmo es entonces de **$O(n^2)$**. El peor caso dependerá de la implementación del algoritmo, aunque **habitualmente ocurre en listas que se encuentran ordenadas**, o casi ordenadas.
- En el caso promedio, el orden es $O(n \cdot \log n)$.

Por lo tanto, por norma general, la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote, buscando obtener siempre el mejor caso.

Un ejemplo de funcionamiento de este algoritmo sería:



Si recorremos todos los nodos hoja que se observan en el anterior grafo, el resultado sería el vector ordenado:

-6 -3 1 2 3 5 6 8 9

1. Implemente el algoritmo de ordenación descrito, usando como pivote el primer elemento. Pruebe con la entrada del ejemplo: 9, -3, 5, 2, 6, 8, -6, 1, 3.
2. Pruebe el algoritmo con:
   - Vector ordenado.
   - Vector ordenado, de mayor a menor.

   ¿Qué diferencia se puede observar en el funcionamiento del algoritmo al usar estas dos entradas?

3. Modifique el algoritmo para establecer la selección del pivote como el último elemento del vector.

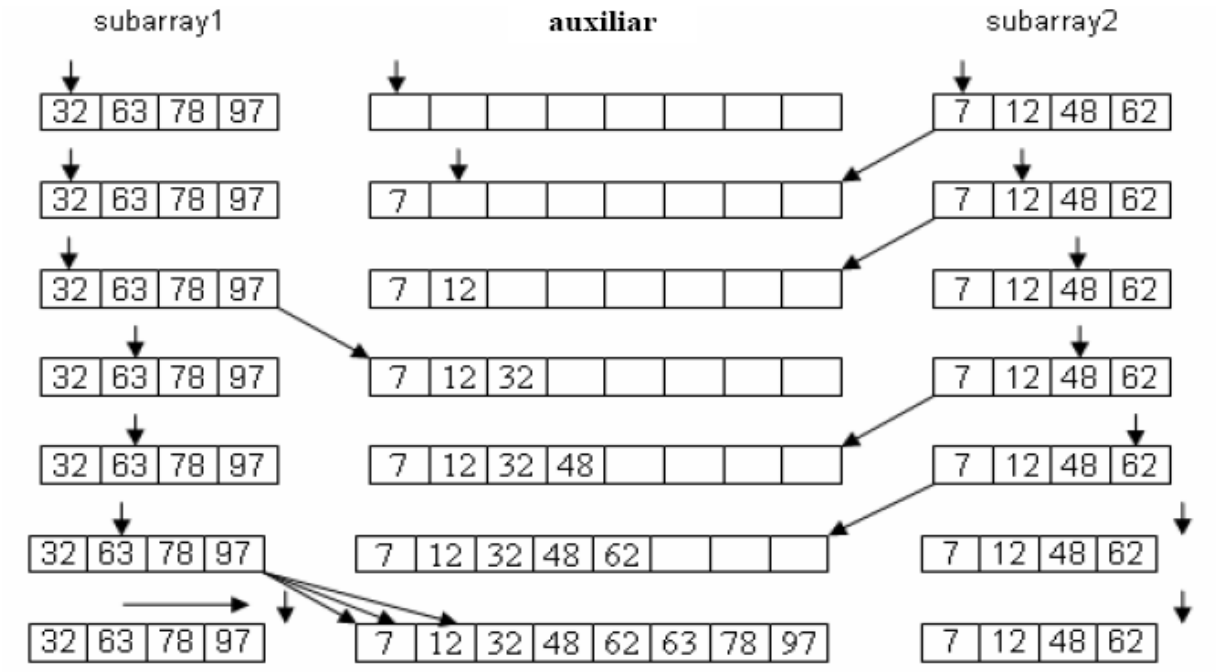
Problemas

P1. (60 min) La ordenación por mezcla, o **Merge sort**, es una técnica que sigue la filosofía de divide y vencerás. Este algoritmo, desarrollado por Von Neumann en 1945, consigue muy buenos tiempos, teniendo una complejidad de O(nlogn) en cualquier caso. Su problema es que necesita más cantidad de memoria ya que requiere de una estructura de datos auxiliar para llevar a cabo la mezcla.

La idea es muy sencilla: dividir un array por la mitad, ordenar cada una de sus mitades y mezclar esas dos mitades en el mismo vector. Así, podremos encontrarnos con las siguientes situaciones:

> **Caso base**: cuando el vector que vayamos a ordenar tenga un solo elemento. En ese caso no hemos de hacer nada
>
> **Caso general**: si tiene más de un elemento tendremos que partirlo en dos y ordenar cada parte (llamada recursiva). Al final, esas dos partes se mezclan.

La mayor complejidad reside en la forma de mezclar los subvectores ordenados. Estos dos subvectores se han de recorrer simultáneamente, pasando cada vez al vector auxiliar el menor de entre los elementos seleccionados en los dos subvectores. El índice del subvector que ha proporcionado el elemento menor aumenta y se repite el proceso. Cuando un subvector acaba, solo tenemos que volcar el otro subvector en el array auxiliar y posteriormente pasar esa parte ordenada a la posición del array original que corresponda. En la siguiente figura partimos de dos subvectores ordenados para mostrar un ejemplo de como se realiza la mezcla:



Al finalizar este proceso solo habría que volcar el array auxiliar sobre el array original.

Este problema consiste en Implementar el algoritmo de ordenación por mezcla y realizar comprobaciones de eficiencia comparándolos con otros algoritmos de ordenación tales como: selección, inserción o burbuja.

Ayuda: considere las siguientes cabeceras para los procedimientos de su programa:

> **Ordenar (int *v, int inicio, int fin)**: método recursivo. El parámetro v es el vector a ordenar e inicio y fin son índices que nos indican a que parte del vector original nos estamos refiriendo.
>
> **Mezclar (int *v, int inicio1, int fin1, int inicio2, int fin2)**: método que realiza la mezcla. De nuevo, v es el vector a ordenar y el resto de parámetros enteros nos muestran el principio y el fin de cada uno de sus subvectores ordenados que hay que mezclar. No olvide utilizar un vector auxiliar para ir volcando el resultado de la mezcla. Tampoco olvide que dicho resultado tendrá que volver al vector original a partir de una determinada posición

P2. (60 min). La búsqueda binaria es un claro ejemplo de aplicación de Divide y Vencerás. Cada vector analizado es dividido en dos trozos y se analiza el trozo donde se estime se encuentre el valor buscado. Nos aprovechamos del orden de los elementos en el vector para ello. Utilizando como modelo la búsqueda binaria vamos a solucionar el problema de encontrar la mediana de dos vectores. Sean X e Y dos vectores de tamaño n, ordenados de forma creciente. Necesitamos implementar un algoritmo para calcular la mediana de los 2n elementos que contienen X e Y. Recordemos que la mediana de un vector de k elementos es aquel elemento que ocupa la posición (k+1)/2 una vez el vector está ordenado de forma creciente. Dicho de otra forma, la mediana es aquel elemento que, una vez ordenado el vector, deja la mitad de los elementos a cada uno de sus lados. Como en nuestro caso k = 2n (y por tanto par) buscamos el elemento en posición n de la unión ordenada de X e Y.

> **Idea**: detectar casos base y general como se hacía en la recursividad
>
> Caso Base1: Dos vectores de un elemento cada uno (n=1). En este caso, la mediana será el menor elemento de los dos ya que ocuparía la primera posición en la unión de los dos vectores: pos= k+1/2 = 2+1/2 = 1.
>
> Hemos de encontrar la manera de dividir el problema en varios subproblemas. Como X e Y están ordenados es fácil encontrar sus medianas: mx y my, ya que serán sus elementos centrales. Podemos encontrarnos con varios casos:
>
> 1. mx==my: En este caso estos dos elementos aparecerían en el centro del vector si los uniéramos, por lo cual representan a la mediana. Podríamos decir pues que también es un Caso base2.
> 2. mx>my : La mediana final será mayor que my y menor que mx. La mediana estará o en la segunda mitad del vector Y o en la primera mitad del vector X. Caso general.
> 3. mx<my: Podríamos decir que la mediana final será mayor que mx y menor que my. Por lo cual tendremos que buscar en la segunda mitad del vector X y en la primera mitad del vector Y. Caso General.

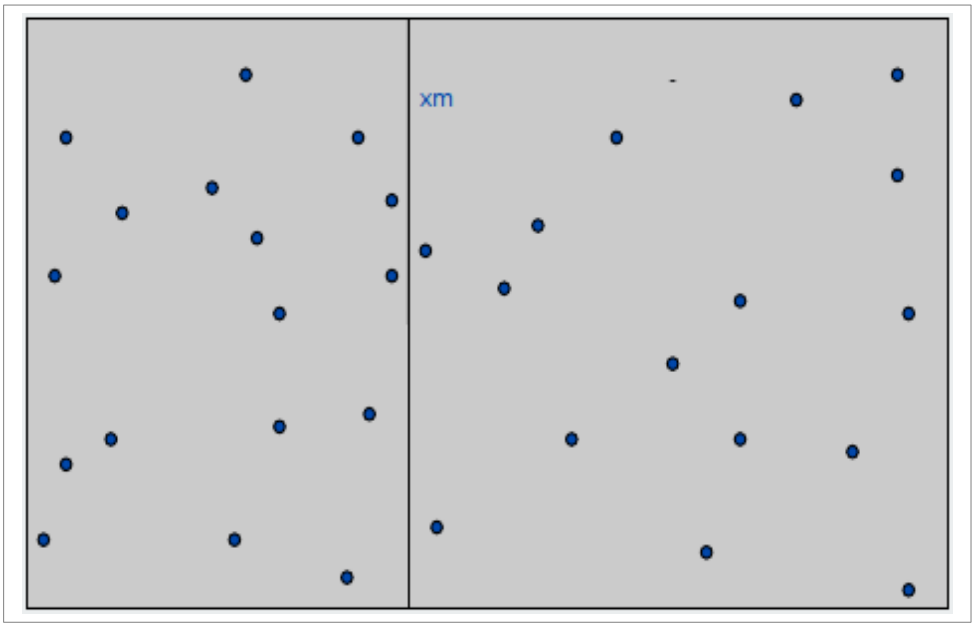
P3. (210 min) Dados n puntos en el plano encontrar la pareja de puntos con la menor distancia euclídea entre ellos. La distancia euclídea entre dos puntos p=(p_x,p_y) y q=(q_x,q_y) es:

$$d = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

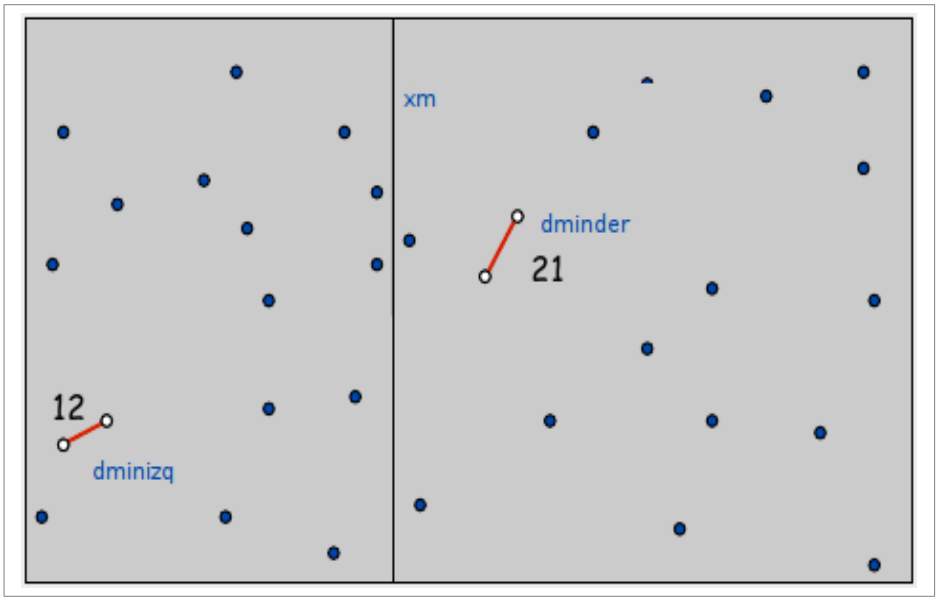
Debe implementar una solución basada en la técnica de Divide y Vencerás siguiendo los pasos siguientes:

Previamente a la ejecución de la función, ordenar el conjunto de puntos según la coordenada x Ordenarlo también (en otro conjunto separado) según la coordenada y.

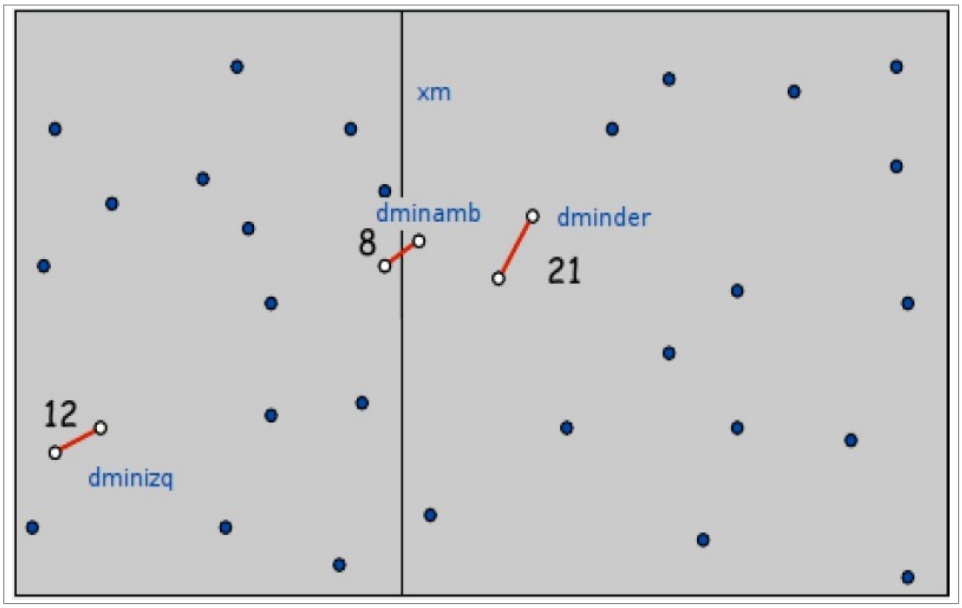
- Dividir: trazar una línea imaginaria $x=x_m$ en la mediana del conjunto, de modo que la mitad de los puntos queden a su izquierda y la otra mitad a su derecha.



- Conquistar: encontrar la pareja más cercana en cada mitad. Aplicando esta función de forma recursiva, calcular la pareja con distancia mínima entre los puntos pertenecientes a la mitad izquierda (d_{minizq}) y la de los puntos de la mitad derecha (d_{minder}).



- Combinar: encontrar la pareja más cercana en cada mitad. Aplicando esta función de forma recursiva, calcular la pareja con distancia mínima entre los puntos pertenecientes a la mitad izquierda (d_{minizq}) y la de los puntos de la mitad derecha (d_{minder}).

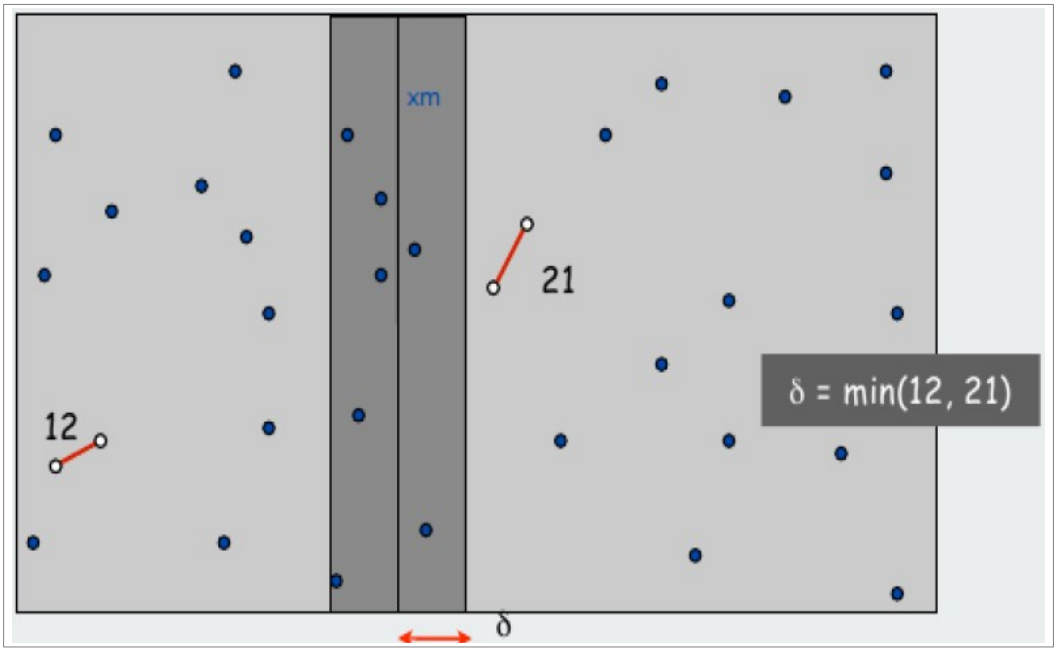


- Vencer: el resultado es min(d_{minizq}, d_{minder}, d_{minamb}).

Puesto de otra forma:

1. Trazar línea vertical que deje N/2 puntos en cada lado.
2. Buscar recursivamente la pareja más cercana en cada mitad.
3. Buscar la pareja más cercana con un punto en cada mitad.
4. Devolver la mejor de las tres soluciones.

El paso 3 (combinar) parece requerir de una búsqueda exhaustiva de O(n2), pero puede simplificarse teniendo en cuenta que nos interesa una pareja con un punto en cada mitad y cuya distancia sea menor que δ, siendo δ= min(d_{minizq}, d_{minder}). Podemos, por tanto, restringir la búsqueda a una franja de ancho 2δ situada alrededor de x_m, es decir, entre x_m-δ y x_m+δ:



Obviamente, los puntos de una mitad que estén fuera de esa franja no pueden estar a distancia menor que δ de un punto en la otra mitad. Teniendo el conjunto de puntos en esa franja central ordenado por su coordenada y podemos recorrerlo rápidamente para hallar la distancia mínima entre puntos de ese conjunto. Una mejora adicional es que solo hay que comparar cada punto con aquellos posteriores (según el orden en y) si la diferencia entre sus coordenadas y es menor que δ.

El programa a implementar deberá tener una función principal que presente en pantalla un menú con las siguientes opciones:

Opción 1) Generación de puntos: se introducirá por teclado el tamaño del conjunto de puntos (con un máximo de MAXP, definido en el programa) y se generaran los puntos aleatoriamente, con coordenadas entre -100000 y +100000. Este vector de puntos podrá ser usado.

Opción 2) Testeo: si el conjunto de puntos ha sido generado (es decir, se ha seleccionado previamente la opción 1) no se pedirán más datos. Pero si no se ha pasado por la opción 1, se pedirán por teclado el tamaño del conjunto y las coordenadas x e y de cada punto. A continuación se aplicara la solución divide y vencerás al conjunto de puntos existente (ya sea el generado aleatoriamente por la opción 1 o el introducido por teclado). Se mostraran por pantalla las coordenadas de los dos puntos con distancia mínima y su distancia.

Nota: para generar un numero aleatorio entre a y b usando rand (que devuelve uno entre 0 y RAND_MAX) debes usar la expresión: (rand()/(double)RAND_MAX)*(b-a)+a

Bibliografía Básica

1. **Documentación Oficial Oracle JAVA**. https://docs.oracle.com/en/java/index.html
2. **Fundamentos de Algoritmia**. G. Brassard, P. Bratley. Prentice Hall.
3. **Introduction to Algorithms**. TH. Cormen, CE. Leiserson, RL. Rivest, C. Stein. MIT Press, 2001.
4. **Técnicas de diseño de algoritmos**, Guerequeta, R., A. Vallecillo. Universidad de Málaga / Manuales, 1997. Capítulo 3, páginas 105 a 108.

Datos de la Práctica

Autor del documento:
Alberto González Rivas (noviembre 2019).

Revisión del documento:

Estimación temporal:

- Parte presencial: 120 minutos.
  - Explicación inicial: 60 minutos.
  - Experimentos: 60 minutos.

- Parte no presencial: 360 minutos.
  - Lectura y estudio del guion y bibliografía básica: 30 minutos
  - Problemas: 330 minutos