Explicación al problema del cálculo del rango de un conjunto de puntos.
Ejercicio 10 del tema 5. Divide y vencerás.[1]
Enero 2014.

Comenzamos explicando una versión más sencilla del problema. Hacemos la suposición de que todos los puntos tienen coordenada x diferente.

El problema se resuelve siguiendo el esquema del algoritmo mergesort. Se realizan dos llamadas recursivas una con la mitad izquierda del vector, I y otra con la mitad derecha del vector, D, cada una de las cuales calcula es rango del conjunto de puntos de su parte del vector.

Para poder calcular los puntos de la parte I que dominan a los puntos de la parte D y viceversa en un tiempo lineal se necesita que los puntos del vector original estén ordenados por la coordenada x. De esta forma los puntos de la parte I no pueden dominar a los puntos de la parte D, ya que su coordenada x es más pequeña. Basta por lo tanto con comprobar a que puntos de la parte I domina cada punto de la parte D.

Sin embargo, la solución directa a este problema que comprobaría cada punto de D con cada punto de I no es válida ya que tiene coste cuadrático y este cálculo es necesario realizarlo en coste lineal para obtener el coste total del orden de $\mathcal{O}(n \log n)$ requerido para el algoritmo.

Si ordenamos cada parte (I y D) por la coordenada y antes de calcular los puntos de D que dominan a los de I podremos resolver el problema en tiempo lineal. La idea es la siguiente: se declaran dos índices: i y j el primero para recorrer la parte I y el segundo para recorrer la parte D. Recorremos cada parte de derecha a izquierda, para que el algoritmo sea mas sencillo. Mas adelante se explicará como se puede hacer el cáculo realizando el recorrido de izquierda a derecha.

1. Si la componente $v[i].y$ es menor que $v[j].y$ entonces el punto j domina al punto i, ya que por estar en la parte D tenemos garantizado que su coordenada x también es mayor. Como los elementos de la parte I se encuentran ahora ordenados por la coordenada y sabemos que $v[j]$ domina también a todos los $v[k]$ tales que $k < i$, o sea, a todos los elementos que están a la izquierda de i. Por lo tanto no hace falta comprobarlos.

2. Si la componente $v[i].y$ es mayor o igual que $v[j]$ entonces el punto j no domina al punto i. Ademas sabemos que como los elementos de D están ordenados ningún punto de D puede dominar al punto i.

El código para calcular los puntos de D que dominan a los puntos de I es el siguiente:

```
int i = m; int j = b;
while (i >= a && j > m) {
     if (v[j].y <= v[i].y) i--;
     else {
          v[j].r+=i-a+1;
           j--;
    }
}
```

Se podría hacer el recorrido de I y D de izquierda a derecha utilizando una variable en la que acumulamos los puntos de la parte izquierda que van siendo dominados por los puntos de la parte derecha. Como en el caso anterior se requiere que ambas partes I y D estén ordenadas por la coordenada y cuando se realiza este cálculo.

```
int aux = a;
for (int i = m+1; i <= b; i++) {
   while (aux <= m && v[aux].y < v[i].y) aux++;
   v[i].r += (aux-a);
   }
```

[1]La solución al problema realizando el recorrido de los subvectores de izquierda a derecha fue propuesta por un/a alumno/a del curso de EDA del doble grado en Informática y Matemáticas

Se observa que aunque hay un doble bucle, el valor de la variable *aux* siempre crece, y no se vuelve a inicializar dentro del primer bucle, por lo que sólo recorre la parte izquierda una vez. El coste por lo tanto es lineal.

Observar la diferencia con el siguiente código, en el que no se utiliza la variable para acumular. En su lugar se modifica el valor del rango de cada componente cada vez que se considera un valor. En este caso el doble bucle hace la solución cuadrática ya que el bucle interior recorre varias veces los mismos elementos. Esta solución por lo tanto no es aceptable.

```
   int i = a; int j = m+1;
   while (i <= m && j <=  b) {
       if (v[j].y <= v[i].y) j++;
       else {
             for (int k = j; k <= b; k++) v[k].r++;
              i++;
      }
}
```

El programa completo con la suposición de que los puntos tienen diferente coordenada x es el siguiente:

```
/* La ordenacion inicial por la coordenada x se realiza con la funcion qsort de la libreria stdlib
 * para mostrar como se utiliza.
 * Se podria utilizar cualquier otro metodo de ordenacion con coste nlog n.
 */


#include <iostream>
using namespace std;
#include <stdlib.h>     /* qsort */


// los campos x,y son las coordenadas del punto y
// el campo r es el rango correspondiente que se va calculando
struct puntos {
   int x;
   int y;
   int r;
};

// El vector p debe estar ordenado entre a y (a+b)/2 y entre (a+b)/2+1 y b de
// forma creciente por la coordenada y.
// Al finalizar el algoritmo el vector p estara ordenado de forma creciente
// entre a y b por la coordenada y
void mezclaOrdenada(puntos p[], int a, int b) {
   puntos u[b-a+1];
   int i,j,k;
   int m = (a+b)/2;
   i=a; j=m+1;k=0;
   while ( (i<=m)&&(j<=b) ) {
      if (p[i].y <= p[j].y) { u[k] = p[i]; i = i+1;}
      else { u[k]=p[j]; j=j+1;}
      k=k+1;
   }
   while (i<=m) {
      u[k] = p[i]; i = i+1; k = k+1;
   }
   while ( j <= b ) {
      u[k] = p[j]; j = j+1; k = k+1;
   }
```

```
   for ( k = a ; k <= b ; k++ ) p [ k ] = u [ k -a ] ;
}

// El vector v esta ordenado de forma creciente entre a y b por la coordenada x
// Al finalizar la ejecucion el vector v estara ordenado de forma creciente
// por la coordenada y. El campo r de cada punto del vector v entre a y b tendra
// el numero de inversiones del punto entre a y b
void rangosRec (puntos v[], int a, int b)
{
   // Casos base
   if (a > b) return;
   else if (a == b) v[a].r = 0;
   else if (a+1 == b) {
      if (v[a].x > v[b].x && v[a].y > v[b].y) {v[a].r = 1; v[b].r = 0;}
      else if (v[a].x < v[b].x && v[a].y < v[b].y) {v[b].r = 1; v[a].r = 0;}
      else {v[a].r = 0; v[b].r = 0;}
   }
   else { // caso recursivo
      int m = (a+b) / 2;
      rangosRec(v,a,m);
      rangosRec(v,m+1,b);
     // mezcla
      int i = m; int j = b;
      while (i >= a && j > m) {
         if (v[j].y <= v[i].y) i--;
         else {
            v[j].r+=i-a+1;
            j--;
         }
      }
      mezclaOrdenada(v, a, b);
   }
}

// Funcion auxiliar para el quicksort. La ordenacion se realiza por la coordenada x
int compare (const void * a, const void * b)
{
   if ( (*(puntos*)a).x <  (*(puntos*)b).x ) return -1;
   else if ( (*(puntos*)a).x == (*(puntos*)b).x ) return 0;
   else return 1;
}

// Se ordena el vector de entrada por la coordenada x y se llama a la funcion recursiva
void rangos (puntos v[], int n)
{
   qsort (v, n, sizeof(v[0]), compare);
   rangosRec(v,0,n-1);
}

int main (){
   const int TAM = 5;
   puntos p;
   puntos v[TAM];
   for (int i = 0; i < TAM; i++) {
      cout << "Punto " << i << endl;
      cin >> p.x >> p.y;
      v[i] = p;
```

```
   }
   rangos(v,TAM);
   for (int i = 0; i < TAM; i++)
      cout << "El rango del punto " << v[i].x << " " << v[i].y << " es " << v[i].r << endl;
   return 0;
}
```

Solución del problema cuando varios puntos pueden tener la misma coordenada x. El problema cuando varios puntos pueden tener la misma coordenada x es que al dividir el vector en dos partes I y D puede haber puntos en ambas partes con la misma coordenada x. Al calcular los puntos de D que dominan a los puntos de I con el código propuesto para el caso anterior suponemos que todos los puntos de D tienen coordenada x estrictamente mayor que los de I y por ello basta con comprobar la coordenada y.

```
int i = m; int j = b;
while (i >= a && j > m) {
     if (v[j].y <= v[i].y) i--;
     else {
         v[j].r+=i-a+1;
          j--;
    }
}
```

Comprobar la coordenada x, para ver que es estrictamente mayor requeriría comprobar cada uno de los puntos a la izquierda de la variable i en cada vuelta del bucle, no podríamos sumarlos directamente como hacemos con la instrucción `v[j].r+=i-a+1;`, lo que tendría coste cuadrático.

Si utilizamos la solución de hacer el recorrido de izquierda a derecha utilizando el acumulador, podríamos comparar las coordenadas x y restar los puntos que tienen la misma coordenada x. Esta solución sigue teniendo coste lineal.

```
int aux = a; marcador = 0;
for (int i = m+1; i <= b; i++) {
   while (aux <= m && v[aux].y < v[i].y) {
      if (v[aux].x == v[i].x)  marcador++;
      aux++;
      }
   v[i].r += (aux-a)-marcador;
   }
```

Otra solución interesante del problema se obtiene si nos damos cuenta de que realmente no se necesita tener el vector inicial ordenado por la coordenada x. El requisito necesario es que antes de realizar las llamadas recursivas los elementos de la parte izquierda del vector sean menores que los elementos de la parte derecha. Podemos utilizar el algoritmo de partición en su versión mejorada, que divide el vector en tres partes, los elementos menores que el pivote, los elementos mayores y los elementos iguales. Las llamadas recursivas se realizan únicamente sobre los elementos menores y mayores. Sin embargo, los elementos iguales habrá que ordenarlos por la coordenada y para poder hacer la mezcla de las tres partes en tiempo lineal y que la llamada recursiva devuelva el vector ordenado por la coordenada y. Esta solución plantea los mismos problemas en cuanto al coste en el caso peor que el quicksort.

```
// Particion del vector en tres zonas, elementos menores que el pivote, iguales y mayores
void particion2(puntos v[], int a, int b, puntos pivote, int& p, int& q)
{
   int k;
   puntos aux;
   p=a;k=a;q=b;
   while (k<=q) {
      if (v[k].x == pivote.x) {k = k+1;}
      else if (v[k].x < pivote.x)  {
```

```
         aux = v[p]; v[p] = v[k]; v[k] = aux;
         p= p+1; k=k+1;
      }
      else {aux = v[q]; v[q] = v[k]; v[k] = aux; q=q-1;}
   }
}

// Ordena los puntos por el metido de mergesort respecto a la coordenada y.
// Utiliza la mezcla ordenada del apartado anterior
void ordenarY(puntos v[],int p,int q) {
   if (p < q) {
      int m = (p+q)/2;
      ordenarY(v,p,m);
      ordenarY(v,m+1,q);
      mezclaOrdenada(v,p,q,m);
   }
}

// Mezcla tres partes ordenadas de un vector
// Primera parte entre a y p-1
// Segunda parte ente p y q
// Tercera parte ente q+1 y b
void mezclaOrdenada3(puntos v[],int a,int p,int q,int b){
   mezclaOrdenada(v,a,q,p);
   mezclaOrdenada(v,a,b,q);
}

// Calcula los puntos entre x2 e y2 que dominan a los puntos entre x1 e y1
void calcularRango(puntos v[], int x1, int y1, int x2, int y2) {
   int i = y1; int j = y2;
   while (i >= x1 && j >= x2) {
      if (v[j].y <= v[i].y) i--;
      else {
         v[j].r+=i-x1+1;
         j--;
      }
   }
}

// Procedimiento recursivo para calcular el rango de un conjunto de puntos
void rangosRec2 (puntos v[], int a, int b)
{
    // Casos base
   if (a > b) return;
   else if (a == b) v[a].r = 0;
   else if (a+1 == b) {
      if (v[a].x > v[b].x && v[a].y > v[b].y) {v[a].r = 1; v[b].r = 0;}
      else if (v[a].x < v[b].x && v[a].y < v[b].y) {v[b].r = 1; v[a].r = 0;}
      else {v[a].r = 0; v[b].r = 0;}
   }
   else { // caso recursivo
      int p,q; puntos piv = v[a];
      particion2(v,a,b,piv,p,q);
      rangosRec2(v,a,p-1);
      rangosRec2(v,q+1,b);
      ordenarY(v,p,q);
      // rango de la parte central y la parte izquierda
```

```
      calcularRango(v,a,p-1,p,q);
      // rango de la parte derecha y la parte izquierda
      calcularRango(v,a,p-1,q+1,b);
      // rango de la parte derecha y la parte central
      calcularRango(v,p,q,q+1,b);
      mezclaOrdenada3(v,a,p,q,b);
   }
}

// Se ordena el vector de entrada por la coordenada x y se llama a la funcion recursiva
void rangos2 (puntos v[], int n)
{
   rangosRec2(v,0,n-1);
}
```