complexité
Examen

Nous nous intéresserons au problème suivant : parmi *n* points rechercher le couple de points dont la distance est minimum, dans le cas à une dimension (points d'une droite), et dans le cas à deux dimensions (points du plan).

Une dimension.

Le premier algorithme qui vient à l'esprit est l'algorithme « force brute » qui énumère tous les couples de points, et calcule ceux dont la distance est minimale :

```
void CPPu1(Point * tabPoints, int nbPoints, int & iP1, int & iP2){
    double distMin = 10.0E10; double dist ;
    for( int i = 0; i <  nbPoints; ++i)
       for( int j = i+1; j <  nbPoints; ++j){
         dist = tabPoints[i].distance(tabPoints[j]) ;
         if  (dist  < distMin){
            iP1 = i; iP2 = j;
            distMin  = dist ;
         }
       }
}
```

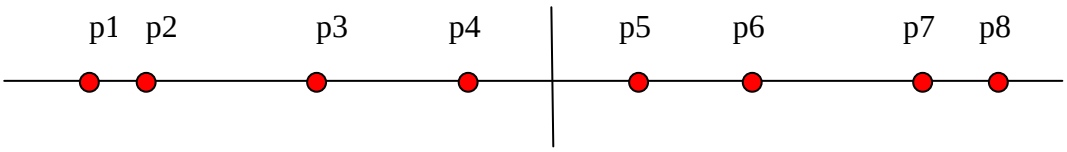
Question 1.

Combien de fois est appelée la fonction *distance* pour un tableau de *Point* de *n* éléments ?

Si le tableau de *Point* est trié suivant l'ordre croissant, on peut envisager deux stratégies différentes :

Question 2.

D'abord une stratégie « diviser pour régner » :



- le tableau des points est coupé en deux, On cherche le couple des points les plus proches à gauche, puis à droite,
- Puis on calcule la distance du point le plus à gauche de la partie de droite au point le plus à droite de la partie de gauche.
- Le couple de points les plus proche est parmi ces trois là.

```
void CPPu2(::Point * tabPoints, int nbPoints, int & iP1, int & iP2){
    trier( tabPoints, nbPoints);
    CPPu21(tabPoints, 0, nbPoints-1, iP1, iP2);
}
```

```
double CPPu21(Point * tabPoints, int deb, int fin, int & iP1, int &
iP2){
    double distMin = 10.0E10;
    if (deb>=fin) return  distMin;
    else{
       int iP11, iP21, iP12, iP22, iP13, iP23;
       int m = (deb+fin)/2;
       double g = CPPu21(tabPoints, deb, m, iP11, iP21);
       double d = CPPu21(tabPoints, m+1, fin, iP12, iP22);
       iP1 = iP11; iP2 = iP21;
       if( g > d){
          iP1 = iP12; iP2 = iP22; g = d;
       }
       d = tabPoints[m].distance(tabPoints[m+1]);
       if( g > d){
          iP1 = m; iP2 = m+1;g = d;
       }
       return g;
    }
}
```

Calculer le nombre d'appels de la fonction *distance* dans la fonction *CPPu21*. En déduire la complexité de *CPPu2*.

Question 3.

Une fois le tableau trié, il suffit d'examiner tous les couples de points adjacents, le couple de points de distance minimale est forcément parmi ceux-là. Programmer cette stratégie, compter le nombre d'appels à la fonction *distance*, et en déduire la complexité de cette stratégie.

Deux dimensions.

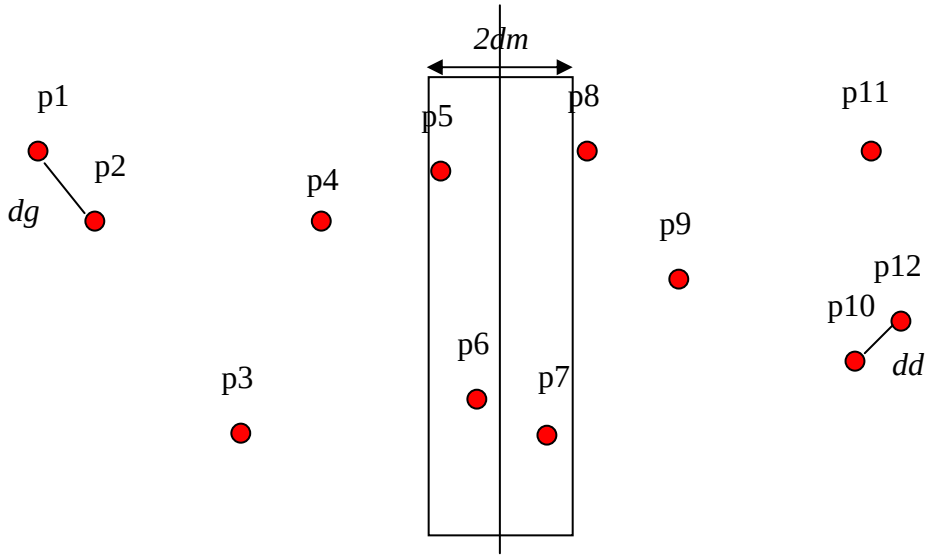
Question 4.

Programmer la stratégie force brute pour un tableau de points du plan, et évaluer le nombre d'appels de la fonction *distance*.

Question 5.

La stratégie « diviser pour régner » peut s'exprimer de la façon suivante :

```
void CPPu3(::Point * tabPoints, int nbPoints, int & iP1, int & iP2){
    trier( tabPoints, nbPoints);
    CPPu31(tabPoints, 0, nbPoints-1, iP1, iP2);
}
```



L'algorithme de *CPPu31* est le suivant :

1. Si le tableau a 0 ou 1 élément renvoyer ∞
2. Si le tableau a plus de 2 éléments, le couper en deux parties (d, m) et (m+1, f) calculer la distance minimale à gauche *dg*, puis la distance minimale à droite *dd*. soit *dm* = min(*dg*, *dd*).
   - Soient b1, b2 les indices délimitant la bande des éléments tels que un élément appartenant à l'intervalle (b1, m) soit distant de au plus *dm* d'un élément de la bande (m+1, b2)
   - Ordonner tous les points de l'intervalle (b1, b2) suivant leurs ordonnées, puis rechercher dans cet intervalle le couple de points de distance minimale. On montre (ce n'est pas à faire) qu'il suffit de comparer un point à ses 7 successeurs.

Montrer que la complexité de l'algorithme de *CPPu31*est donné par la récurrence :

$t_1 = 1$
$t_n = 2t_{n/2} + 7n + n\log_2(n)$ et résoudre cette récurrence.