CPA

Généralité sur le temps d'exécution des algorithmes

n	kB	mB	gB	tB
$O(n)$	~µs	~ms	~sec	~h
$O(n\log(n))$	30 µs	60 ms	9 sec	~h
$O(n^2)$	~ms	~h	~A	~M
$O(n^3)$	~s	~A	~Ma	N/A
$O(n^4)$	~h	~Ma	N/A	N/A

Avec un ordinateur de l'ordre du Ghz (Soit 10^9 calcul/s)

Notions de mathématiques a connaitre

Produit vectoriel

Le produit vectoriel de deux vecteurs notés *a* et *b* dans un repère en deux dimensions :

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} = a_1 b_2 - a_2 b_1$$

Donc dans le cas de deux vecteurs entre deux points mettons *u*(A, B) et *v*(C, D)

$$(x_B - x_A) * (y_D - y_C) - (y_B - y_A) * (x_D - x_C)$$

Si le résultat est :

- < 0 CD est **sous** AB
- = 0 CD et AB sont colinéaires
- > 0 CD est **au dessus** de AB dans le plan

Implémentation

```
int crossproduct(Point p, Point q, Point s, Point t){
  return (((q.x-p.x)*(t.y-s.y))-((q.y-p.y)*(t.x-s.x)));
}
```

Produit scalaire

Orthogonalité : les vecteurs $\overrightarrow{OA}$ et $\overrightarrow{OB}$ sont orthogonaux si l'un ou l'autre des vecteurs est nul ou si l'angle géométrique AOB est droit. En matière de produit scalaire, cela se traduit par une seule condition $\overrightarrow{OA}$ et $\overrightarrow{OB}$ sont orthogonaux si et seulement si $\overrightarrow{OA} \cdot \overrightarrow{OB} = 0$.

Colinéarité : les vecteurs $\overrightarrow{OA}$ et $\overrightarrow{OB}$ sont colinéaires si et seulement si les points O, A et B sont sur une même droite. En matière de produit scalaire, cela se traduit par $\overrightarrow{OA}$ et $\overrightarrow{OB}$ sont colinéaires si et seulement si $|\overrightarrow{OA} \cdot \overrightarrow{OB}| = OA \times OB$. Cette définition se lit aussi ainsi : deux vecteurs sont colinéaires si la valeur absolue de leur produit scalaire est égale au produit de leurs longueurs.

Angle géométrique : si $\overrightarrow{OA}$ et $\overrightarrow{OB}$ sont deux vecteurs non nuls, l'angle géométrique $\widehat{AOB}$ est déterminé par l'égalité $\cos(\widehat{AOB}) = \dfrac{\overrightarrow{OA} \cdot \overrightarrow{OB}}{OA \times OB}$.

Implémentation

```
1   public double dotProd(Point p, Point q, Point s, Point t){
2     return ((q.x-p.x)*(t.x-s.x)+(q.y-p.y)*(t.y-s.y));
3   }
```

Positions barycentriques

En mathématiques, le barycentre d'un ensemble fini de points du plan ou de l'espace est un point qui permet de réduire certaines combinaisons linéaires de vecteurs. Les coordonnées de ce barycentre dans un repère cartésien correspondent alors aux moyennes arithmétiques des coordonnées homologues de chacun des points considérés.

Collision d'objets géométriques

Problème du cercle couvrant minimum

Entrée : *Points*, liste de points ayant des coorodnnées x et y dans un plan.

Sortie : Cercle de taille minimum couvrant tous les points d'un nuage de point.

Algorithme naïf

Cet algorithme repose sur les lemmes :

- Si un cercle de diamètre égale à la distance de deux points de la liste couvre tout autre point de la liste, alors ce cercle est un cercle couvrant de rayon minimum.
- En 2D, il existe un et un seul cercle passant par 3 points non-colinéaires.

Corps

```
1  Pour tout point p dans points :
2      Pour tout point q dans points :
3          c = cercle de centre (p+q)/2 de diamètre |pq|
4          Si c couvre tous les points de points alors :
5              retourner c
```

```
resultat = cercle de rayon infini
Pour tout point p dans points :
    Pour tout point q dans points :
        Pour tout point r dans points :
            c = cercle circonscrit de p, q et r
            Si c couvre Points et que c < Resultat alors :
                reusltat = c

Retourner c
```

Coûts

- 1 - 5 : $O(n^3)$
  - 3 : $O(1)$
  - 4- 5 : $O(n)$
- 6 : $O(1)$
- 7 - 13 : $O(n^3)$
  - 10 : $O(1)$
  - 11 - 12 : $O(n)$
- 14 : $O(1)$

Au total $O(n^3 + n^4 + 1) = O(n^4)$

Algorithme incrémental (Algorithme de Ritter)

Idée : Si le cercle ne couvre pas tous les points alors créer un nouveau cercle incluant l'ancien cercle et au moins un nouveau point.

Tout problème algorithmique possède (au moins) un trade-off: temps de calcul vs. qualité du résultat.

L'algorithme de Ritter est un algorithme d'approximation du cercle minimum : on dégrade la qualité du résultat pour un temps de calcul plus rapide. Ceci dit, l'algorithme Ritter a à la fois le mérite de calculer très vite, tout en ne pas trop dégradant la qualité du résultat retourné.

Corps

```
dummy = Point aléatoire de Points
p = point en 0, 0
q = point en 0, 0
distMax = -infini
Pour tout point i de Points:
    Si dummy.distance(i) > distMax :
        distMax = dummy.distance(i)
        p = i

distMax = -infini
Pour tout point i de Points:
    Si p.distance(i) > distMax:
        distMax = p.distance(i)
        q = i

C = (p+q)/2

CERCLE = cercle centré en C

```

```
Retirer p et q de Points

Tant que points n'est pas vide:
    S = point aléatoire de Points
    Si S couvert par CERCLE alors:
        Retirer S de Points
    Sinon :
        T = Intersection de (CS) et CERCLE la plus éloignée de S
        C' = (S+T)/2
        CERCLE = cercle centré en C'
Retourner CERCLE
```

Coûts

- 1 = $O(1)$
- 2 = $O(1)$
- 3 = $O(1)$
- 4 = $O(1)$
- 5 - 8 = $O(n)$
    - 7 : $O(1)$
    - 8 : $O(1)$
- 10 : $O(1)$
- 11 - 14 : $O(n)$
    - 13 : $O(1)$
    - 14 : O(1)
- 16 : $O(1)$
- 18 : $O(1)$
- 20 : $O(1)$
- 22 - 29 : $O(n)$
    - 23 : $O(1)$
    - 25 :$O(1)$
    - 27 : $O(1)$
    - 28 : $O(1)$
    - 29 : $O(1)$
- 30 : $O(1)$

Soit au total $O(9 + n + n + n) = O(n)$

Implementation

Une implémentation de cet algorithme est disponible ici.

Algorithmes de précalcul

Idée : Des algorithmes retirant les mauvaises solutions évidentes

Entrée : Liste *Points* de point ayant des coordonnées x et y

Sortie : Liste de points allégée des mauvaises solutions évidentes

Tri par pixel

Idée : Si plusieurs points ont le même abcisse seul deux sont sur l'enveloppe.

Corps

ATTENTION : Ne pas oublier qu'un repère orthonormé en informatique a l'axe des y inversé (plus on est haut sur l'écran plus le y est proche de 0, plus ou est vers la bas plus on est loin de 0 **EN POSITIF**)

```
1  ymin, ymax = table de hachage de forme <Entier, Chaine de caractère>
2  res = liste de points vide
3
4  Pour tout point P dans Points:
5      Si ymin[P.x] == NULL OU ymin[P.x].y > P.y alors:
6          ymin[P.x] = P
7
8  Pour tout point P dans Points:
9      Si ymax[P.x] == NULL OU ymax[P.x].y < P.y alors:
10         ymax[P.x] = P
11
12 Mettre ymax dans l'ordre décroissant des abcisses
13
14 res = ymin + ymax
15
16 retourner res
```

Coût

- 1 : $O(2)$
- 2 : $O(1)$
- 4 - 6 : $O(n)$
    - 5 - 6 : $O(1)$
- 8 - 9 : $O(n)$
    - 9 - 10 : $O(1)$
- 12 : $O(a)$ avec *a* la taille de ymax
- 14 : $O(a+b)$ avec b taille de ymin
- 16 : $O(1)$

Précisons que $a < n,\ b < n,\ a + b \leq n$

Donc $O(4 + n + n + a + (a + b)) = O(n)$

Implémentation

Une implémentation est diponbile ICI

Filtre d'Akl-Toussaint

Corps

Idée : Tracer un rectangle reliant les 4 points cardinaux du nuage de points (Les 4 points aux extrémités négatives et positives en abcisse et ordonnée)

```
Pour tout point p de Points:
    Nord = Point le plus haut du nuage de point
    Est = Point le plus a gauche du nuage de point
    Sud = Point le plus bas du nuage de point
    Ouest = Point le plus a droite du nuage de point

Pour tout point P de Points:
    Si P contenu dans triangle (Nord, Est, Sud) OU P contenu dans triangle (Nord, Ouest, Sud):
        Points.remove(P)

Retourner Points
```

Coût

- 1 - 5 : $O(n)$
- 7 - 9 : $O(n)$

Donc au total $O(n+n) = O(n)$

Implémentation

```java
private ArrayList<Point> exercice3(ArrayList<Point> points){
    if (points.size()<4) return points;

    Point ouest = points.get(0);
    Point sud = points.get(0);
    Point est = points.get(0);
    Point nord = points.get(0);
    for (Point p: points){
        if (p.x<ouest.x) ouest=p;
        if (p.y>sud.y) sud=p;
        if (p.x>est.x) est=p;
        if (p.y<nord.y) nord=p;
    }
    ArrayList<Point> result = (ArrayList<Point>)points.clone();
    for (int i=0;i<result.size();i++) {
        if (triangleContientPoint(ouest,sud,est,result.get(i)) ||
                triangleContientPoint(ouest,est,nord,result.get(i))) {
            result.remove(i);
            i--;
        }
    }
    return result;
}
private boolean triangleContientPoint(Point a, Point b, Point c, Point x) {
    double l1 = ((b.y-c.y)*(x.x-c.x)+(c.x-b.x)*(x.y-c.y))/(double)((b.y-c.y)*(a.x-c.x)+(c.x-b.x)*(a.y-c.y));
    double l2 = ((c.y-a.y)*(x.x-c.x)+(a.x-c.x)*(x.y-c.y))/(double)((b.y-c.y)*(a.x-c.x)+(c.x-b.x)*(a.y-c.y));
    double l3 = 1-l1-l2;
    return (0<l1 && l1<1 && 0<l2 && l2<1 && 0<l3 && l3<1);
}
```

Problème de l'enveloppe convexe

Algorithme naïf

Idée : Pour chaque paire de points dans Points vérifier si elle forme un côté de l'enveloppe conexe en vérifiant que tous les autres points de Points sont du même côté du segment.

Corps

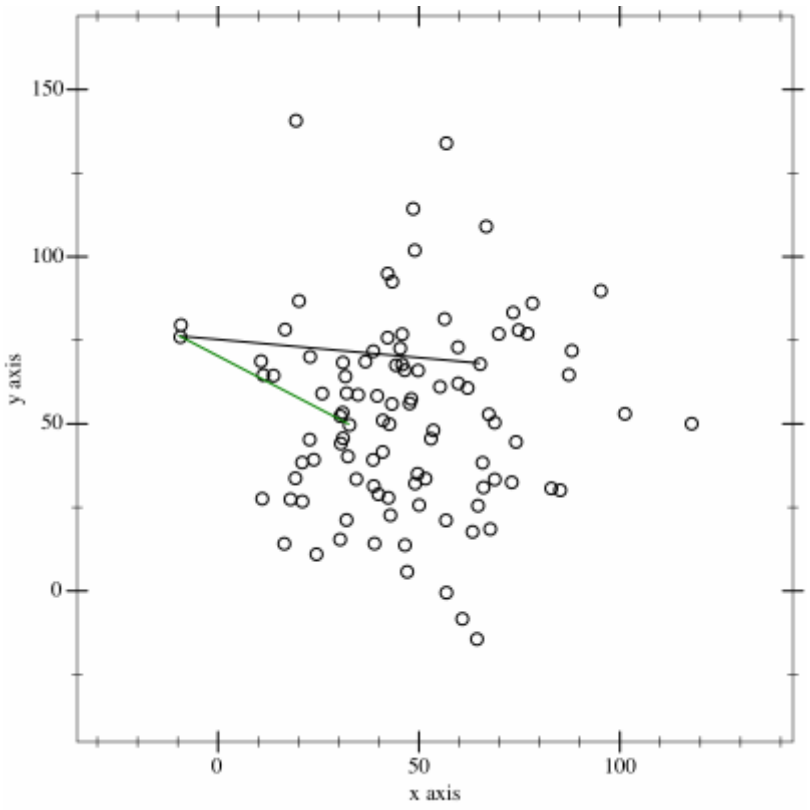
```
A = Un point aléatoire de Points
B = Point en 0, 0
res = liste de point
res.add(A)
Tant que (B != A):
    Pour tout point P de Points:
        distMax = -infini
        Pour tout point Q de Points:
```

```
            Vérfier que Q est du même côté de [AP]
        Si tous les Q sont du même côté de [AP]:
            Si A.distance(P) > distMax :
                distMax = A.distance(P)
                B = P
     res.add(B)
Retourner res
```

Coût

D'après le cours cet algorithme coûte $O(n^3)$

Algorithme de Jarvis



Corps

```
res = liste de point vide
P = point d'abcisse minimum
res.add(P)
Q = Point en 0, 0
angle = +infini
R = Point en 0, 0

Pour tout point p de Points:
    Si [Pp] forme un côté de l'enveloppe Connexe :
        Q = p
        Break
res.add(Q)
Tant que R != P:
    Pour tout point i de Points:
        Si l'angle PQi < angle:
            angle = angle PQi
            R = i
    res.add(R)
    P = Q
```

```
    Q = R
Retourne res
```

Coût

dans le pire des cas on est en $O(n^2)$

Implémentation

Une implementation est disponible ICI

Algorithme de Graham

Corps

```
Points = TriParPixel(Points)

Pour i entre 0 et len(Points):
    Si produit vectoriel de (Points[i], Point[i+1]) et (Points[i], Points[i+2]) < 0:
        Alors retirer Points[i+1] et retourner un cran en arrière

Retourner Points
```

Coût

D'après wikipedia c'est $O(n \log n)$

Voici l'explication complète :

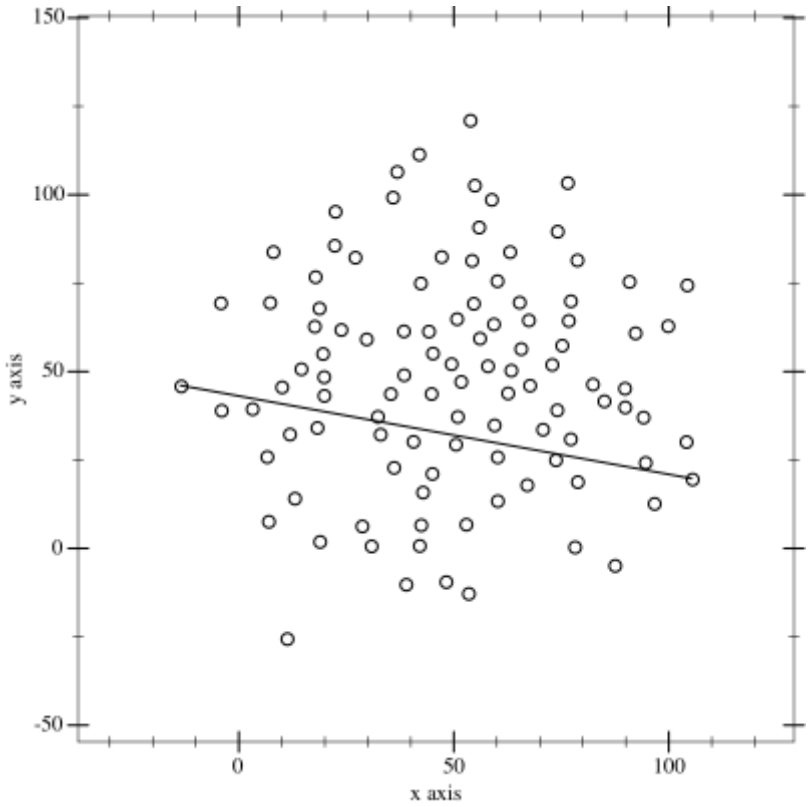
> La complexité en temps du choix du pivot est en Θ(n), n étant le nombre de points de l'ensemble. Le tri des points peut se faire avec une complexité en temps en O(n log n). La complexité de la boucle principale peut sembler être Θ(n2), parce que l'algorithme revient en arrière à chaque point pour évaluer si l'un des points précédents est un « tournant à droite ». Mais elle est en fait en O(n), parce que chaque point n'est considéré qu'une seule fois. Ainsi, chaque point analysé ou bien termine la sous-boucle, ou bien est retiré de T et n'est donc plus jamais considéré. La complexité globale de l'algorithme est donc en O(n log n), puisque la complexité du tri domine la complexité du calcul effectif de l'enveloppe convexe.

Implémentation

Voici une implémentation qui marche pas trop trop : ICI.

Algorithme de QuickHull

Idée : Prendre le principe d'Akl-Toussaint mais aggrandir de manière incrémental la zone de suppression



Corps

```
QuickHull(Points):
Faire passer Points dans Akl-Toussaint

A = Est
B = Ouest

Tant que Points non vide:
    Appeler récursivement Hull(A,B, Points)

Retourner Points
```

```
Hull(A, B, Points):
Pour tout point P de Points:
    X = P le plus éloigné de (A, B)

Pour tout point P de Points:
    Si P contenu dans triangle (A, X, B):
        Points.remove(P)
Retourner AX, BX, Points
```

Coût

D'après wikipédia cet algorithme est de compléxité $O(n^2)$ dans le pire cas et de $O(n \log n)$ en moyenne.

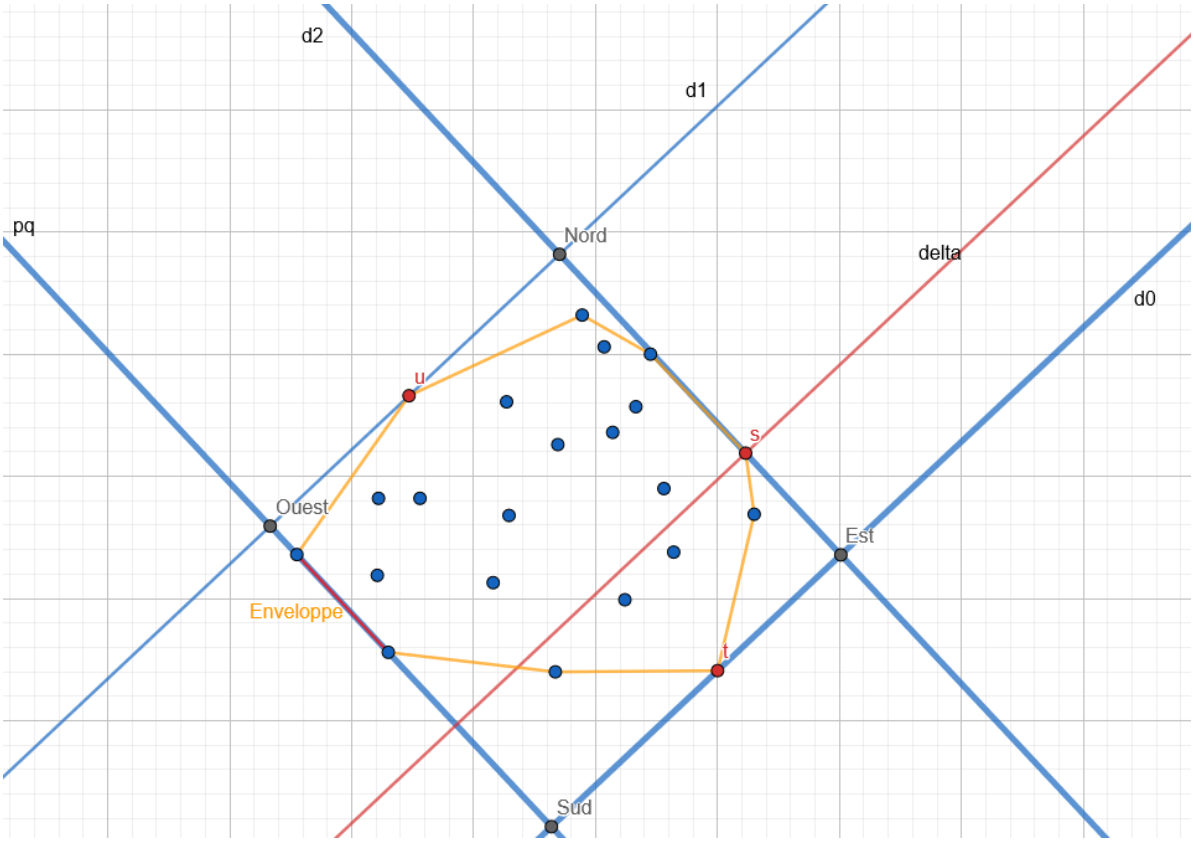
Implémentation

Une implémentation est disponible ICI

Problème du rectangle couvrant minimum

Algorithme naïf

Corps

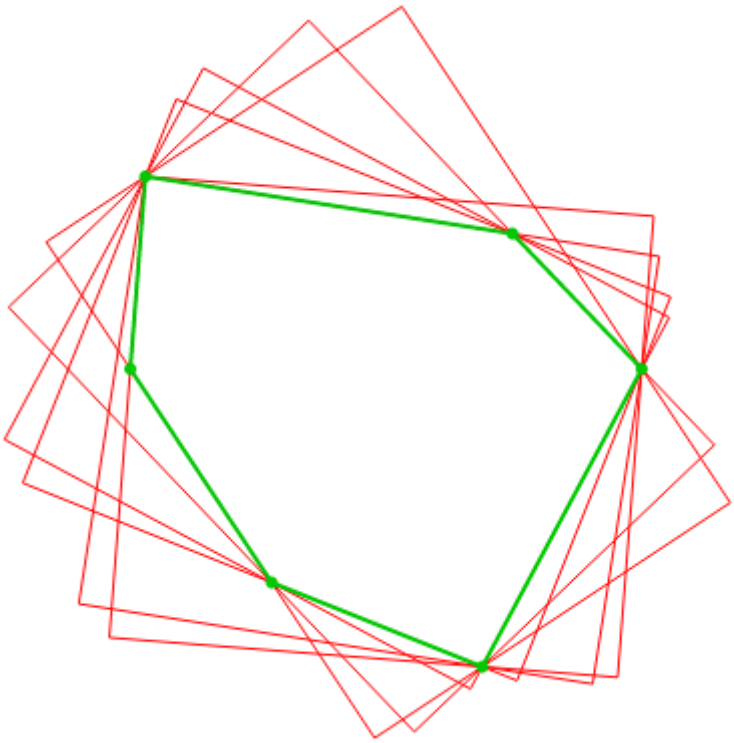



```
enveloppe = enveloppe convexe de Points
aireMin = +infini
rect = liste de sommet vide
Pour tout côté pq de enveloppe :
    s = coin de l'enveloppe le plus loin de (pq)
    delta = droite passant par s, orthogonale à (pq)
    t, u = coins de l'enveloppe le plus loin de delta dans les deux demi-plans définis par delta
    d0 = droite passant par t, orthogonale à (pq)
    d1 = droite passant par u, orthogonale à (pq)
    d2 = droite passant par s, parallèle à (pq)
    Nord, Est, Sud, Ouest = Intersections des droites (pq), (d0), (d1) et (d2)
    Si aire(Nord, Est, Sud, Ouest) < aireMin:
        aireMin = aire(Nord, Est, Sud, Ouest)
        rect = [Nord, Est, Sud, Ouest]
Retourner rect
```

Coût

- Selon le cours $O(n^2)$

Algorithme de Shamos



Idée : Prendre un "pied a coulisse" pour représenter le rectangle couvrant minimum tournant autour de l'enveloppe convexe.

Corps

```
enveloppe = enveloppe convexe de Points
W = point le plus à l'ouest de enveloppe
E = point le plus à l'Est de enveloppe
left = droite verticale passant par W
right = droite parralèle à left passant par E
maintenir ce parralélisme tout le long le l'algo
répéter jusqu'à ce que la condition (left passe par E ou right passe par W)
se réalise deux fois :
    next = côté de enveloppe formant angle min avec left
    box = rectangle min contenant enveloppe passant par next
    pivoter left et right pour l'un des deux contiennent next
retourner le min des valeurs de box
```

Coût

D'après le cours : $O(n^2)$

Algorithme de Toussaint

Corps

```
enveloppe = enveloppe convexe de Points
Nord, Est, Sud, Ouest = Les points cardinaux de enveloppe
aireMin = +Infini
rect = rectangle
construire les droites dN, dE, dS et dO passant respectivement par Nord, Est, Sud et Ouest
Pour tout côté pq de enveloppe :
    alpha = angle minimum formé par l'une des droites et l'un des côtés de enveloppe
    faire effectuer aux 4 droites une rotation selon l'angle alpha
    RECTANGLE = rectangle formé alors par dN, dE, dS et dO
    Si aire(RECTANGLE) < aireMin :
        aireMin = aire(RECTANGLE)
        rect = RECTANGLE
retourner rect
```

Coût

- 1 : Ici nous admettons une complexité de $O(n \log n)$ en utilisant Graham
- 2 - 5 : Constructions en temps constant $O(1)$
- 6 - 12 : $O(n)$

Donc : $O(n)$

Implémentation

Une implémentation est disponible ICI

Problème du diamètre optimisé

Problème du diamètre d'un ensemble de points

Entrée : *Points* un ensemble de points

Sortie : *Diamètre* liste de deux points de distance maximum

Paire antipodales

Idée : Une paire { A , B } de points de Points est antipodale s'il existe 2 droites parallèles passant par A et B telles que la bande du plan entre ces deux droites contient tous les points de Points.

Propriété : le diamètre de Points est la distance maximum entre deux points d'une paire antipodale de Points

Propriété : il y a au plus 2·n (plus quelques) de paires antipodales d'un ensemble Points de n points.

Implémentation

```
  public ArrayList<Line> calculPairesAntipodales(ArrayList<Point> points){
    ArrayList<Point> p = algoJarvis(points);
    int n = p.size();
    ArrayList<Line> antipodales = new ArrayList<Line>();
    int k = 1;
    while(distance(p.get(k), p.get(n-1), p.get(0)) < distance(p.get((k+1)%n), p.get(n-1), p.get(0))) ++k;
    int i = 0;
```

```
    int j = k;
    while (i<=k && j<n){
      while (distance(p.get(j), p.get(i), p.get(i+1)) <
distance(p.get((j+1)%n), p.get(i), p.get(i+1)) && j<n-1){
        antipodales.add(new Line(p.get(i), p.get(j)));
        ++j;
      }
      antipodales.add(new Line(p.get(i), p.get(j)));
      ++i;
    }
    return antipodales;
  }
```

```
  public Line trouverDiametreOpti(ArrayList<Point> points){
    if (points.size()<3) {
      return null;
    }

    ArrayList<Line> pa = calculPairesAntipodales(points);
    double taille_max = Double.MIN_VALUE;
    Line diametre = new Line(new Point(), new Point());

    for (Line l : pa){
      if(l.getP().distance(l.getQ()) > taille_max){
        taille_max = l.getP().distance(l.getQ());
        diametre = l;
      }
    }

    return diametre;
  }
```