**Biblio :** Cormen, Intro à l'algorithmique

**Prérequis :**

- Complexité asymptotique en pire cas
- Notation $O(), \theta(), \Omega()$
- 
  - $\theta(\ln n)$ : algo (poly)logarithmique
  - $\theta(n)$ : algo linéaire (min ou max d'un tableau)
  - $\theta(n \ln n)$ : algo quasi linéaire (tri par tas)
  - $\theta(n^2)$ : algo quadratique (tri par insertion, tri bulle)
  - $\theta(n^3)$ : algo cubique (calcul de tous les plus courts chemins dans un graphe)
  - $\theta(2^n)$ : voyageur de commerce (pas sur, peut être en $\theta(n^k)$)
  - $\theta(n!)$ : Bogo-sort

**Ce qu'on va faire :**

- Analyse des algos récursifs
- Analyse amortie

**Fonction** $f(X)$ : $\begin{cases} \text{si } \alpha(X) \text{ alors val} \\ \text{sinon } f(X') \end{cases}$

**Simplification :** Pour analyser le temps d'exécution d'une fonction récursive, on va essayer de résumer à l'aide d'un seul entier $n$ le contexte $X$ de façon à ce que $n$ (correspondant généralement à la taille associée au contexte) décroit à chaque appel de $f$.

**Exemple 1 :**
**Fonction F(m : entier) : entier** $\theta(1)$
**Debut**
**Si m** $= 0$ $\theta(1)$ **alors retourner** $\varnothing$ $\theta(1)$
**Sinon retourner m*F(m-1)** $\theta(1) + T(m-1)$
**Fin**

En pire cas :

- $T(0) = \theta(1)$
- $T(n) = \theta(1) + T(n-1)$
- Suite arithmétique de premier terme 1 et de raison 1 donc $T(n) = \theta(n+1) = \theta(n)$
- Donc factorielle s'exécute en temps linéaire

**Exemple 2 :**

**Fonction Max(V:Tableau[] de Elem, a,b : entier) : Elem** $\theta(1)$

**Debut**

**Si a = b alors retourner V[a]** $\theta(1)$

**Sinon retourner Max2(V[a],Max(V,a+1;b))** $\theta(1)+T(n-1)$

**Fin**

En pire cas :

- $n = b - a$ (car $b$ et $a$ désignent les indices du tableau entre lesquels on cherche le max)
- Question : cout du passage de paramètre (par adresse : constant, par valeur : $n$)
- $T(0) = \theta(1)$
- $T(n) = \theta(1) + \theta(n-1)$
- Suite arithmétique de premier terme 1 et de raison 1 donc $T(n) = \theta(n+1) = \theta(n)$
- Donc Max s'exécute en temps linéaire

**Exemple 3 :** Recherche dichotomique récursive

**Fonction Dicho(V:Tab[]; a,b:entier; x: Elem) :booléen** $\theta(1)$

**Var m: entier**

**Debut**

**Si a = b alors retourner (x=V[a])** $\theta(1)$

**Sinon**
**m** $\leftarrow \frac{a+b}{2}$
**Si x** $\leq$ **V[m] alors retourner Dicho(V,a,m,x)** $T(\frac{n}{2})$
**Sinon retourner Dicho(V,m+1,b,x)** $T(\frac{n}{2})$
**FinSi**
**FinSi**
**Fin**

En pire cas :

- $n = b - a$
- $T(0) = \theta(1)$
- $T(n) = \theta(1) + T(\frac{n}{2})$
- Donc $T(n) = \theta(\ln n)$

**Exemple 4 :** Fibonacci
**Foncion Fib(m:entier) :entier**
**Debut**
**Si m = 0 alors retourner 0**
**Sinon si m = 1 alors retourner 1**
**Sinon retourner Fib(m - 1) + Fib(m - 2)**
**Fin Si**
**Fin**

En pire cas :

- Le nombre d'appels à Fib est exactement le résultat Fib(m) + ou - 1

- Avec utilisation de l'exemple : Fib(m) << Myst(m)
- $T(0) = T(1) = \theta(1)$
- $T(n) = \theta(1) + T(n-1) + T(n-2)$
- Or Fib(n)/Fib(n-1) tend vers le nombre d'or $\phi$

**Exemple intermédiaire :** **Fonction myst(m:entier):entier**
**Debut**
**Si m = 0 retourner 1**
**Sinon retourner myst(m-1) + myst(m-1)**
**FinSi**
**Fin**

- $T(0) = \theta(1)$
- $T(n) = \theta(1) + 2T(n-1)$
- $T(n) = 1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1 = \theta(2^n)$

# Algo N$^o$ 3 Analyse des récurrences linéaires

**Récurrence linéaire :**

$$T(n+p+1) = f(n) + \sum_{k=0}^{p} a_k T(n+k)$$

$$T(n+1) = aT(n) + 1$$

$$T(n+2) = aT(n+1) + bT(n) + 1$$

**Résolution du cas d'ordre 1 :**

$$T(n) = \begin{cases} n+1 = \theta(n) \text{ si } a = 1 \\ \frac{a^{n+1}-1}{a-1} = \theta(a^n) \end{cases}$$

**Exemple 5 :** Hanoï

**Fonction Hanoi(n,i,j,k)**
**Debut**
**Si n $>$ 0 alors**
**Hanoi(n-1,i,j,k)**
**Affiche("Je déplace de i vers j")**
**Hanoi(n-1,k,j,i)**
**FinSi**
**Fin**

Temps de calcul :

- $T(0) = \theta(1)$

- $T(n) = \theta(1) + 2T(n-1)$
- Donc $T(n) = \theta(2^n)$

**Résolution du cas d'ordre 2 :** On simplifie le problème en faisant abstraction du 1 : On considère $T(n+2) = aT(n+1) + bT(n)$ ie $T(n+2) - aT(n+1) - bT(n) = 0$.
On pose $P(X) = X' - aX - b$. Il a deux racines $r_1$ et $r_2$ et la solution s'écrit sous la forme $\alpha r_1^n + \beta r_2^n$ (il suffit de le vérifier).

$$T(n) = \theta(r_1^n)$$

**Fin du cas d'ordre 2 :** Le 1 ajoute une fois Fibonacci à la fin, donc ne change pas le $\theta$

**Fin de l'exemple 4 :** Fibonacci

- $P(X) = X^2 - X - 1$ donc $r_1 = \frac{1+\sqrt{5}}{2} = \phi$ et $r_2 = \frac{1-\sqrt{5}}{2}$ donc $r_2^n$ tend vers 0.
- $T(n) = \theta(\phi^n)$

**Exemple 6 :**
**Fonction Dummy(n:entier) :entier**
**Si n $\leq$ 3 retourner 1**
**Sinon retourner Dummy(n-2) + Dummy(n-4)**
**Fin Si**
**Fin**

Solution :

- Le polynome est de la forme $X^4 - X^2 - 1$

**Complexité des algos "diviser pour régner" :** Cas $T(n) = aT(\frac{n}{b}) + f(n)$ avec $a \geq 1$ , $b > 1$ et $T(0) = \theta(1)(= 1)$ (on peut remplacer $\frac{n}{b}$ par sa partie entière inférieure ou supérieure)

**Propriété :**

- Si $f(n) = O(n^{\log_b a - \varepsilon})$ $(\varepsilon > 0)$ alors $T(n) \in \theta(n^{\log_b a})$
- Si $f(n) = \theta(n^{\log_b a})$ alors $T(n) = \theta(n^{\log_b a} \ln n)$
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ $(\varepsilon > 0)$ et $af(\frac{n}{b}) \geq cf(n)$ pour $n$ assez grand et $c$ une constante $< 1$. Alors $T(n) \in \theta(f(n))$

**Exemple 7 :** La recherche dichotomique

- $T(n) = T(\frac{n}{2}) + 1$
- On est dans le deuxième cas, donc $T(n) = \theta(\ln n)$

**Exemple 8 :** Tri fusion

**Fonction TriFusion(V:Tab,n:entier)**
**Var** $n_1, n_2$ **: entier**
**Var** $V_1, V_2$ **: Tab**
**Debut**
**Si** $n \geq 2$ **Alors**
**Diviser**$(V, n, V_1, n_1, V_2, n_2)$ $(\theta(n))$
**TriFusion**$(V_1, n_1)$
**TriFusion**$(V_2, n_2)$

**Fusionner**$(V_1, n_1, V_2, n_2, V)$ $(\theta(n))$
**FinSi**
**Fin**

Temps d'exécution :

- $T(n) = 2T(\frac{n}{2}) + \theta(n)$
- $f(n) = \theta(n)$ donc on est dans le deuxième cas
- $T(n) = \theta(n \ln n)$

**Exo 1 :** Soit la fonction :

```
void tri3(int* V, int i, int j)
{
if (V[i] > V[j])
{
int tmp = V[i]; V[i] = V[j]; V[j] = tmp;
}

if (i+1 ≥ j) return;
int k = (j-i+1)/3;
tri3(T,i,j-k);
tri3(T,i+k,j);
tri3(T,i,j-k);
}
```

1. Montrer qu'il trie correctement le tableau
2. Temps d'exécution asymptotique de tri3
3. Est il meilleur que trifusion et autres

1. 
   - Montrons que tri3 fonctionne pour $n \leq 3$
     - $n = 1$ le tableau est déjà trié
     - $n = 2$ Si il n'est pas trié, l'échange se fait, sinon rien, et ça sort

- $n = 3$ $i = 0, j = 2, k = 1$. Au premier tour on tri les 2 premières valeurs, puis les 2 dernières, et encore les 2 premières. Donc c'est bon.

- Montrons que tri3 fonctionne jusqu'à $n$ alors tri3 fonctionne pour $n + 1$
  - Simplifions en supposant $n + 1$ multiple de $k$
  - Au premier stade, la zone $i..j - k$ est triée ($V(l) < V(m)$), donc sur $i + k..j - k$ on a des valeurs supérieures aux valeurs dans $i..i + k$
  - Au second stade, la zone $i + k..j$ est triée ($V(l) < V(m)$), donc les valeurs dans $j-k..j$ sont supérieures aux valeurs dans $i+k..j-k$ et dans $i..i+k$ et elles sont triées.
  - On trie encore le début du tableau.

2. Complexité asymptotique de tri3 ?

- $n = j - i + 1$
- $T(n) = \theta(1) + 3T(\frac{2}{3}n)$
- Cas 1 du théorème avec $b = \frac{3}{2}$
- Donc $T(n) = \theta(n^{\frac{\ln 3}{\ln 3 - \ln 2}}) = \theta(n^{2,71})$

# Algo N$^o$ 6 — Analyse asymptotique expérimentale

**Principe :** On mesure de façon expérimentale le temps d'exécution de 2 programmes pour des valeurs de $n$ données, avec $T_1(n) = n^2$ et $T_2(n) = 1000n$ :

| $n$ | 10 | 20 | 40 | 80 | 160 |
|---|---|---|---|---|---|
| $T_1$ | | | | | |
| $T_2$ | | | | | |

**Idée :** On va tracer les courbes dans un repère utilisant une échelle logarithmique. Alors la fonction $an^b$ devient $\log_2(a) + b\log_2(n)$

Preuve : $\log(T(n)) = \log(an^b) = \log a + \log n^b = \log a + b\log(n)$

**Pour les complexités en $an^b\log(n)$ :** Poser $T_2(n) = \frac{T(n)}{n^b}$ et mettre l'échelle logarithmique sur l'axe des abscisses.

**Principe :** Calculer le coût moyen de chaque opération dans le pire cas de séquence d'opérations.

**3 techniques standard :**

- Méthode par agrégat
- Méthode comptable
- Méthode du potentiel

**Méthode par agrégat :** On montre que pour toute suite d'opérations, le temps total est $T(n)$ dans le pire cas et le cout amorti (= cout moyen) de chaque opération est $\frac{T(n)}{n}$.

**Exemple : La pile**

On a 3 opérations sur les piles :

- Empiler(S,x) $\theta(1)$
- Depiler(S) $\theta(1)$
- MultidepilerS,k $\theta(min(k,s))$ ($s$ =Card$S$)

L'utilisateur faire une série de $n$ opérations parmi ces 3 là.

- $n$ fois empiler,depiler $\theta(n)$
- $n$ Multidepiler Cout borné par les $k$ choisis et le nombre d'éléments de la pile

- $\frac{n}{2}$ empiler et $\frac{n}{2}$ multidépiler Donc $\frac{n}{2}\theta(n) = O(n^2)$

En réalité, on empile et dépile $n$ fois quoi qu'il arrive donc la complexité en pire cas est $\theta(n)$

Cout amorti de chaque opération = Cout total / $n = \theta(1)$

**Exemple : Le compteur de bits**

On a un tableau $A[0..k-1]$ de bits. Donc $A$ représente un entier $x = \sum_{i=0}^{k} A[i]2^i$ qu'on appelle le compteur.

On a l'opération incrémenter qui ajoute 1 à $x$

```
Action Incrémenter(InOut A:Tableau[0..k-1] de bits)
  var i:entier
Debut
  i <- 0
  TantQue i<k et A[i]=1 Faire
    A[i] <- 0
    i <- i + 1
  FinTantQue
  Si i < k Alors A[i] <- 1
  FinSi
Fin
```

On compte le nombre de fois que chaque bit est modifié

Cout total $= n\sum_{i=0}^{k'}(\frac{1}{2})^{k'}$ avec $k' = \log_2 n$ donc Cout total $\leq 2n$

Le cout amorti est donc $\theta(1)$

**Exercice 5 :**

```
Action Defiler
Debut
  Si P2 est vide Alors
    Empiler tous les éléments de P1 sur P2 (sauf le dernier)
    Depiler P1
  Sinon
    Depiler P2
  FinSi
Fin

Action Enfiler
Debut
  Empiler P1
Fin
```

On effectue $n$ opérations : $E$ l'ensemble des $e_i$ les enfilements, et $D$ l'ensemble de $d_i$ les défilements.

Cout total $= \sum e_i + \sum d_i$.

On a $\sum e_i \leq n$ car le cout d'un $e_i$ est 1.

Une fois qu'on a dépilé $P_1$ avec un cout égal à la taille de $P_1$, il se passe exactement taille de $P_1$ opérations avec un cout de 1, puis on recommence ce cycle.

$\sum d_i = tP1 + \sum_{k=1}^{tP1} 1 + t'P1 + \sum_{k=1}^{t'P1} 1 \ldots \leq 2n$

Conclusion : Cout total $\leq 3n$. Donc Cout total est $O(n)$.

De plus, si on ne fait que des empilements le cout est $\theta(n)$. Donc Cout total est $\theta(n)$.

Donc le cout amorti est $\frac{\theta(n)}{n} = \theta(1)$

## Algo N$^o$ 9 Structures de données pour ensembles disjoints

**Définition :** On appelle ensembles disjoints une collection $S = S_1, \ldots, S_n$ d'ensembles deux à deux disjoints. On va effectuer des opérations sur ces ensembles (ex : union, déterminer si un élément est dans un des ensembles, si deux ensembles sont contenus dans un autre ensemble etc).

**Utilisations classiques :**

- Calculer les composantes connexes d'un graphe (fermeture transitive de la relation d'adjacence)
- Calculer des classes d'équivalence de façon générale
- Déterminer si un graphe contient un cycle (algo de Kruskal)

**Opérations sur ces structures :** Les éléments des ensembles sont appelés "objets" (pointeur vers un élément avec des informations) :

- Creer-ensemble(x:objet) : Créer un ensemble réduit au singleton x
- Trouver-ensemble(x:objet) : retourne un objet qui est le représentant de tous les objets du meme ensemble.
- Union(x,y:objet) avec x $\neq$ y et Trouver-ensemble(x) $\neq$ Trouver-ensemble(y). Attribue un nouveau représentant et réalise l'union.

- On pourrait rejouter des opérations de division d'ensemble mais ça ne présente aucune difficulté.

**Exemple :** Calcul de composantes connexes. On voudrait savoir si deux sommets d'un graphe sont dans la même composante connexe (ie existe-t-il un chemin entre ces deux sommets), combien il y a de composantes connexes dans le graphe, etc.

Fichier : Document sans nom 1

```
Action Composantes-Connexes(E G:graphe non orienté)

var u,v:sommet; cptComposante = 0;

Debut
  Pour tout sommet u de G Faire
    Creer-Ensemble(u)
    cptComposante++;
  FinPour
  Pour toute arête (u,v) de G Faire
    Union(Trouver-ensemble(u),Trouver-ensemble(v));
    cptComposante--;
  FinPour
Fin



Fonction Meme-Composante-Connexe(E u,v:sommet) : booléen

Debut
  Retourner Trouver-ensemble(u) = trouver-ensemble(v)
Fin
```

Le cout est donc n Creer-ensemble + 2m Trouver-ensemble + min(n,m) Unions

**Représentation par listes chaînées :** On représente chaque ensemble par une liste chainée enrichie.

- Creer-ensemble est en $\theta(1)$
- Trouver-ensemble est en $\theta(1)$

- Union : pour pouvoir fusionner la plus petite à la grande, il nous faut rajouter une donnée : le nombre d'éléments de chaque liste. Cela permet de réduire la complexité au point où $n$ creer-ensemble et $m$ opérations quelconques, alors le coût amorti total est de $\theta(m + n\ln n)$ On le démontre avec la méthode des agrégats.

**Preuve :**
La complexité totale correspond au nombre de fois où le lien "représentant" est modifié.
Pour un objet x, son représentant est modifié lors d'une union où x était dans le plus petit ensemble.
Au début, x est dans un ensemble à un élément.
Si son représentant est modifié une fois alors il est dans un ensemble à au moins 2 éléments.
Si son représentant est modifié une deuxième fois alors il est dans un ensemble à au moins 4 éléments.
La taille double à chaque fois, d'où le $\ln n$
Et il y a n éléments, donc on a au maximum $n\ln n$ changements de représentants.

**Représentation par forêt :** Un ensemble a un représentant unique qui va être la racine d'un arbre. Tous les éléments d'un ensemble vont être dans le même arbre, et on a un arbre par ensemble. La forêt est l'union disjointe des arbres. Chaque élément, en plus de sa valeur, stockera un pointeur vers son père dans l'arbre (ou lui même si c'est la racine), ainsi qu'u entier appelé rang.

```
Creer-Ensemble(x:objet)
Debut
  x.pere <- x;
  x.rang <- 0;
Fin

fonction Trouver-Ensemble(x:objet) : objet
var y : objet
Debut
  y <- x;
  Tant que y.pere != y Faire
    y <- y.pere;
  FinTantQue
  Retourner y;
Fin

Union-Ensemble(x,y:objet)
Debut
  x <- Trouver-Ensemble(x);
  y <- Trouver-Ensemble(y);
  Si (x != y) Alors
    x.pere <- y
    y.rang ++
  FinSi
Fin
```

## Idées :

- Union par rang : on connecte l'arbre qui a le plus petit rang à l'autre.

```
Union-Ensemble(x,y:objet)
Debut
  x <- Trouver-Ensemble(x);
  y <- Trouver-Ensemble(y);
  Si (x != y) Alors
    Si (x.rang > y.rang) alors y.pere <- x
    Sinon x.pere <- y
    Si (x.rang = y.rang) alors y.rang++
  FinSi
Fin
```

- Compression de chemin : On profite d'un appel à Trouver-

Ensemble pour mettre à jour tous les pères sur le chemin de l'objet jusqu'à sa racine.

```
Trouver-Ensemble(x:objet) : objet
Debut
  Si x = x.pere alors retourner x
  Sinon
    x.pere <- Trouver-Ensemble(x.pere)
    retourner x.pere
  FinSi
Fin

Union-Ensemble(x,y:objet)
Debut
  x <- Trouver-Ensemble(x);
  y <- Trouver-Ensemble(y);
  Si (x != y) Alors
    Si (x.rang > y.rang) alors y.pere <- x
    Sinon x.pere <- y
    Si (x.rang = y.rang) alors y.rang++
  FinSi
Fin
```

**Théorème de Tarjan (1975) :** Si on utilise ce type de structure Union-Find avec union par rang et compression de chemin, alors une séquence arbitraire de m opérations Creer-Ensemble, Trouver-Ensemble et Union-Ensemble prend un temps en $\theta(m\alpha(n))$ où $\alpha(n)$ croit très lentement (impossible de représenter en mémoire un entier $n$ qui rend $\alpha(n) \leq 4$).

**Explication :** $\alpha$ est l'inverse d'une fonction qui croit très vite : $A_k(1)$ ... $\alpha(n) =$ plus petit $k$ tel que $A_k(1) \geq n$. Avec $A_k$ la fonction de Ackermann : qui à $k \geq 0, j \geq 1$ associe $A_k(j) = \begin{cases} j+1 \text{si} k = 0 \\ A_{k-1}(A_{k-1}(\ldots A_{k-1}(j)))) \text{sinon} \end{cases}$

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A_0(j)$ | 2 | 3 | 4 | 5 | 6 | 7 |
| $A_1(j)$ | 3 | 5 | 7 | 9 | 1 | 13 |
| $A_2(j)$ | 7 | 23 | 63 | ... | | |
| $A_3(j)$ | 2047 | $> 2^{2^{27}}$ | ... | | | |
| $A_4(j)$ | $>> 2^{2048}$ | ... | | | | |

**Conclusion :** Union-Find peut être considéré comme linéaire.

**Exercice :** Quel est le rang max que l'on peut obtenir avec 7 unions sur 8 éléments ? Réponse : 3 (en réalité pour faire un rang $k$ il faut $2^k$ éléments. On voit donc qu'il y a majoration par $m \ln n$.

**Exercice :** Si on fait trouver-Ensemble sur un sommet de profondeur $k$, quelle est la variation de la somme des profondeurs de l'arbre ?

**Objectif :** Etudier les algortithmes pour résoudre des problèmes géométriques.

**Applications :**

1. CAO
2. Jeux vidéos en 2D et 3D
3. Résistance des matériaux
4. Robotique
5. Réseaux

**Géométrie euclidienne :** Points et vecteurs du plan ou de l'espace. Cela suppose qu'on dispose d'un type nombre réel (problème : sensibilité aux perturbations).

**Solutions à ces problèmes :**

1. Méthodes algébriques
2. Utiliser des opérations "pas dangereuses" (addition, soustraction), pour la multiplication il faut une précision doublée, la division est dangereuse
3. Utiliser des nombres à précision arbitraire (fonctionne mais ralentit le code d'un facteur 100 à 200)

4. Utiliser seulement le signe plutôt que la valeur

5. Ne faire des calculs qu'en cas de "doute" (calcul paresseux)

6. Tout faire avec des entiers (géométrie discrète)

Il existe des bibliothèques standard pour faire ça (ex : CGal en C++)

```
Fonction Segments-Intersection(E: p1,p2,p3,p4 : Points) : booléen

Var d1,d2,d3,d4 : Réels

Debut
  d1 <- orientation(p3,p4,p1)
  d2 <- orientation(p3,p4,p2)
  d3 <- orientation(p1,p2,p3)
  d4 <- orientation(p1,p2,p4)

  Si [ ((d3 > 0 et d4 < 0) ou (d3 > 0 et d4 < 0)) et ((d1 > 0 et d2 <
  Alors retourner Vrai;
  Sinon si d1 = 0 et Sur-Segment(p3,p4,p1) Alors Retourner Vrai
  Sinon si d2 = 0 et Sur-Segment(p3,p4,p2) Alors Retourner Vrai
  Sinon si d3 = 0 et Sur-Segment(p1,p2,p3) Alors Retourner Vrai
  Sinon si d4 = 0 et Sur-Segment(p1,p2,p4) Alors Retourner Vrai

Fin

Fonction Sur-Segment(E: p,q,r : Points) : booléen

Debut

  Retourner (min(p.x,q.x) <= r.x <= max(p.x,q.x) et min (p.y,q.y) <= r

Fin
```

**Définition des éléments de base dans le plan :**

1. typedef struct  double x; double y;  Point;

2. typedef struct  Point O; Point E;  Segment; C'est l'ensemble des points entre O et E $\rightarrow$ comment faire ça précisément ?  On utilise les combinaisons linéaires sur O et E : $P =$

$\alpha O+(1-\alpha)E$ avec $\alpha \in [0;1]$ ie $[OE] = \{\alpha O+(1-\alpha)E \mid \alpha \in [0;1]\}$ cet ensemble est appelé ensemble des combinaisons convexes.

3. Longueur du segment $[OE]$ = distance euclidienne entre $O$ et $E$ = norme de $\vec{OE} = \sqrt{(E.x - O.x)^2 - (E.y - O.y)^2}$. En géométrie algorithmique, on privilégie la distance au carrée (pour les erreurs et le temps de calcul).

**Question typique :** Etant donné un point $P$ et une droite $(P_1P_2)$, le point est il sur la droite, à sa gauche et à sa droite ? On utilise le déterminant de $\vec{P_1P_2}, \vec{P_1P}$ (positif : $P$ est à gauche, négatif : $P$ est à droite, nul : $P$ est sur la droite.) Cela s'appelle la fonction d'orientation : retourne $(P_2.x - P_1.x)(P_2.y - P_1.y) - (P_2.y - P_1.y)(P.x - P_1.x)$

**Question :** Comment déterminer si deux segments consécutifs $[P_1P_2]$ et $[P_2P_3]$ forment un vira ge à droite ? Réponse on regarde si Orientation$(P_1, P_2, P_3)$ est négatif.

**Question :** Comment déterminer si deux segments sont sécants ? $[P_1P_2]$ et $[P_3P_4]$ s'intersectent si $P_1$ et $P_2$ ne sont pas du même côté de $[P_3P_4]$ et que $P_3$ et $P_4$ ne sont pas du même côté de $[P_1P_2]$

**Polygone :** Courbe du plan refermée sur elle même, composée d'une suite de segments de droite consécutifs appelés les côtés du polygone. Un sommet est l'intersection de 2 côtés consécutifs, et le nombre de sommets est égal au nombre de côtés.

**Polygone simple :** Seuls les segments consécutifs s'intersectent en leurs extrémités.

**Structure de données :** On caractérise un polygone par la séquence de ses sommets (modulo un choix de sommet de départ arbitraire)

**Propriété :** Si un polygone est simple, alors il découpe le plan en deux zones connexes : l'intérieur et l'extérieure.

**Exercice :** Ecrire un algorithme de complexité en $O(n^2)$ qui décide si un polygone donné sous forme de tableau est simple. ($n =$ nombre de côtés).

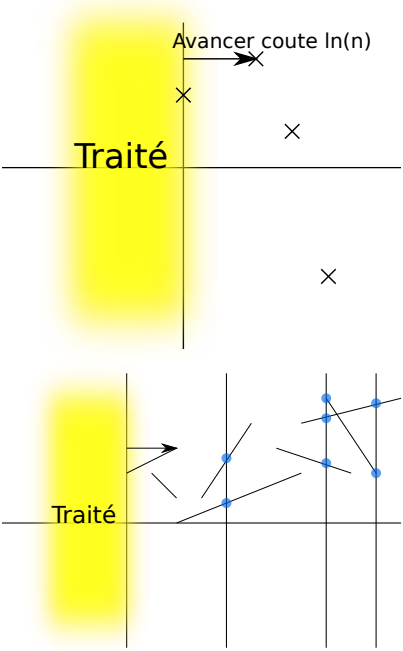
```
Algo estSimple

Debut

  Pour i de 0 à n - 2
    Pour j de i + 2 à n - 1
      Si (j != n - 1 et i != 0) alors
        Si ( Segment-intersection(T[i],T[(i+1) mod n],T[j],T[(j+1) mod
          retourne faux
        FinSi
      FinSi
    FinPour
  FinPour
Fin
```

**Question :** Complexité minimale pour déterminer la liste des points d'intersections d'un polygone ($\Omega(n^2)$) (cf polygone type grille).

**Complexité minimale pour décider si un polygone est**

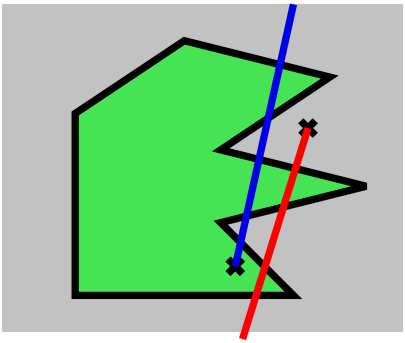
**simple :** $\theta(n \ln n)$ (algo pour tester si il y a une intersection au sein d'un ensemble quelconque de segments adapté au cas des polygones).

**Algorithme par balayage d'une droite :** On se donne une direction de calcul (l'axe des abscisses), et on fait "avancer un droite virtuelle" (perpendiculaire à l'axe des ordonnées)



```
Algo

Trie des points suivant x (nln(n))
Pour i de 0 à n - 1 faire (n fois)
  p <- P[i]
  Si p est extrémité gauche alors
    Insere(T,p) (ln(n))
    Est ce que p instersecte celui au dessus ou en dessous (1)
  Sinon
    Supprime(T,p) (ln(n))
    Est-ce que Au dessus intersecte Au dessous (1)
  FinSi
FinPour
```

**Question :** Etant donné un point x, est-ce que x est à l'intérieur d'un polygone simple ? Astuce depuis x on trace un rayon vers l'infini et on compte les intersections avec le bord (si je pars de l'intérieur c'est impair, sinon c'est pair)



**Convexité :** Une partie $C$ du plan est convexe si et seulement si $\forall x, y \in C$ le segment $[xy]$ est inclus dans $C$. (exemple : algo GJK pour la détection de collisions)

**Utilisation :** A partir d'une forme non convexe, on cherchera son enveloppe convexe.

**Remarque :** Si $P$ est un ensemble de points ou un polygone, alors le contour de $\text{Conv}(P)$ est un polygone simple.

**Propriété :** Dans un polygone simple convexe, deux côtés consécutifs font toujours un "virage à gauche".

**Propriété :** Si $P_iP_{i+1}$ est un côté de l'enveloppe convexe des points $(P_i)_{i\in\{0,\dots,n-1\}}$, alors tous les autres points dont du côté gauche.

**Algo naïf de calcul de l'enveloppe convexe :**