NE PAS IMPRIMER CE DOCUMENT
il va être modifié et complété

Document de travail pour le TP5

Liens vers les questions

1) Algorithmes TD1-TD4.....2
2) Enveloppe convexe (seconde version).....5
3) Si vous avez du temps : exceptions en Java.....6
4) Arbres binaires.....7

1) Algorithmes TD1-TD4

Tri rapide

Vous pouvez voir : http://lwh.free.fr/pages/algo/tri/tri_rapide.htm

L'algorithme de tri rapide choisit une valeur du tableau comme pivot et partage le tableau en trois parties pré-triées : les éléments du tableau inférieurs au pivot, ceux égaux au pivot et enfin ceux supérieurs au pivot.

```
triRapide ( tablo, deb,  fin)
     // tri entre les indices deb et fin compris
     // le pivot est l'élément d'indice deb
     si fin > deb
     alors pivot=tablo[deb]
           drapeau ( tablo, deb, fin, pivot, debEG, finEG )
           triRapide (tablo, deb , debEG-1)
           triRapide (tablo, finEG+1,fin)
     fsi
```

```
drapeau ( tablo, deb, fin, pivot, debEG, finEG )
     dEg ← deb          // dernier des égaux
     fInc ← fin         // dernier des inconnus
     placer ← deb+1     // prochain à placer
     tantque placer <= fInc
     faire si tablo[placer]==pivot
           alors            // on étend vers la droite la zone des égaux
                 dEg++
                 placer++
           sinon si tablo[placer] < pivot
                 alors      // on déplace vers la droite les égaux
                       permuter tablo[dEg] et tablo[placer]
                       dEg++
                       placer++
                  sinon     // on réduit à droite les inconnus
                        permuter tablo[fInc] et tablo[placer]
                        fInc--
                 fsi
           fsi
      ftq
      debEG ← fInc
      finEG ← dEg
      renvoyer debEG et finEG comme résultats
```

Le drapeau renvoie deux valeurs qui sont les indices de début et de fin des égaux au pivot. En Java, vous devrez définir une classe intervalle ayant deux champs (les indices de début et de fin de l'intervalle) ; c'est un objet de cette classe qui sera renvoyé comme résultat de la méthode drapeau.

Tri par fusion

L'algorithme récursif de par fusion (merge sort) partage le tableau en deux moitiés, trie de façon indépendante ces deux moitiés de tableau puis interclasse les deux moitiés triées :

```
triFusion (t, deb, fin)
   // partie de tableau à trier entre deb et fin compris
   si fin > deb
   alors milieu=(fin+deb)/2
         triFusion (t, deb, milieu)
         triFusion (t, milieu+1,fin)
         fusion(t,deb,milieu,fin)
   fsi
```

Pour l'interclassement (ou fusion), il y a trois cas à considérer : il reste des éléments à récupérer dans les deux tableaux, il n'en reste plus que dans l'un ou l'autre des deux tableaux. Deux structures sont possibles pour l'interclassement : une boucle *pour* qui permet de parcourir les indices entre *deb* et *fin* du tableau interclassé ou des boucles *tantque* pour chacun des trois cas ci-dessus :

<u>version avec une boucle *pour*</u>

```
fusion ( T , deb, milieu, fin)
      // deux moitiés de tableau à interclasser
      // entre deb et milieu compris
      // entre milieu+1 et fin compris
      // création d'un tableau temporaire qui sera recopié dans T

    pour i de deb à fin
    faire si i1 <= milieu && i2 <= fin
             // les deux demi-tableaux contiennent encore des éléments
             si T[i1] <= T[i2]
             alors choisir T[i1]
             sinon choisir T[i2]
             fsi
          sinon si i1 <= milieu
                alors choisir T[i1]
                fsi
                si i2 <= fin
                alors choisir T[i2];
                fsi
          fsi
    fpour

    recopier tempo dans T entre les indices deb et fin compris
```

<u>version avec des boucles *tantque*</u>

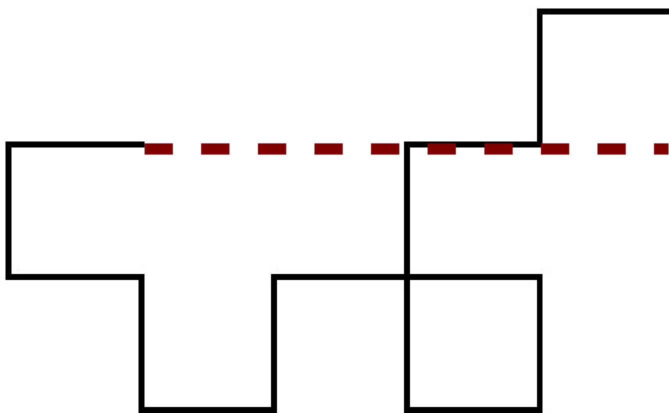
```
fusion ( T , deb, milieu, fin)
      // deux moitiés de tableau à interclasser
      // entre deb et milieu compris
      // entre milieu+1 et fin compris
      // création d'un tableau temporaire qui sera recopié dans T

      tant que  i1 <= milieu ET i2 <= fin
      faire // les deux demi-tableaux contiennent encore des éléments
            si T[i1] <= T[i2]
            alors choisir T[i1]
            sinon choisir T[i2
            fsi
            i++
      ftq
```

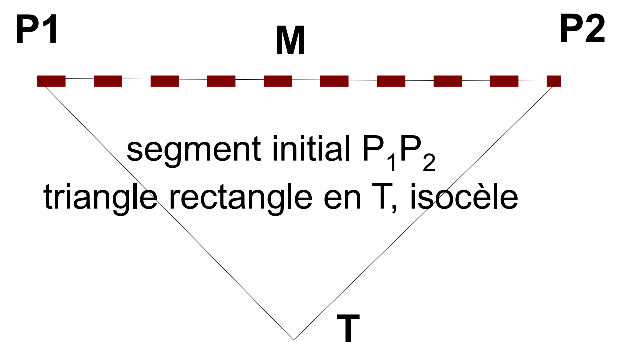
```
      tant que i1 <= milieu ET  i2 > fin
      faire choisir T[i1]
            i++
      ftq

      tant que i2 <= fin ET  i1 > milieu
      faire choisir T[i2]
            i++
      ftq

    recopier tempo dans T entre les indices deb et fin compris
```

Courbe du dragon

Vous pouvez voir :

http://www.mathcurve.com/fractals/dragon/dragon.shtml/






```
dessine( n, x1, y1, x2, y2 )
        si n=0
       alors // pas d'appel récursif
             tracer le segment entre P1(x1,y1) et P2(x2,y2)
       sinon   dx ← x2-x1
               dy ← y2-y1
                      // M(xm, ym) est le milieu du segment P₁P₂
                xm ←  x1 + dx/2
                ym ← y1 + dy/2
                      // T(xt,yt) sommet du triangle rectangle isocèle
                xt ←  xm-dy/2
                yt ← ym+dx/2
                      // appels récursifs sur les côtés de ce triangle
                dessine( n-1, x1, y1, xt, yt)      // sur P1-T
                dessine( n-1, x2, y2, xt, yt)     // sur P2-T
        fsi
```

2) Enveloppe convexe (seconde version)

Vous pouvez voir :
https://interstices.info/jcms/c_16014/un-joli-algorithme-geometrique-et-ses-vilains-problemes-numeriques/

Les données sont structurées de la même façon que dans la première version :

- *max* est le nombre maximum de points du nuage ;
- *nbEC* est le nombre de points de l'enveloppe convexe ;
- les deux tableaux *PTS* et *EC* contiennent à l'indice *i* : en ligne *0,* la coordonnée *x* et en ligne *1,* la coordonnée *y* d'un point qui sera identifié par son indice *i* dans le tableau *PTS* ou *EC*

```
int[][] PTS = new int[2][max];  // pour l'ensemble de points
int[][] ENV = new int[max];  // pour l'enveloppe convexe
```

- le déterminant du produit vectoriel de *PQ* et *QI* permet de déterminer l'orientation relative des points *q* et *i* par rapport au point *p* :

```
orientation ( p,  q, i ) :
    Xpq ← PTS[0][q]-PTS[0][p]
    Ypq ← PTS[1][q]-PTS[1][p]
    Xqi ← PTS[0][i]-PTS[0][q]
    Yqi ← PTS[1][i]-PTS[1][q]
 résultat : Xpq * Yqi – Ypq * Xqi
```

- lorsque les points *q* et *i* sont alignés avec *p*, le produit scalaire de *PQ* et *QI* est positif lorsque *i* est plus éloigné de *p* que *qCapture-*

```
scalaire ( p,  q, i ) :
    Xpq ← PTS[0][q]-PTS[0][p]
    Ypq ← PTS[1][q]-PTS[1][p]
    Xqi ← PTS[0][i]-PTS[0][q]
    Yqi ← PTS[1][i]-PTS[1][q]
 résultat : Xpq * Xqi + Ypq * Yqi
```

- le calcul de l'enveloppe convexe se fait alors de la façon suivante :

```
gift ( PTS, max, EC, nbEC ) :
   p0 ← minimal (PTS,max)
   copier p0 en première position dans EC
   p ← p0
   répéter
      q ← p+1 modulo max
      pour i de 0 à max
      faire t ← orientation (p, q, i)
            si t < 0 ou ( t=0 et scalaire(p, q, i) > 0 )
            alors q ← i
            fsi
      fpour
   enregistrer q dans EC
   p ← q
   jusqu'à p=p0
```

3) Si vous avez du temps : exceptions en Java

Si vous avez un peu de temps vous pouvez tester l'utilisation des exceptions pour contrôler le problème de génération des points uniques signalé en TD

Rappel : l'algorithme ne fonctionne que si tous les points sont uniques (ils sont indentifiés par leur indice dans le tableau). Vous devez donc avoir écrit une méthode de génération aléatoire de points du style :

```
pour i de 1 à max
faire répéter choisir x et y au hasard
      jusqu'à (x,y) n'existe pas dans le tableau PTS
      enregistrer le point (x,y) dans PTS
fpour
```

Vous pouvez assez facilement avoir ainsi une boucle infinie (en tirant aléatoirement des coordonnées entières au hasard entre 0 et 10, vous êtes limités à 100 points distincts). Il est donc prudent de limiter le nombre d'itérations … mais que faire lorsque le nombre maximum d'itérations est atteint ?
Vous pouvez traiter ce problème en utilisant le mécanisme des exceptions de Java :

– définissez une classe exception qui aura pour seul champ la valeur de *i* sur laquelle vous avez atteint le nombre maximum d'itérations autorisées : *i-1* indique le nombre maximum de points disponibles ;

```
public class unicitePoints extends Exception
{  private int ind;
   public unicitePoints (int ind)
   { this.ind = ind; }

   public int getIndice()
   { return ind; }
}
```

– vous devez indiquer que votre méthode de génération des points distincts est susceptible de générer une telle exception ;

```
  public  int[][] genereENSdis ( int max ) throws unicitePoints
```

– dans la boucle de génération des points uniques, activez cette exception lorsque le maximum d'itérations est atteint ;

```
public int[][] genereENSdis ( int max ) throws unicitePoints
  { ...
    for ( int i=1; i< max; i++ )
        { int nbEssais=0;
          do { // choix aléatoire entre 0 et maxAleat
               ...
               nbEssais++;
               if ( nbEssais > maxEssais)          // dépassement
                  { throw new unicitePoints (i);   // DONC exception
                  }
                 ...
```

– lorsque vous appelez la méthode *genereENSdis*, indiquez ce qui doit être fait en cas d'exception (en l'occurrence, modifier le nombre de points, *max :*

```
    try { PTS = genereENSdis ( max );
        }
      catch ( unicitePoints e)
            { max= e.getIndice();
              System.out.println("nombre de points REDUIT à " + max);
            }
```

4) Arbres binaires

Pour ce TP, les arbres binaires seront implémentés comme indiqué en TD :

```
public class arbreB
{
      private arbreB fg, fd;
      private int rac;

        // constante arbre vide
  public static final arbreB Avide= null;

        // pour la génération aléatoire de valeurs entre 0 et 9
  public static final int maxAleat=10;
```

Commencez par la mise en place des **méthodes d'accès en lecture aux champs**. Vous devez ajouter à la classe *arbreB*, les méthodes :

```
estVide   : arbreB → boolean
racine    : arbreB → int
filsG     : arbreB → arbreB
filsD     : arbreB → arbreB
```

Ajoutez à votre classe une **méthode permettant d'initialiser un arbre binaire complet** de façon aléatoire. Vous pouvez par exemple tirer au hasard la profondeur de l'arbre et écrire une méthode récursive de création d'un arbre de profondeur donnée :

```
aleatoireC (hauteur) :
    // res est l'arbre binaire qui sera renvoyé comme résultat
    // en Java : n'oubliez pas les "new" sur les arbreB

    si hauteur==0
    alors res ← arbre vide
    sinon res.rac ← valeur au hasard entre 0 et maxAleat
          res.fg ← aleatoireC( hauteur-1)
          res.fd ← aleatoireC( hauteur-1)
    fsi
    renvoyer res
```

Pour **initialiser un arbre binaire éventuellement incomplet** de façon aléatoire, ajoutez un tirage aléatoire pour décider de l'existence d'un fils gauche et fils droit non vide :

```
aleatoireI (hauteur) :
    // res est l'arbre binaire qui sera renvoyé comme résultat
    // en Java : n'oubliez pas les "new" sur les arbreB

    si hauteur==0
    alors res ← arbre vide
    sinon res.rac ← valeur au hasard entre 0 et maxAleat
          existe ← valeur booléeene au hasard
          si existe==vrai
          alors res.fg ← aleatoireI( hauteur-1)
          fsi
          existe ← valeur booléenne au hasard
          si existe==vrai
          alors res.fd ← aleatoireI( hauteur-1)
          fsi
    fsi
    renvoyer res
```

Pour **initialiser un arbre binaire à partir d'un tableau** comme ceux décrits au TP3 :

```
initTableau( T, rac, max)
    // T est le tableau, indicé entre 1 et max
    // r est l'indice de la racine du sous-arbre à prendre dans le tableau
    // res est l'arbre binaire qui sera renvoyé comme résultat
    // en Java : n'oubliez pas les "new" sur les arbres

    si r <= max
    alors res.rac ← T[r]
          res.fg ← arbre vide
          res.fd ← arbre vide
          si r <= max/2
          alors // il existe des fils dans le tableau
                res.fg ← tableau( T, 2*rac, max)
                res.fd ← tableau ( T, 2*rac+1, max)
          fsi
    fsi
    renvoyer res
```

Pour tester l'initialisation par un tableau d'entiers, vous pouvez utiliser les tableaux Tmin, Tmax et Tdico ainsi que les tableaux T1 à T5 du TP3.

Ajoutez les **méthodes de parcours** (avec affichage de la valeur entière stockée sur chaque nœud) : infixe, préfixe, postfixe et en largeur.

Pour l'affichage en largeur n'oubliez pas d'écrire deux méthodes, la méthode récursive étant une méthode de travail qui ne traite pas la racine de l'arbre initial.

N'oubliez pas d'ajoutez à votre classe :

- les **méthodes d'évaluation des propriétés** de l'arbre : hauteur et équilibrage
- les **méthodes de recherche** simples (arbres quelconques), dans des tas minimum ou maximum, dans des arbres dictionnaires.