Algorithmique & Programmation
Semestre 2 ST
La récursivité

Rappels du semestre 1

Types agrégés | Structuration en sous-algorithmes | Tableaux de variables | Algorithmes de recherche et de tri | Précision, rapidité et complexité

Matrices de variables | Récursivité | Travaux dirigés | Travaux pratiques | Informations

Sujets de projet | Evaluation intermédiaire | Evaluation finale | Evaluation complémentaire | Archives

Cours | TD | TP

Problématique | Définition

Comment bien faire? | Exemples

Version PDF

Clavier.class - Ecran.class - Documentation

Problématique

- Dans certains cas, présentation des problèmes à résoudre sous forme récurrente
  - Formulation utilisée car c'est la plus naturelle ou tout simplement la seule existante
- Forme récurrente: Situation connue à l'étape initiale, calcul de l'état à l'étape i à partir de l'état à l'étape i-1
- Formalisation au moyen de deux ou trois items
  - Les conditions initiales (i.e. l'état du problème à étape 0)
  - Le processus permettant de calculer l'état à l'étape i à partir de l'état à l'étape i-1

- Eventuellement une condition définissant comment le processus récurrent doit se terminer

- Exemple
  Les suites mathématiques: Définition d'une suite f par la donnée de f_0, et par la formule de calcul de f_i à partir de f_{i-1}

- Conditions initiales pas forcément définies par la seule donnée de l'étape 0
  - Etat initial plus n états suivants
  - Pour le calcul de l'état i, utilisation de l'état à l'étape i-1 mais aussi les n états qui l'ont précédée

- Inconvénients d'une telle formulation pour une implantation informatique classique:
  - Nécessité d'une itération avec utilisation d'une boucle "pour" ou d'une boucle "tant que"
  - Nécessité d'une reformulation de la présentation récurrente
    - Risque que cette opération ne soit ni simple ni même possible
    - Risque de commettre une erreur au cours de cette reformulation

- **Exemple**: La suite de Fibonacci $Fib_0 = 0$, $Fib_1 = 1$, $Fib_i = Fib_{i-1} + Fib_{i-2}$
  - Valeurs:
    - $F_0 = 0$
    - $F_1 = 1$
    - $F_2 = 1$
    - $F_3 = 2$
    - $F_4 = 3$
    - $F_5 = 5$
    - $F_6 = 8$
    - $F_7 = 13$
    - $F_8 = 21$
    - $F_9 = 34$
    - $F_{10} = 55$
    - ...

- Implantation séquentielle classique du calcul d'une valeur de la suite de Fibonacci:
  - Une boucle "pour"
  - Deux variables auxiliaires pour stocker les deux dernières valeurs Fib_{i-1} et Fib_{i-2} nécessaires au calcul de Fib_i
  - Une fois Fib_i connu, préparation des valeurs utilisées à l'itération suivante:

- Variable Fib_{i-2} réaffectée avec Fib_{i-1}
- Variable Fib_{i-1} réaffectée avec Fib_i

```
{ Calcul de Fib(n) par methode sequentielle     }
{   - Fib(0) = 0                                }
{   - Fib(1) = 1                                }
{   - Fib(n) = Fib(n-1)+Fib(n-2)                }

entier fonction fib(n)
   Données
    n : entier
   Locales
    res : entier
    a : entier
    b : entier
    i : entier
   dans le cas de n
    0 :
      res <- 0
    1 :
      res <- 1
     autres cas :
      res <- 0
      a <- 0
      b <- 1
       pour i de 2 à n faire
        res <- a+b
        a <- b
        b <- res
       fait
   fcas
   retourner res
fin action
```

```
/* Calcul de Fib(n) par methode sequentielle   */
/*   - Fib(0) = 0                              */
/*   - Fib(1) = 1                              */
/*   - Fib(n) = Fib(n-1)+Fib(n-2)              */

static long fib(int n) {
   long res;
   long a;
   long b;
   int i;
   switch (n) {
     case 0 : {
      res = 0; }
       break;
     case 1 : {
```

```
        res = 1; }
         break;
       default : {
         res = 0;
         a = 0;
         b = 1;
          for ( i = 2 ; i <= n ; i = i+1 ) {
           res = a+b;
           a = b;
           b = res; } }
          break; }
    return res;
}
```

FibonacciSequentiel.lda
FibonacciSequentiel.java
Exemple d'exécution

- **Solution plus directe**: Utiliser la récursivité

La récursivité: Définition

- Outil de développement proposé par la grande majorité des langages informatiques (Java, C, C++, ADA, Basic, Pascal, Lisp, ...)
- Ecriture de sous-algorithmes dont le corps contient un(plusieurs) appel(s) au sous-algorithme lui-même

- Développement d'un sous-algorithme qui s'"utilise" lui-même de la même manière qu'une définition récurrente "s'utilise" elle-même dans la mesure où son terme récurrent définit l'état à étape i par référence à un(des) état(s) précédent(s)

- Pas de syntaxe particulière pour indiquer l'utilisation de la récursivité en langage algorithmique ou en langage Java
- Appel récursif en utilisant le nom de fonction défini en entête et en passant en entête les paramètres nécessaires

- **Exemple**: La suite de Fibonacci

```
/* Calcul de Fib(n) par methode recursive        */
/*  - Fib(0) = 0                                 */
/*  - Fib(1) = 1                                 */
/*  - Fib(n) = Fib(n-1)+Fib(n-2)                 */

entier fonction fib(n)
   Données
    n : entier
```

```
   Locales
    res : entier
   dans le cas de n
    0 :
      res <- 0
    1 :
      res <- 1
     autres cas :
      res <- fib(n-1)+fib(n-2)
   fcas
   retourner res
fin action
```

```
/* Calcul de Fib(n) par methode recursive       */
/*  - Fib(0) = 0                                */
/*  - Fib(1) = 1                                */
/*  - Fib(n) = Fib(n-1)+Fib(n-2)                */

static long fib(int n) {
   long res;
   switch (n) {
     case 0 : {
      res = 0; }
       break;
     case 1 : {
      res = 1; }
       break;
     default : {
      res = fib(n-1)+fib(n-2); }
       break; }
   return res;
}
```

FibonacciRecursif.lda
FibonacciRecursif.java
Exemple d'exécution

- Comparaison entre les versions séquentielle et récursive:
  - Disparition de la boucle "pour" et simplification apparente de l'écriture
  - Boucle "pour" implicitement remplacée par une itération réalisée par la récursivité
    - Appel à Fib(n) -> appel à Fib(n-1) -> appel à Fib(n-2) et ainsi de suite jusqu'à Fib(1)
      -> Recréation implicite mais effectif du processus itératif conséquence de la boucle "pour"

Comment bien programmer la récursivité?

- Programmation d'une fonction récursive -> Résolution de 2 problèmes:
  - Définition du terme récurrent (i.e. où et sous quelle forme l'appel (ou les appels) récursif est réalisé dans le corps de la fonction)
  - Définition de la condition sous laquelle l'appel (ou les appels) récursif n'est pas réalisé
    - Assurance qu'il y aura bien systématiquement fin des appels récursifs successifs en profondeur (implantation des conditions initiales)

- Si non interruption de la récursivité en profondeur:
  - Plantage avec une erreur de type "Stack overflow" (dépassement de capacité de la pile)
  - Erreur liée au fait que tout appel de fonction (récursif ou non) consomme une certaine quantité de mémoire **pendant** l'exécution de la fonction
  - Mémoire allouée au sein d'une zone mémoire spécifique de taille limitée: la "pile"
  - Consommation d'un peu plus de mémoire à chaque appel récursif successif en profondeur jusqu'à atteindre la taille maximale
    -> Génération d'un "plantage" si la récursivité n'est pas stoppée

- Remarque: Même dans le cas d'un fonctionnement normal, la taille finie de la pile limite la profondeur récursive maximale.
  -> Existence d'une limite.

- **Exemple**: La suite de Fibonacci
  - Terme récurrent: $Fib_n = Fib_{n-1} + Fib_{n-2}$
  - Conditions d'arrêt: $Fib_0 = 0$, $Fib_1 = 1$
  - n = 1 ou n = 0
    -> Condition d'arrêt vérifiée
    -> Pas d'amorçage de la récursivité
  - n plus grand ou égal à 2
    -> Condition d'arrêt non vérifiée
    -> Amorçage de la récursivité
    -> Décréments de 1 lors du premier appel puis de 2 lors du second

Cas typiques d'utilisation de la récursivité

Utilisation fréquente de la récursivité dans les cas suivants:

- Algorithmes implantant une méthode dichotomique
  - Recherches dichotomiques
  - Tris dichotomiques
  - ...

- Algorithmes d'exploration
  - Parcours
  - Coloriage
  - ...

Exemples d'utilisation de la récursivité

- **Multiplier deux nombres entiers strictement positifs sans utiliser l'opérateur multiplication (difficulté faible)**

- a*b = b si a = 1
- a*b = b+(a-1)*b si a <> 1

- **Inverser une chaîne de caractères (difficulté faible)**

- Inversion d'une chaîne de caractères st composée de plus de 1 caractère:
  - Extraction de la chaîne s formée du premier caractère de st
  - Concaténation de s à la fin de la chaîne de caractères obtenue par inversion de st moins son premier caractère
    inverse("abc") = inverse("bc")+"a"
- Inversion d'une chaîne de caractères st composée de 0 ou 1 caractère:
  - Déjà inversée

```
{ Inversion d'une chaine de caracteres          }

chaine fonction inversionChaine(st)
   Données
    st : chaine
   Locales
    s : chaine
    s1 : chaine
    s2 : chaine
    lst : entier
  lst <- longueur(st)
   si ( lst = 0 ) ou ( lst = 1 ) alors
    s <- st
     sinon
    s1 <- substring(st,0,1)
    s2 <- inversionChaine(substring(st,1,lst))
    s <- concatener(s2,s1)
   fsi
   retourner s
fin action
```

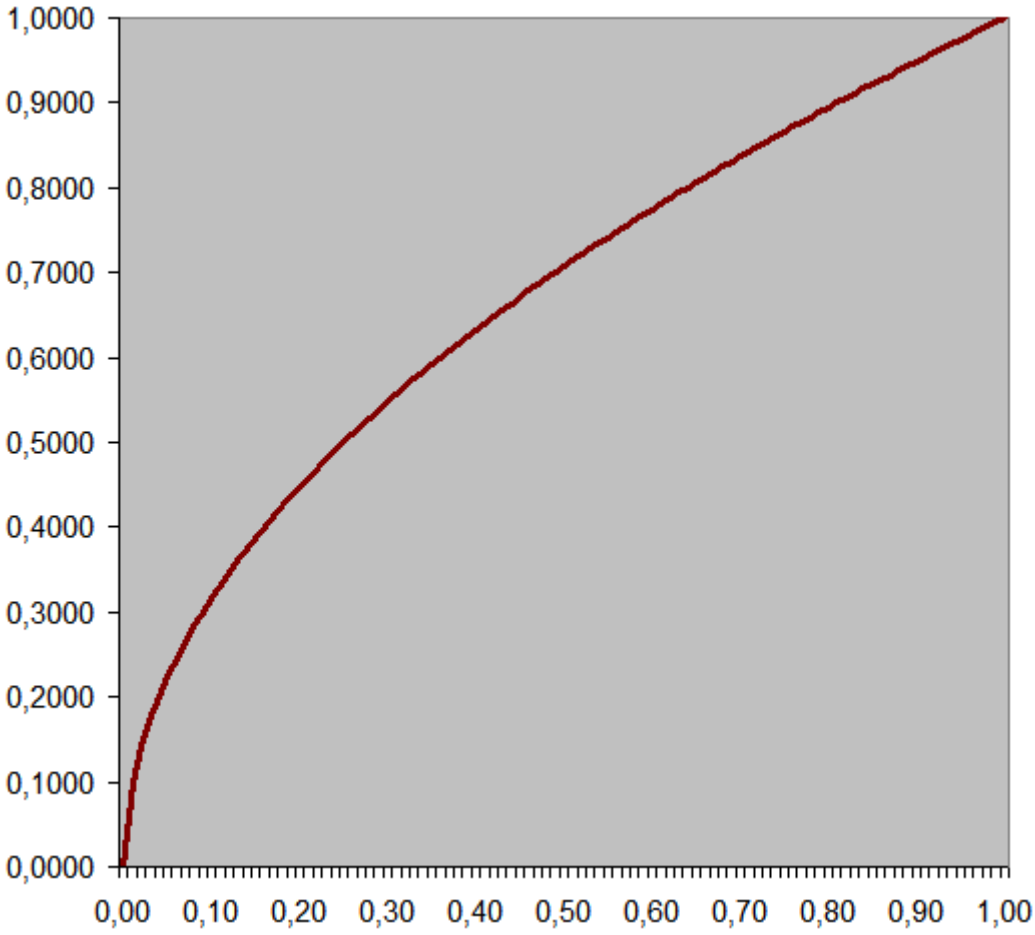
```
/* Inversion d'une chaine de caracteres          */
```

```
static String inversionChaine(String st) {
   String s;
   String s1;
   String s2;
   int lst = st.length();
   if ( ( lst == 0 ) || ( lst == 1 ) ) {
    s = st; }
     else {
    s1 = st.substring(0,1);
    s2 = inversionChaine(st.substring(1,lst));
    s = s2+s1; }
   return s;
}
```

InversionChaineCaracteres.lda
InversionChaineCaracteres.java
Exemple d'exécution

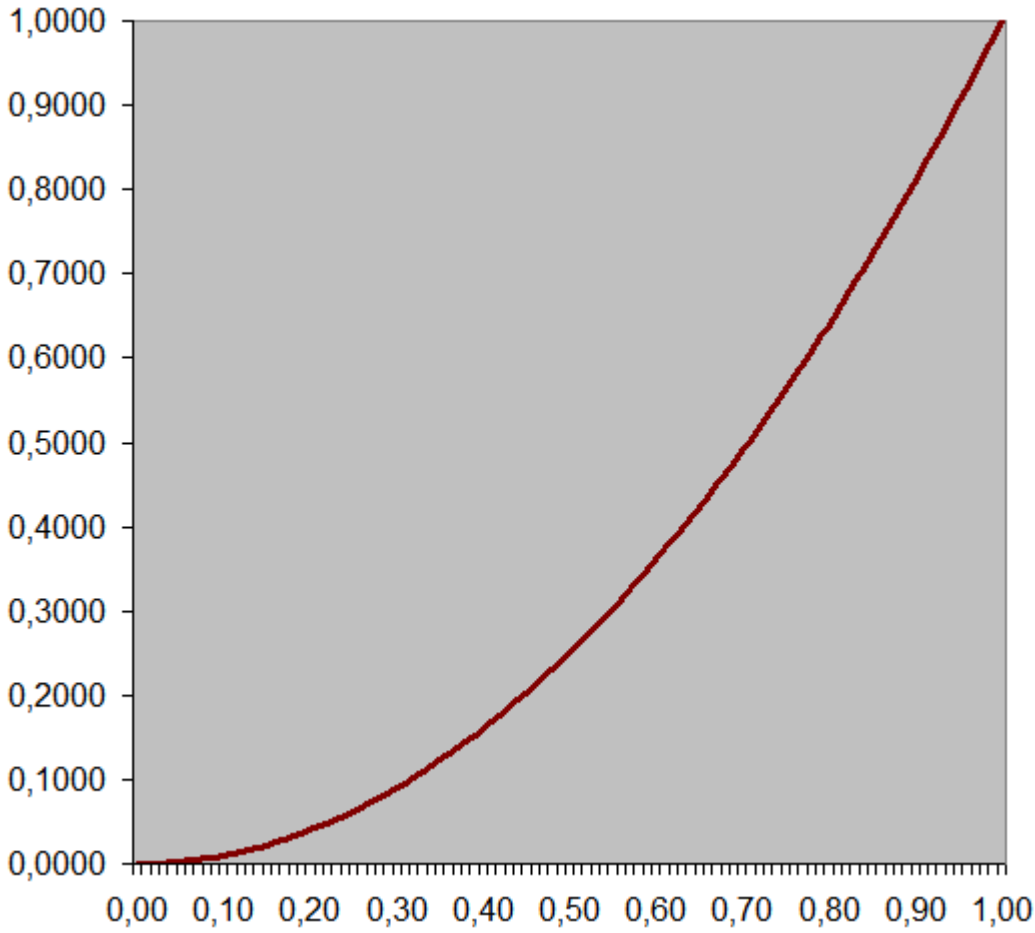
- **Trouver, sans utiliser sqrt, la racine carrée positive d'un nombre réel compris dans l'intervalle [0.0,1.0] (difficulté moyenne)**

- Détermination de la valeur y positive telle que y = f(x)
  où f est la fonction racine carrée et x est une valeur réelle positive contenue dans l'intervalle [0.0,1.0]
- Valeur recherchée comprise dans l'intervalle [0.0,1.0]



Fonction "Racine carrée" sur l'intervalle [0.0,1.0]

- Reformulation du problème en $y' = f^{-1}(x')$
  où $y' = x$, $x' = y$ et f^{-1} est la fonction "carré" que l'on peut implanter en multipliant une valeur par elle-même
- Fonction f^{-1} **continue** et **croissante** sur l'intervalle [0.0,1.0]



Fonction "Carré" sur l'intervalle [0.0,1.0]

- Recherche de la valeur x' telle que x'*x'=y'

- Technique dichotomique

- Intervalle de recherche d'ouverture réduite d'un facteur 2.0 à chaque étape de recherche à partir de l'intervalle initial [0.0,1.0]
- A chaque étape, déplacement de l'une des deux bornes pour adopter leur valeur moyenne m
- Choix de la borne déplacée réalisé en comparant la valeur y' au carré de m (m*m)
  - Si m*m plus grand que y', report de la borne supérieure sur m
  - Si m*m plus petit que y', report de la borne inférieure sur m
- "Relance" récursive sur le nouvel intervalle ainsi calculé

- Implantation récursive
  -> Problème de l'arrêt de la récursivité
  - Implantation d'un test sur l'ouverture de l'intervalle
  - Arrêt quand ouverture inférieure à une valeur limite ε
    -> Résultat: La valeur moyenne m
    -> Erreur commise inférieure à ε

```
{ Constante de definition de la precision     }
{ de calcul                                   }
```

```
constante EPSILON : reel <- 0.00000000001

{ Methode recursive de calcul                    }
{ de la racine carree d'un nombre reel           }

reel fonction racine(x,a,b)
   Données
    x : reel
    a : reel
    b : reel
   Locales
    res : reel
    m : reel
  m <- (b+a)/2.0
   si b-a > EPSILON alors
     si m*m > x alors
      res <- racine(x,a,m)
       sinon
      res <- racine(x,m,b)
     fsi
     sinon
    res <- m
   fsi
   retourner res
fin action

{ Calcul d'une racine carree                     }

reel fonction racineCarree(x)
   Données
    x : reel
   Locales
    v : réel
  v <- racine(x,0.0,1.0)
   retourner v
fin action
```

```
/* Constante de definition de la precision      */
/* de calcul                                    */

static final double EPSILON = 0.00000000000001;

/* Methode recursive de calcul                  */
/* de la racine carree d'un nombre reel         */

static double racine(double x,double a,double b) {
   double res;
   double m;
  m = (b+a)/2.0;
```

```
    if ( b-a > EPSILON ) {
      if ( m*m > x ) {
       res = racine(x,a,m); }
        else {
       res = racine(x,m,b); } }
      else {
     res = m; }
    return res;
}

/* Calcul d'une racine carree                    */

static double racineCarree(double x) {
    double v = racine(x,0.0,1.0);
    return v;
}
```

RacineCarree.lda
RacineCarree.java
Exemple d'exécution

- **Parcourir un labyrinthe sans cycle (difficulté moyenne)**

- Labyrinthe:
  Matrice de booléens où les vrais codent pour les couloirs
  Couloirs épais de 1 "booléen"
  Déplacements autorisés à "droite", à "gauche", en "haut" et en "bas" si booléen à vrai dans cette direction
  Entrée par une cellule à vrai située sur le bord
  Sortie par une autre cellule, s'il en existe une, située sur le bord

- Parcours:
  Parcours possible d'une cellule si elle est à vrai
  Continuation du parcours à partir d'une cellule en lançant récursivement l'algorithme de parcours sur ces 4 voisines

```
FFFFFFFFFFFFFFFFFFF
FVVVVVVVVVVVVVVVVFF
FVFFVFFFFFFFFFVFFFF
VVFFVFFFVVVVVFVFFFF
FFFFVFFFVFFFFFVFFFF
FFFFVFFVVVVFVVVVVFF
FFVVVFFFFFVFVFFFFVF
FFVFFFFFFFVFVFFFFVF
FFVFFVFFFFVFFFVFFVF
FFVFFVVVVVVFFFVFFVF
FFVFFFFVFFFFFFVVVVF
```

```
FFVVVVVVVVVVFFVFFFF
FFFFVFFVFFFFVFFVVVFF
FVFFVFFFFFFFVFFFFVFF
FVFFFFVVVVVVFFFFFFF
FVVFFFVFFFFFFFFVVVV
FFVFFFVFVVVVVFFFVFFF
FFVFFFVFVFFFFFFFVFFF
FFVVVVVVVVVVVVVVFFF
FFFFFFFFFFFFFFFFFFF
```

```
/* Type structure de definition d'une position    */
/* par un numero de ligne et un numero de colonne */

static class Position {
   int l = -1;
   int c = -1; };

/* Test d'egalite de deux position                */

static boolean egal(Position p1,Position p2) {
   boolean res;
   if ( ( p1.c != p2.c ) || ( p1.l != p2.l ) ) {
    res = false; }
     else {
    res = true; }
   return res;
}

/* Fonction de test de l'existence d'une sortie   */
/* a un labyrinthe represente par un tableau      */
/* de booleen                                     */
/* Le parametre p est la position a partir        */
/* de laquelle rechercher                         */
/* Le parametre origine est la position qui vient */
/* d'etre quittee pour atteindre la position p et */
/* vers laquelle il est interdit de retourner     */
/* Le parametre sortie est destine a recueillir   */
/* la position de la sortie si celle-ci existe    */
/* Le booleen retourne indique si une sortie      */
/* a ete trouvee                                  */

static boolean trouveSortie(boolean [][] lb,
                            Position p,
                            Position origine,
                            Position sortie) {
   boolean res = false;
   Position np = new Position();
   if ( ( p.l == -1 ) || ( p.l == lb.length ) ||
        ( p.c == -1 ) || ( p.c == lb[0].length ) ) {
    sortie.l = origine.l;
```

```
    sortie.c = origine.c;
    res = true; }
     else {
     if ( lb[p.l][p.c] == true ) {
      np.c = p.c+1;
      np.l = p.l;
       if ( egal(np,origine) == false ) {
        res = trouveSortie(lb,np,p,sortie); }
       if ( res == false ) {
        np.c = p.c-1;
        np.l = p.l;
         if ( egal(np,origine) == false ) {
          res = trouveSortie(lb,np,p,sortie); }
         if ( res == false ) {
          np.c = p.c;
          np.l = p.l+1;
           if ( egal(np,origine) == false ) {
            res = trouveSortie(lb,np,p,sortie); }
           if ( res == false ) {
            np.c = p.c;
            np.l = p.l-1;
             if ( egal(np,origine) == false ) {
              res = trouveSortie
(lb,np,p,sortie); } } } } } }
   return(res);
}
```

Labyrinthe.java
Exemple d'exécution

- **Parcourir des arborescences (difficulté importante)**

- Arborescence: Structure informatique mimant un arbre
  - Composée de noeuds et d'arêtes reliant ces noeuds
  - Un et un seul chemin entre tout couple de noeuds

- Désignation usuelle d'un noeud comme étant la "racine" de l'arborescence
  - Généralement utilisée pour la repérer car accès possible à tous les autres noeuds en "descendant" (ou "montant" suivant le vocabulaire employé) dans l'arborescence à partir de la racine

- Pour un noeud n:
  - Noeud "père": Premier noeud sur le chemin partant de n en direction de la racine
  - Noeuds "fils": Autres noeuds que le noeud père connectés à n

- "Feuille" (noeud terminal): Noeud sans fils

- Seul noeud sans père: Le noeud racine

- Arborescence binaire: Arborescence pour laquelle aucun noeud n'a plus de 2 fils
- Arborescence binaire stricte: Arborescence où chaque noeud possède soit 2, soit 0 fils

Parcours une arborescence

- Utilisation naturelle de la récursivité pour manipuler les arborescences (binaires ou non) dont on ne connaît généralement que la racine
- Réalisation de toutes les opérations nécessitant un parcours complet en lançant la fonction récursive de traitement sur la racine
- Programmation du corps de la fonction pour lancer successivement une exécution récursive sur tous les sous-arbres (sous-arbre "droit" puis sous-arbre "gauche" (ou l'inverse) pour les arbres binaires strictes) du noeud passé en paramètre

Modélisation d'une arborescence en langage Java

Méthode 1: Un tableau de noeuds

- Classe java pour représenter un noeud d'une arborescence où aucun noeud n'a plus de 10 fils:

```
static class Noeud {
  int pere = -1;
  int [] fils = { -1,-1,-1,-1,-1,-1,-1,-1,-1,-1 }; };
```

- Classe java pour représenter un noeud d'une arborescence binaire:

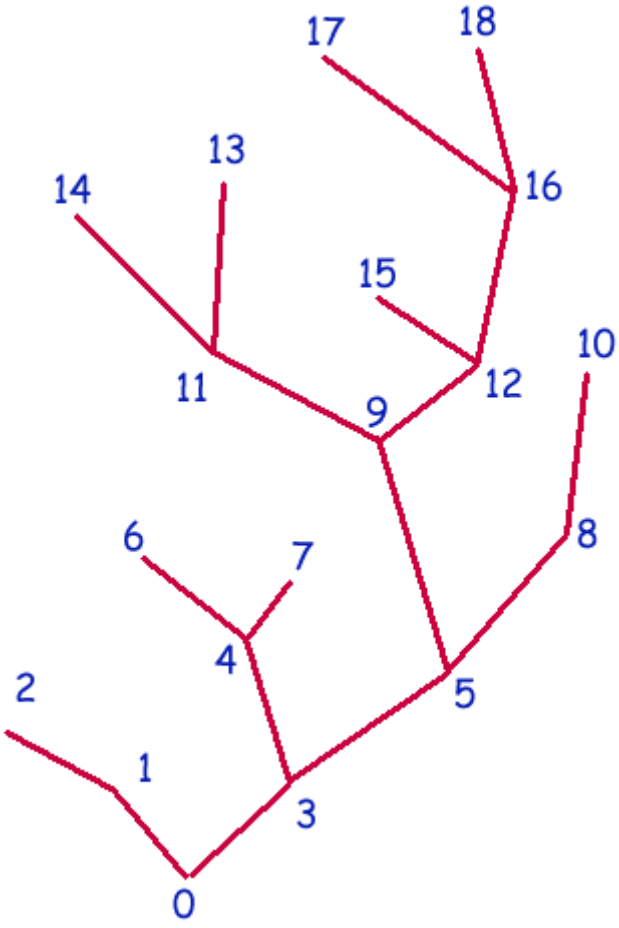
```
static class Noeud {
  int pere = -1;
  int filsDroit = -1;
  int filsGauche = -1; };
```

- Arborescence codée dans un tableau de `Noeud`.
  Pour chaque noeud:
  - Champ pere: indice du noeud père dans le tableau de noeuds de l'arborescence
  - Champs fils: indices des fils dans le tableau de noeuds de l'arborescence

  Noeud racine placé à l'indice 0 du tableau et seul noeud dont l'indice du père est égal à -1.
  Noeuds internes caractérisés par le fait que l'indice d'au moins un fils est différent de -1.
  Noeuds terminaux caractérisés par le fait que l'indice de tous leurs fils est égal à -1.

- Exemple d'arborescence binaire

Indice	Père	FilsDroit	FilsGauche
0	-1	1	3
1	0	2	-1
2	1	-1	-1
3	0	4	5
4	3	6	7
5	3	8	9
6	4	-1	-1
7	4	-1	-1
8	5	-1	10
9	5	11	12
10	8	-1	-1
11	9	13	14
12	9	15	16
13	11	-1	-1
14	11	-1	-1
15	12	-1	-1
16	12	17	18
17	16	-1	-1
18	16	-1	-1



Arborescence binaire modélisée par le tableau de noeuds ci-contre

- Exemples:
  - Recherche du nombre de noeuds d'une arborescence
  - Recherche de la distance maximale existant entre la racine et l'ensemble des feuilles d'une arborescence

```
/* Structure de stockage d'un noeud                */
/* d'arborescence                                  */

static class Noeud {
   int pere = -1;
   int filsDroit = -1;
   int filsGauche = -1; };

/* Creation d'une arborescence                     */

static Noeud [] generation() {
   Noeud [] t = new Noeud[19];
   for ( int i = 0 ; i < 19 ; i++ )
    t[i] = new Noeud();
```

```
  t[0].filsDroit = 1;
  t[0].filsGauche = 3;
  t[1].pere = 0;
  t[1].filsDroit = 2;
  t[2].pere = 1;
  t[3].pere = 0;
  t[3].filsDroit = 4;
  t[3].filsGauche = 5;
  t[4].pere = 3;
  t[4].filsDroit = 6;
  t[4].filsGauche = 7;
  t[5].pere = 3;
  t[5].filsDroit = 8;
  t[5].filsGauche = 9;
  t[6].pere = 4;
  t[7].pere = 4;
  t[8].pere = 5;
  t[8].filsGauche = 10;
  t[9].pere = 5;
  t[9].filsDroit = 11;
  t[9].filsGauche = 12;
  t[10].pere = 5;
  t[11].pere = 9;
  t[11].filsDroit = 13;
  t[11].filsGauche = 14;
  t[12].pere = 9;
  t[12].filsDroit = 15;
  t[12].filsGauche = 16;
  t[13].pere = 11;
  t[14].pere = 11;
  t[15].pere = 12;
  t[16].pere = 12;
  t[16].filsDroit = 17;
  t[16].filsGauche = 18;
  t[17].pere = 16;
  t[18].pere = 16;
   return t;
}

/* Creation d'une arborescence aleatoire       */
/* n: Le nombre de noeuds                      */
/* p: La probabilite qu'un noeud ait 2 fils    */
/* A chaque iteration de croissance, un noeud  */
/* est choisi au hasard parmi les feuilles     */
/* et un ou deux fils lui sont crees           */
/* en fonction de la probabilite p             */

static Noeud [] generationAleatoire(int n,double p) {
   Noeud [] t = new Noeud[n];
   for ( int i = 0 ; i < n ; i++ )
```

```
     t[i] = new Noeud();
   int nbn = 1;
   while ( nbn < n ) {
     int nn = 0;
     do {
      nn = ((int) (Math.random()*10e9))%nbn; }
     while ( ( t[nn].filsDroit != -1 ) || ( t
[nn].filsGauche != -1 ) );
     if ( ( Math.random() < p ) && ( nbn < n-1 ) ) {
      t[nn].filsDroit = nbn;
      t[nbn].pere = nn;
      nbn += 1;
      t[nn].filsGauche = nbn;
      t[nbn].pere = nn;
      nbn += 1; }
       else {
       if ( Math.random() < 0.5 ) {
        t[nn].filsDroit = nbn;
        t[nbn].pere = nn;
        nbn += 1; }
         else {
        t[nn].filsGauche = nbn;
        t[nbn].pere = nn;
        nbn += 1; } } }
   return t;
}

/* Calcul du nombre de noeuds                    */
/* d'une arborescence                            */

static int taille(Noeud [] tn,int n) {
   int nb;
   if ( n == -1 ) {
    nb = 0; }
     else {
    nb = 1 + taille(tn,tn[n].filsDroit)
           + taille(tn,tn[n].filsGauche); }
   return nb;
}

/* Calcul de la profondeur d'une arborescence  */

static int profondeur(Noeud [] tn,int n) {
   int prf;
   int prfd;
   int prfg;
   if ( n == -1 ) {
    prf = 0; }
     else {
    prfd = profondeur(tn,tn[n].filsDroit);
```

```
     prfg = profondeur(tn,tn[n].filsGauche);
      if ( prfd > prfg ) {
       prf = 1 + prfd; }
        else {
       prf = 1 + prfg; } }
    return prf;
}

/* Affichage des noeuds d'une arborescence     */

static void affichage(Noeud [] tn,int n) {
    if ( n != -1 ) {
     Ecran.afficher(String.format("%4d",n));
     Ecran.afficher(String.format("%4d",tn[n].filsDroit));
     Ecran.afficher(String.format("%4d",tn[n].filsGauche));
     Ecran.afficher(String.format("%4d",tn[n].pere));
     Ecran.sautDeLigne();
     affichage(tn,tn[n].filsDroit);
     affichage(tn,tn[n].filsGauche); }
}

/* Affectation de son pere a chaque noeud      */
/* d'une arborescence                          */

static void affectationPeres(Noeud [] tn,int n,int p) {
  tn[n].pere = p;
    if ( tn[n].filsDroit != -1 ) {
     affectationPeres(tn,tn[n].filsDroit,n); }
    if ( tn[n].filsGauche != -1 ) {
     affectationPeres(tn,tn[n].filsGauche,n); }
}

/* Affectation de son pere a chaque noeud      */
/* d'une arborescence                          */

static void affectationPeres2(Noeud [] tn,int n,int p) {
    if ( n != -1 ) {
     tn[n].pere = p;
     affectationPeres2(tn,tn[n].filsDroit,n);
     affectationPeres2(tn,tn[n].filsGauche,n); }
}
```

Arborescence.java
Exemple d'exécution

- **Méthode 2 (java niveau "expert")**

- Classe java pour une arborescence quelconque:

```
static class Noeud {
```

```
  Noeud pere = null;
  Noeud [] fils; };
```

- Classe java pour une arborescence binaire:

```
static class Noeud {
  Noeud pere = null;
  Noeud [] fils; };
```

- Classe java pour une arborescence binaire stricte:

```
static class Noeud {
  Noeud pere = null;
  Noeud filsDroit = null;
  Noeud filsGauche = null; };
```

- Exemples:
  - Recherche du nombre de noeuds d'une arborescence
  - Recherche de la distance maximale existant entre la racine et l'ensemble des feuilles d'une arborescence

```
/* Structure de stockage d'un noeud              */
/* d'arborescence                                */

static class Noeud {
   Noeud pere = null;
   Noeud filsDroit = null;
   Noeud filsGauche = null; };

/* Creation d'une arborescence                   */

static Noeud generationAleatoire(double prb,double inc) {
   Noeud n;
   if ( Math.random() < prb ) {
    n = null; }
     else {
    n = new Noeud();
    n.filsDroit = generationAleatoire(prb+inc,inc);
     if ( n.filsDroit != null )
      n.filsDroit.pere = n;
    n.filsGauche = generationAleatoire(prb+inc,inc);
     if ( n.filsGauche != null )
      n.filsGauche.pere = n; }
   return n;
}

/* Calcul du nombre de noeuds                    */
/* d'une arborescence                            */

static int taille(Noeud n) {
   int nb;
```

```
    if ( n == null ) {
     nb = 0; }
      else {
     nb = 1 + taille(n.filsDroit)
            + taille(n.filsGauche); }
    return nb;
}

/* Calcul de la profondeur d'une arborescence  */

static int profondeur(Noeud n) {
    int prf;
    int prfd;
    int prfg;
    if ( n == null ) {
     prf = 0; }
      else {
     prfd = profondeur(n.filsDroit);
     prfg = profondeur(n.filsGauche);
      if ( prfd > prfg ) {
       prf = 1 + prfd; }
        else {
       prf = 1 + prfg; } }
    return prf;
}
```

ArborescenceBinaire.java

Exemple d'exécution

Auteur: Nicolas JANEY
UFR Sciences et Techniques
Université de Besançon
16 Route de Gray, 25030 Besançon
nicolas.janey@univ-fcomte.fr