La récursivité

Maria Virginia Aponte

CNAM-Paris

24 avril 2017

Définitions récursives

Une définition est *récursive* si est formulée en termes d'elle même.

- A est un ancêtre de B =
  si A est un parent de B, ou si A est un ancêtre d'un parent de B ;
- un *répertoire* est une liste de fichiers ou de *répertoires* ;
- une *instruction Java* peut être une instruction composite (while, for, etc) composée d'autres *instructions*.
- il existe un chemin entre (X et Y)=
  si X=Y ou s'il y a une route directe entre X et Y, ou s'il y a une route entre X et Z et un chemin entre (Z et Y).
- X appartient au tableau T[0..N]=
  si X=T[0] ou si X appartient au tableau T[1..N]

Méthodes récursives

Une méthode *m* est récursive :

- si dans son corps, elle contient un appel vers elle-même *m* (*récursivité directe*), ou
- si elle contient un appel vers une deuxième méthode *p* contenant un appel vers *m* (*récursivité indirecte*).

Méthodes récursives

- Le récursivité est une technique de programmation.
- Si *m* s'appelle elle-même, elle recommence ses calculs, donc elle boucle !

La récursivité permet de *boucler* dans un programme.

La factorielle d'un nombre entier

La factorielle d'un nombre positif n est défini par :

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ 1 \times 2 \times \ldots (n-2) \times (n-1) \times n & \text{si} \quad n > 0 \end{cases}$$

On constate que

$$1 \times 2 \times \ldots (n-2) \times (n-1) = (n-1)!$$

correspond au calcul de $(n-1)!$. On peut donc reformuler $n!$ sous forme récursive :

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ (n-1)! \times n & \text{si} \quad n > 0 \end{cases}$$

factorielle : formulation récursive

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ (n-1)! \times n & \text{si} \quad n > 0 \end{cases}$$

Pour calculer la factorielle $n!$ d'un nombre n

- si $n = 0$ le résultat est 1,
- si $n > 0$ il suffit de calculer $(n-1)!$ puis de multiplier ce résultat par n.

factorielle récursive en Java (version 1)

```
int factorielle (int n){
  int resultat;
  if(n==0) resultat = 1;
  else {
    int sous_resultat = factorielle (n-1);
    resultat = sous_resultat * n;
  }
  return resultat;
}
```

Pour calculer la factorielle $n!$ d'un nombre n

- si $n = 0$ le résultat est 1,
- si $n > 0$ il suffit de calculer $(n-1)!$ puis de multiplier ce résultat par n.

factorielle : version 2

Nous ne sommes pas obligés d'utiliser des variables locales pour ce calcul :

```
int factorielle (int n){
 if(n==0) return 1;
  else {
    return (n * factorielle (n-1));
  }
}
```

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ (n-1)! \times n & \text{si} \quad n > 0 \end{cases}$$

factorielle : version 3

1 ! = 0 ! = 1. On peut les factoriser en un seul cas de base

```
int factorielle (int n){
  int resultat;
  if(n<=1) resultat = 1;
  else {
    int sous_resultat = factorielle (n-1);
    resultat = sous_resultat * n;
  }return resultat;
}
```

- En quoi cette nouvelle écriture est intéressante ?
- Est-elle fidèle à la définition mathématique donnée précedamment ?

factorielle : version un peu plus efficace

```
static int factorielle (int n){
  int resultat;
  if(n<=1) resultat = 1;
  else {
    int sous_resultat = factorielle (n-1);
    resultat = sous_resultat * n;
  }  return resultat;
}
```

- En quoi cette nouvelle écriture est intéressante ?
  - elle permet d'éviter le dernier appel récursif : `factorielle(0)` ;
  - elle permet d'éviter que le programme ne boucle de façon infinie (pour n négatif) ;
- Est-elle fidèle à la définition mathématique donnée précedamment ?
  Non, car elle fournit un résultat pour n négatif

Déroulement d'un appel récursif

Un appel récursif se déroule en général :

1. la réalisation de certains calculs sur *l'argument de l'appel*,
2. puis, possiblement *par un nouvel* appel récursif sur un *nouvel argument*,
   - ce nouvel appel permet de recommencer les calculs, mais appliqués sur le nouvel argument
3. éventuellement, l'appel récursif se *termine* par la réalisation d'un *calcul final, qui ne comporte pas d'appel récursif*.

Déroulement d'un appel récursif (version 3)

Appel `factorielle(3)` (pour la version 3) ⇒

1. comme $n = 3$ on doit exécuter :

```
sous_resultat = factorielle (3-1); //1er recursif
resultat = sous_resultat * n; // en attente!
return resultat;
```

⇒ exécute 1er appel récursif et laisse 2 instructions *en attente* du retour de cet appel.

2. 1er appel récursif `factorielle(2)` ⇒

```
sous_resultat = factorielle (2-1); //2eme recursif
resultat = ...; // en attente!
```

⇒ 2ème appel récursif + 2 instructions en attente.

3. 2ème appel récursif `factorielle(1)` ⇒ appel *terminal*.

Déroulement de `factorielle(3)`

1.
```
sous_resultat = factorielle (3-1); //1er recursif
resultat = sous_resultat * n; // en attente!
return resultat;
```

2. 1er appel récursif `factorielle(2)` ⇒
```
sous_resultat = factorielle (2-1); // 2eme
resultat = sous_resultat * n; // en attente!
return resultat;
return resultat;
```

3. 2ème appel récursif `factorielle(1)` ⇒ appel *terminal* :
```
if(n==1) resultat = 1;
return resultat;
```

Ce dernier appel retourne 1

Déroulement de `factorielle(3)`

- Chaque appel récursif retourne vers son appelant avec le résultat obtenu ;
- ce retour se fait pour continuer avec les instructions restées en attente !
- le retour se fait dans l'ordre inverse des appels : le dernier appelé est le premier à retourner.

Déroulement de `factorielle(3)`

3. Retour de `factorielle(1)` $\Rightarrow$ $\boxed{1}$
2. on retourne $\boxed{1}$ vers le 1er appel récursif (`factorielle(2)`), où $n = 2$:

```
sous_resultat = 1; //on reprend ici
resultat = sous_resultat * n;
return resultat;
```

Donc, `factorielle(2)` $\Rightarrow$ $\boxed{2}$

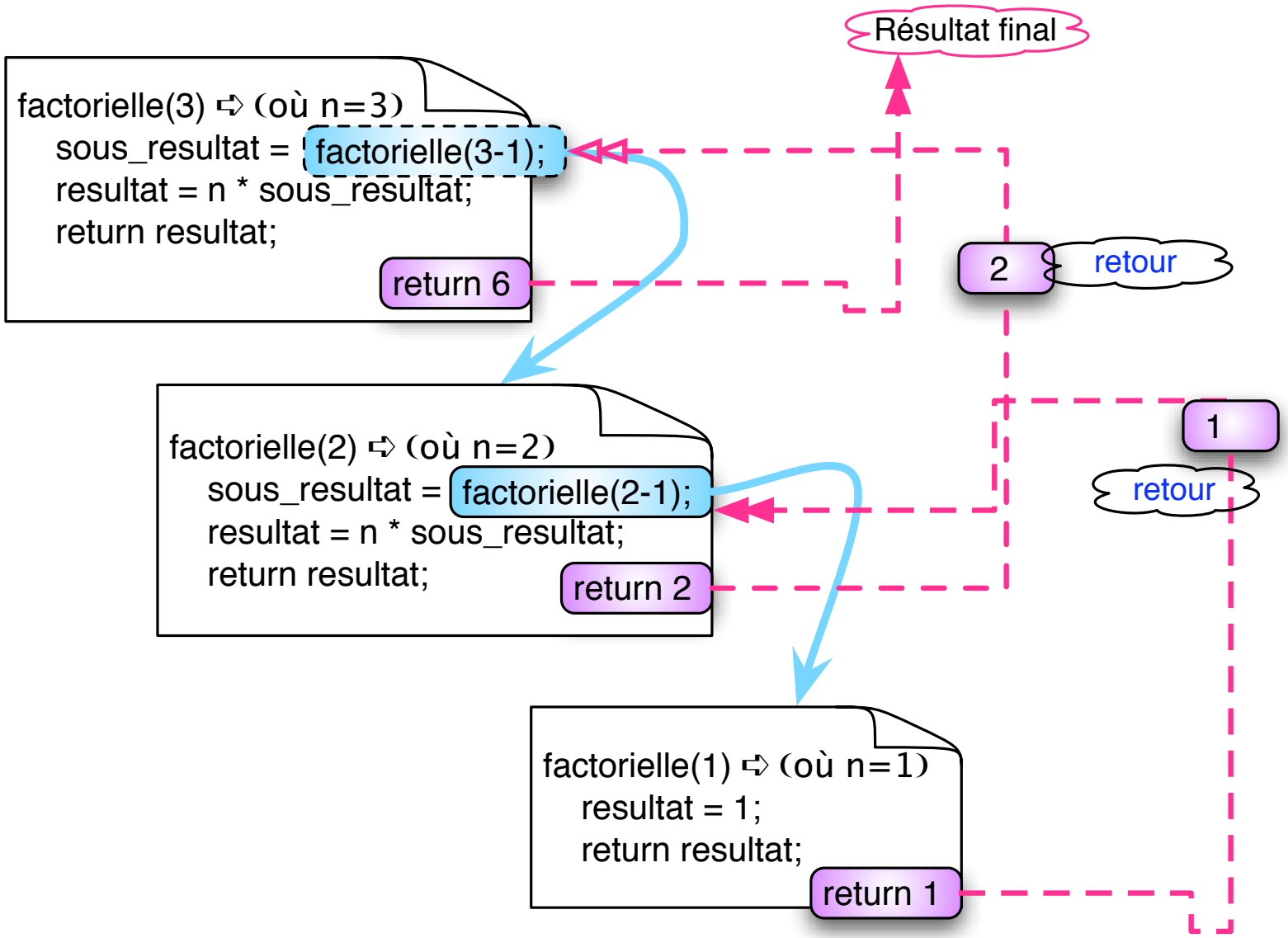
1. on retourne $\boxed{2}$ vers le 1'appel initial (`factorielle(3)`), où $n = 3$:

```
sous_resultat = 2; //reprend ici
resultat = sous_resultat * n; //
return resultat;
```

Donc, `factorielle(3)` $\Rightarrow$ $\boxed{6}$

Déroulement de `factorielle(3)`

(Pour la version 3)



Modèle de la mémoire

Chaque *appel* de sous-programme Java utilise une *zone mémoire locale* afin de stocker :

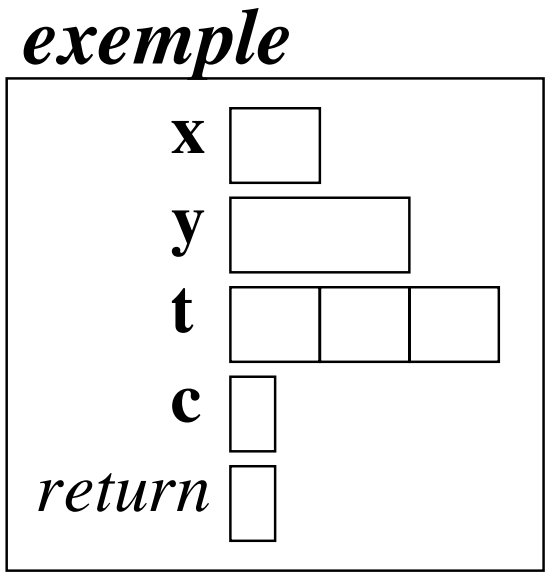
- les *paramètres + variables locales*,
- si la méthode est récursive : stocke en plus son *résultat retourné* ;

Cette mémoire est *nouvellement allouée pour chaque appel*.

C'est le contexte d'exécution de la méthode.

Contexte d'exécution pour la méthode `exemple`

```
static boolean exemple (int x, double y){
   int [] t = new int[3];
   char c;
   ...
}
```



Contexte d'exécution d'une méthode

Mémoire locale qui sert à stocker paramètres et variables locales *pendant l'exécution d'une méthode* :

- allouée *au début de chaque appel* à la méthode ;
- dans une zone mémoire appelée *pile d'exécution* ;
- supprimée de cette zone mémoire en fin d'exécution ;

Déroulement d'un appel de méthode

Lors d'un appel à la méthode $m(v_1, v_2, \ldots)$:

1. on alloue son contexte d'exécution dans la pile d'exécution, en y recopiant la valeur des paramètres $v_1, v_2, \ldots$;
2. on parle d'*empiler* ce contexte dans la pile ;
3. à la fin de l'exécution, on *dépile* ce contexte (enlevé à la fin de la méthode).
4. Seule sa valeur de retour y reste (pour être récupérée par le contexte précédent qui est son appelant) ;
5. si plusieurs méthodes sont appelées en cascade, leurs contextes sont empilés successivement dans la pile, et dépilés en commençant par le dernier appelé (empilé).

Exemple d'appel

Un appel à `exemple` depuis le `main` :

```
public class Principal{
  static boolean exemple (int x, double y){ ... }
  public static void main(String[] args){
    double x = 3.5;
    int n = 2;
    boolean c = true;
    .........
    c = exemple(n, x);
    .........
  }
}
```

Exécution d'un appel

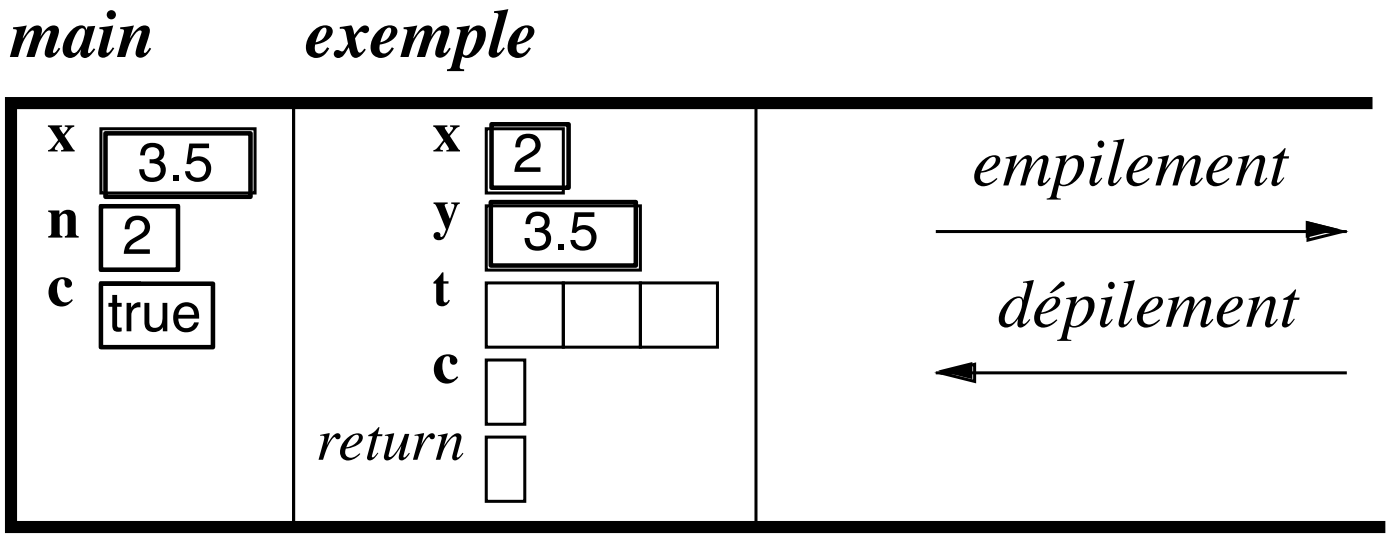
Ce programme s'exécute par :

1. Appel initial du `main` $\Rightarrow$ on empile son contexte afin de pouvoir l'exécuter,
2. Suite de l'exécution de `main` ...
3. Toujours dans `main` : appel vers `exemple` $\Rightarrow$ on empile son contexte en recopiant la valeur des paramètres passés lors de l'appel.

Le contexte de `exemple` est empilé après celui de `main` .

Pile d'exécution

Appel depuis `main` : `exemple(n,x)`



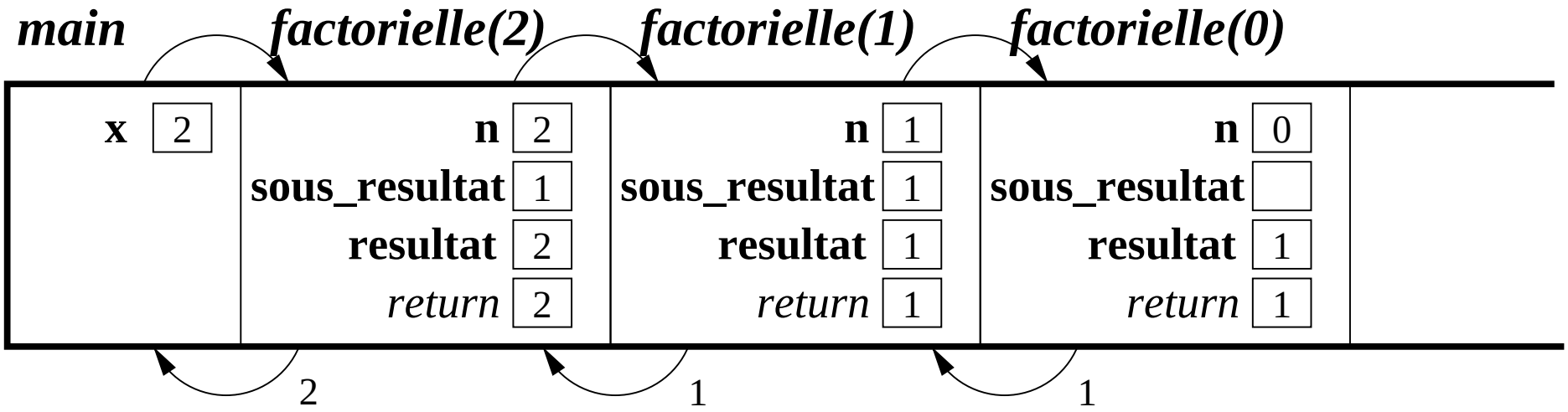
factorielle : version 1

```
static int factorielle (int n){
  int resultat;
  if(n==0) resultat = 1;
  else {
    int sous_resultat = factorielle (n-1);
    resultat = sous_resultat * n;
  }
  return resultat;
}

public static void main(String[] args){
    int x = 2;
    Terminal.ecrireString("Factorielle de "+x+" = ");
    x = factorielle(x);
    Terminal.ecrireIntln(x);
}
```

Exécution de `factorielle(x)` dans la pile

Appel (depuis main) : `factorielle(x)`



Chaque appel a son contexte propre, avec son paramètre, variables locales et valeur de retour.

Terminaison d'une méthode récursive

Un appel récursif peut se traduire en un 2ème appel récursif qui lui-même fait un 3ème appel, etc, donnant lieu à une *suite infinie* d'appel récursifs.

Conclusion :

- Pour tout appel récursif il doit toujours y avoir une *condition d'arrêt*.
- Elle ne garantît à elle seule la terminaison (condition nécessaire mais non suffisante).

Terminaison de factorielle

Cette version de factorielle, termine-t-elle ?

```
int factorielle (int n){
  int sous_resultat = factorielle (n-1);
  int resultat = sous_resultat * n;
  return resultat;
}
```

Pourquoi ?

Terminaison de factorielle

Et celle-ci ?

```
int factorielle (int n){
  if(n==1) return 1;
  else return factorielle(n-1) * n;
}
```

Y-a-t-il des cas où elle ne termine pas ?
Lesquels ?

Terminaison de factorielle

Cette version de factorielle est incorrecte.

- Elle boucle pour $n \leq 0$;
- en particulier, elle ne sait pas traiter le cas $n = 0$ qui est pourtant bien défini ;
- elle boucle pour tous les n non définis ($n < 0$).

```
int factorielle (int n){
  if(n==1) return 1;
  else return factorielle(n-1) * n;
}
```

Comment faire mieux ?

Version complète de factorielle

Cette version termine pour tous les cas de n.
Si $n < 0$, elle provoque une erreur.
Si $n = 0$, elle donne la réponse correcte.

```
public class TestFactorielle{
  static int factorielle (int n){
    int resultat;
    if(n<0) throw new MauvaisParametre();
    else if(n==0) resultat = 1;
    else {
      int sous_resultat = factorielle (n-1);
      resultat = sous_resultat * n;
    }
    return resultat;
  }
```

Test de factorielle

```
public class TestFactorielle{
  static int factorielle (int n){
    int resultat;
    if(n<0) throw new MauvaisParametre();
    else if(n==0) resultat = 1;
    else {
      int sous_resultat = factorielle (n-1);
      resultat = sous_resultat * n;
    }
    return resultat;
  }
 public static void main(String[] args){
    Terminal.ecrireString("Entrez un entier positif : ");
    int x = Terminal.lireInt();
    Terminal.ecrireStringln(x + "! = " + factorielle(x));
  }
}
class MauvaisParametre extends Error{}
```

Concevoir un algorithme récursif

1. Trouver une décomposition récursive du problème.
   1. Trouver *l'élément de récursivité qui sera passé en paramètre* : il doit changer (par exemple diminuer) lors de chaque appel ;
   2. Trouver son *cas le plus simple* : on doit lui associer une *réponse non récursive*. Ce sera le *cas d'arrêt* ;
   3. Compléter avec le *cas général* : sa solution *contient un appel récursif* vers un cas plus simple.
2. Vérifier que tous les cas sont pris en compte ;
3. Vérifier que la cas d'arrêt est nécessairement atteint ;

Exemple d'algorithme récursif

Calculer le nombre d'occurrences d'un caractère dans une chaîne s.

1. Décomposition récursive :
   1. une chaîne $s = c_1$ (1er caractère) + s_{reste} (reste de la chaîne).

$$s = \boxed{c_1 \mid s_{reste}}$$

   2. *élément de récursivité* : la sous-chaîne à traiter ;
   3. *cas le plus simple* : la sous-chaîne à traiter est vide $\Rightarrow$ 0 occurrences (réponse non récursive) ;
   4. *cas général* : si c_1 est le caractère cherché, on le compte + appel récursif sur s_{reste} $\Rightarrow$ addition des 2 résultats ;

Remarques

$$s = \boxed{c_1 \mid s_{reste}}$$

$$n = (1 \ ou \ 0) + n_1$$

1. la sous-chaîne à traiter *diminue de taille* pour chaque appel ;
2. *cas le plus simple* : quand il n'y a plus de sous-chaîne à traiter (elle devient vide) ;
3. *cas général* : le nombre d'occurrences $n =$:
   - compter 1 si c_1 est le caractère cherché et 0 sinon ;
   - calcuer $n_1 =$ nombre d'occurrences dans s_{reste}

Occurrences d'un caractère dans une chaîne

```
static int nbOccurrences (char c, String s){
  if(s.length()==0) return 0;  //cas d'arret
  else{
    String sReste= s.substring(1,s.length());
    int n1 = nbOccurrences(c,sReste);
    if(s.charAt(0)==c) return(1 + n1);
    else return(n1);
  }
}
```

Procédures récursives

Afficher en ordre inverse les éléments d'un tableau par une itération récursive.

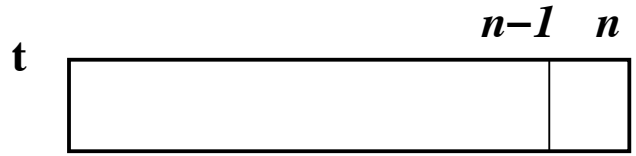
Itération : sur l'indice final du tableau restant à afficher.

```
static void affichageInverse(int[] t, int k){
  if(k==0) Terminal.ecrireIntln(t[0]);
  else{
    Terminal.ecrireIntln(t[k]);
    affichageInverse(t, k-1);
  }
}
```

Affichage récursif d'un tableau

Décomposition récursive :

1. *élément de récursivité* : indice k de l'élément à afficher (1ère fois $k = n$)



2. *cas le plus simple* : $k = 0 \Rightarrow$ dernier affichage ;
3. *cas général* :
   1. Affichage de $t[k]$
   2. Affichage récursif pour l'indice $k - 1$

Récursivité vs Boucles

Pour effectuer des actions ou calculs répètes, on peut choisir entre un codage récursif ou à l'aide de boucles.

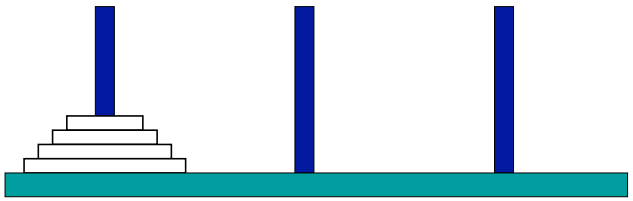
1. tout algorithme récursif peut être écrit avec des boucles et inversement ;
2. les solutions avec boucles sont *à priori*, plus efficaces (cela dépend du compilateur) ;
3. certains algorithmes sont très difficiles à écrire *autrement* qu'en style récursif ;
4. pour d'autres, le choix est soit indifférent, soit plus facile avec boucles $\Rightarrow$ choisir les boucles ;

Raisonnement récursif

Les problèmes dont la solution est obtenue par un raisonnement récursif sont beaucoup plus faciles à écrire récursivement.

Exemple : Le problème des tours de Hanoi.

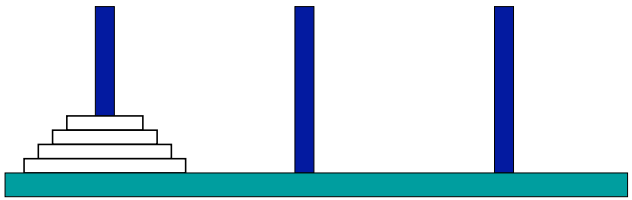
- n disques de tailles différentes ;
- pouvant être empilés sur 3 piliers ;
- au départ, tous les disques sont empilés dans le pilier de gauche, en ordre décroissant de taille (du bas vers le haut) ;



Les tours de Hanoi

But du jeu : reconstituer la pile de n disques, dans le même ordre, sur le pilier de droite.

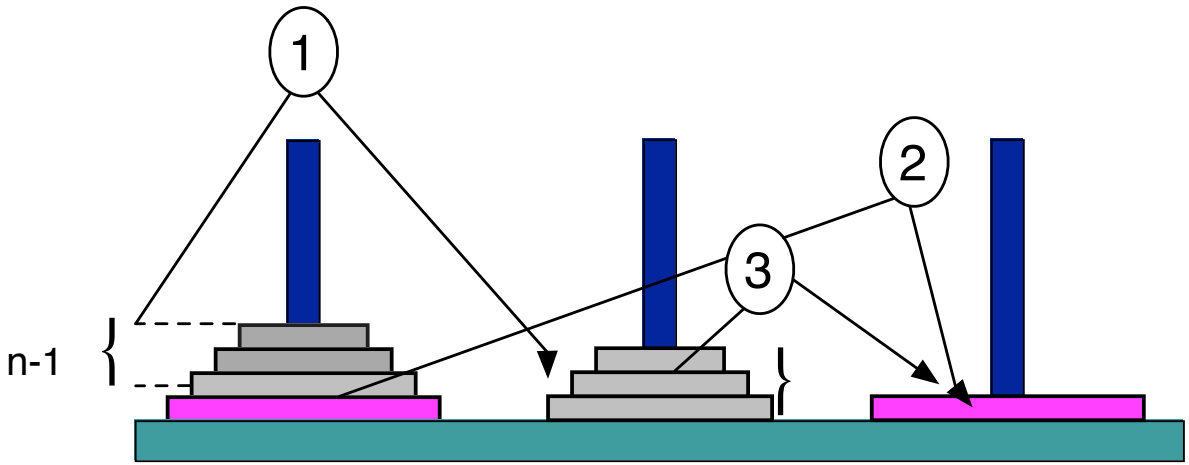
- on peut déplacer un disque seulement s'il est au sommet de sa pile (non recouvert par un autre disque) ;
- aucun disque ne peut être placé sur un autre plus petit ;



Solution récursive

Problème : déplacer n disques du pilier de source vers le pilier de destination en utilisant un pilier intermédiaire.

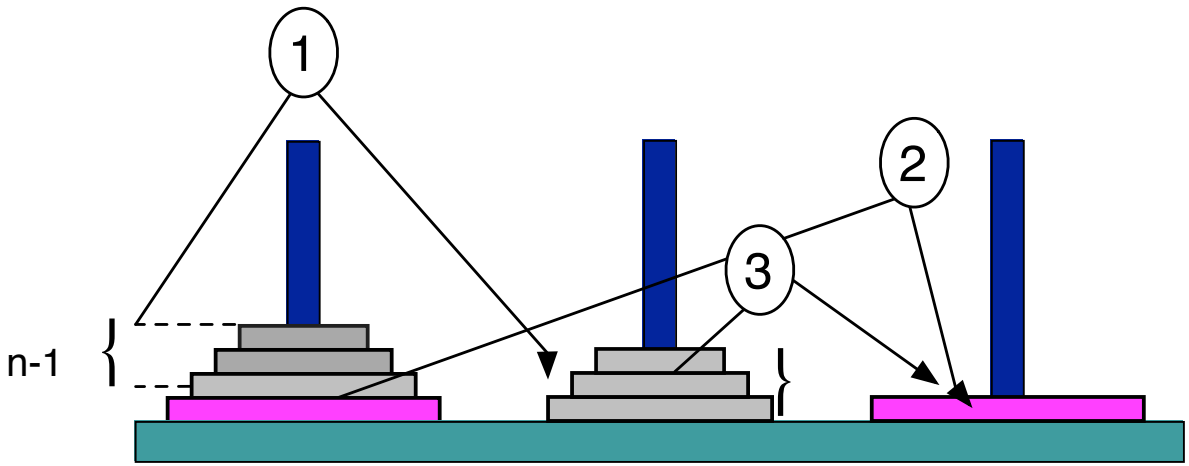
1. déplacer $n - 1$ disques de gauche vers milieu en passant par droite.
2. déplacer 1 disque de gauche vers droite.
3. déplacer $n - 1$ disques de milieu vers droite en passant par gauche.



Solution récursive

Problème : `deplace(n,source,dest,inter)`

1. `deplace(n-1,gauche,milieu,droite)`
2. `deplace(1,gauche,droite,milieu)`
3. `deplace(n-1,milieu,droite,gauche)`



Solution

```
static void deplaceUn(String source, String dest){
  Terminal.ecrireStringln(source + "␣-␣" + dest);
}

static void deplaceT(int n,String s,String d,String in){
  if(n==1) deplaceUn(s, d);
  else{
    deplaceT(n-1, s, in, d);
    deplaceUn(s, d);
    deplaceT(n-1, in, d, s);
}

public static void main(String[] args){
  Terminal.ecrireString("Combien␣de␣disques␣?␣");
  int n = Terminal.lireInt();
  deplaceT(n, "gauche", "droite", "milieu");
}
```

Solution : affichages

Ce programme affiche la suite des mouvements d'un pilier :

```
% java Hanoi
Combien de disques ? 3
gauche - droite
gauche - milieu
droite - milieu
gauche - droite
milieu - gauche
milieu - droite
gauche - droite
```

Une structure de données récursive : les arbres

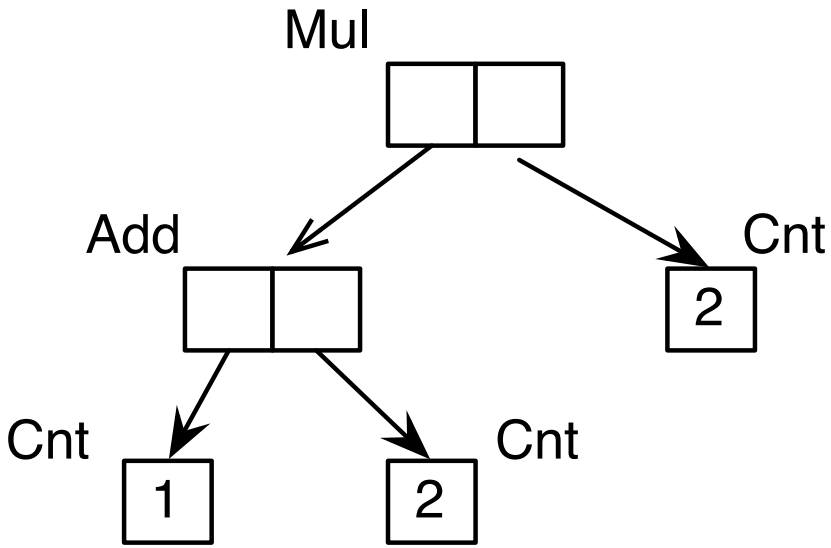
Arbre

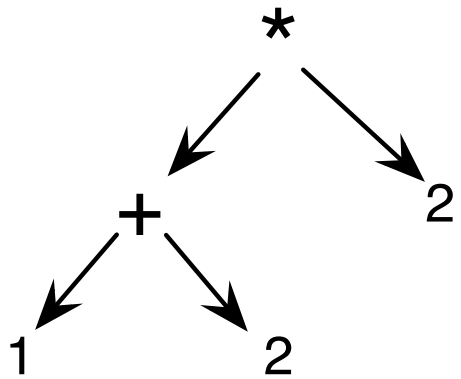
Type de données récursif composé de *noeuds internes*, d'*arêtes* entre deux noeuds, d'une *racine*, et de *feuilles* :

- *racine* : noeud d'origine ;
- *noeuds internes* : chacun a un ou plusieurs *sous-arbres* ;
- *feuilles* : noeuds terminaux de l'arbre(sans sous-arbres) ;

Expressions arithmétiques avec constantes

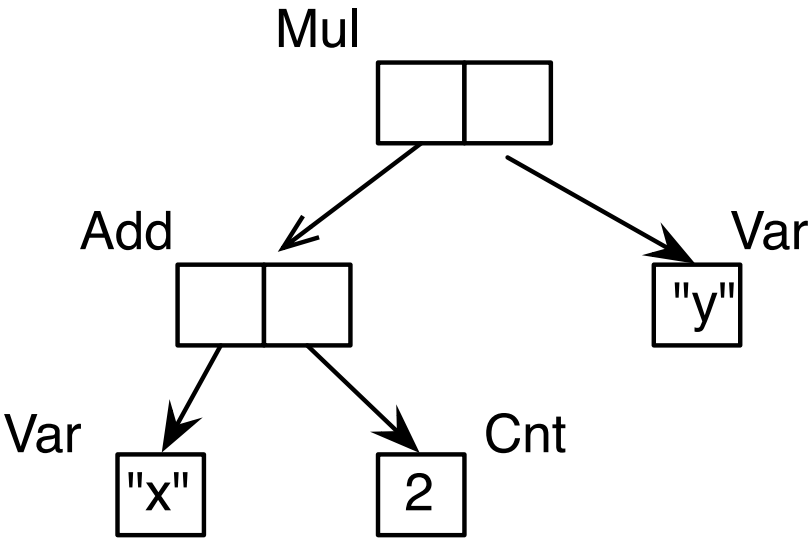
Expression :$(1 + 2) * 2$

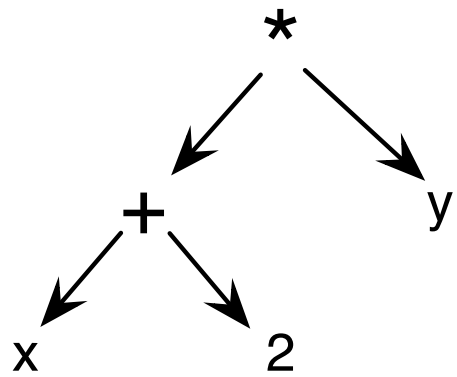




Expressions avec variables

Expression b = $(x + 2) * y$ représentée ici par un arbre :





Une expression arithmétique sous forme d'arbre

Trois cas :

- les expression arithmétiques : un arbre avec opérateur à la racine et 2 sous-arbres pour les opérandes ;
- les constantes : un entier ;
- les variables : une chaîne (nom de la variable)

Quels opérations sur les expressions modélisées de la sorte ?

- afficher une expression ;
- évaluer (obtenir la valeur de l'expression) :
  - besoin d'un contexte d'évaluation pour obtenir la valeur des variables ;
- créer une expression à partir d'une chaîne, puis l'évaluer.

Interface Expr

Toutes les formes d'expressions ont un point commun : on veut pouvoir calculer leur valeur dans un *contexte d'évaluation* donné.

- Opération `int val(Contexte c)` :
  - retourne un entier correspondant à la valeur calculée pour l'expression.
  - prend en paramètre un `Contexte c` :
    - ★ sorte de dictionnaire, où l'on stocke les noms de variables pouvant apparaître dans les expressions, avec leurs valeurs.

```
public interface Expr {
  /** Valeur de l'expression dans le contexte c */
  int val(Contexte c);
}
```

Expressions constantes (entiers)

```
public class Entier implements Expr{
  private int v;  // valeur de la constante
  public Entier (int i) { v = i;}

  public int val(Contexte c) {return v;}

  public String toString() {
    return ""+v;
  }
}
```

- la valeur d'une constante est elle-même.
- on re-définit également la méthode `toString()`

Opérateurs

Nous utilisons un type énuméré (qui donne la liste de constantes à utiliser en tant qu'opérateurs dans nos formules).

```
public enum Op {
    Plus,
    Sub,
    Mul,
    Div;
}
```

- les opérateurs pouvant apparaître dans une expression de type « opération arithmétique »

Opérations arithmétiques

```
public class Operation implements Expr{
 private Op op;           // operateur
 private Expr e1, e2;     // operandes

 public Operation(Op o, Expr a1, Expr a2){
      op=o; e1=a1; e2=a2; }

 public int val(Contexte env) {
    switch (op) {
     case Plus: return e1.val(env) + e2.val(env) ;
     case Sub : return e1.val(env) - e2.val(env) ;
     case Mul : return e1.val(env) * e2.val(env) ;
     case Div : return e1.val(env) / e2.val(env) ;
    }
  throw new Error ("eval : arbre Exp incorrect") ;
}
```

Opérations(suite)

Construire récursivement la chaîne associée à une expression.

```
public String toString(){
  switch (op) {
    case Plus:
      return "+(␣" +e1.toString()+",␣"+ e2.toString()+")"
    case Sub:
      return "-(␣"+e1.toString()+",␣"+e2.toString()+")" ;
    case Mul:
      return "*(␣"+e1.toString()+",␣"+e2.toString()+")" ;
    case Div:
      return "/("+e1.toString()+",␣"+e2.toString()+")" ;
  }
  throw new Error ("eval␣:␣arbre␣Exp␣incorrect") ;
}
```

Exemples

Dans cet exemple, le contexte est vide :

```
// Contexte vide
 Contexte c = new Contexte();
// Expression constante
 Expr deux = new Entier(2);
 // Expression operation d'addition
 Expr dp2 = new Operation (Op.Plus, deux, deux);

 // Afficher et evaluer l'expression
 Terminal.ecrireStringln(dp2.toString() +
                         " = " + dp2.val(c);
```

Affiche : `(2 + 2) = 4`

Une clase pour le contexte d'évaluation

Permet d'associer un nom de variable (String) a sa valeur (entier) dans une table d'associations.

```
public class Contexte {
  private Map<String,Integer> env =
        new HashMap<String,Integer>();

 /** Ajouter une variable */
  public void add(String v, int i){
    env.put(v, e);
  }
 /** Obtenir la valeur d'une variable */
  public int get(String v){
   return env.get(v);
  }
}
```

Expressions « variable »

```
public class Variable implements Expr{
  private String name;  // nom de la variable

  public Variable(String n) { name = n;}

  public String toString(){ return name;}

 /** La valeur d'une variable s'obtient
     en interrogeant le contexte argument */
  public int val(Contexte c){
    return c.get(name);
  }
```

Exemples avec variables

On ajoute dans un contexte la variable "X" avec valeur 7 :

```
Contexte c = new Contexte();    // Contexte vide
Expr deux = new Entier(2);      // Constante
Expr x = new Variable ("X");    // Variable X
c.add("X", 7);                  // Ajouter X et sa valeur
// Expression avec variable
Expr xfois2 = new Operation (Op.Mul, x, deux);

// Evaluer l'expression dans c
Terminal.ecrireStringln(xfois2.toString()
                    + " = " + xfois2.val(c);
```

Affiche : `(X * 2) = 14`

Version avec »pointeurs «

Principe :

- une variable est associée à une expression (et non plus à un entier).

```
public class Contexte {
 private Map<String,Expr> env=
        new HashMap<String,Expr>();

 public void add(String v, Expr e){... }
}
```

Que faire si :

```
  Contexte c = new Contexte();
  c.add("X", new Operation(un, new Variable ("X"));
```

Empêcher l'introduction d'un cycle

Principe : On refuse l'ajout d'une paire (x,e) dans le contexte, lorsque le nom de variable x apparaît dans l'expression e.

```
public class Contexte {
  private Map<String,Expr> env = new HashMap<String, Expr>(

   /* Echoue si e contient le nom de variable v (cycle) */
  public void add(String v, Expr e){
    if (!e.containsVar(v, this))
       env.put(v, e);
    else
      throw new Error("Ajout impossible: cycle");
  }
   ...
}
```

Conséquence : Chaque expression doit fournir une méthode qui teste si un nom de variable est dedans...

Nouvelle interface Expr

Nouvelle opération `containsVar` : pour tester si tentative d'introduction d'un cycle.

```
public interface Expr {
  /** Valeur de l'expression dans le contexte c  */
  int val(Contexte c);

  /** Teste si le nom de variable n apparait dans
   *  cette expression */
  boolean containsVar(String n, Contexte c);
}
```

containsVar

Pour les opérations : regarder dans les opérandes.

```
public class Operation implements Expr{
   ...
   public boolean containsVar(String n, Contexte c){
       return e1.containsVar(n,c) || e2.containsVar(n,c);
   }
```

Pour les constantes : toujours « false »

```
public class Entier implements Expr{
   ...
   public boolean containsVar(String n, Contexte c){
     return false;
   }
}
```

containsVar pour les variables

```
public class Variable implements Expr{
  ...
  /** Teste si l'expression associee a cette variable
   *  dans le contexte c contient le nom n:
   *    - soit le nom de cette variable egal a n
   *    - soit, chercher l'expression pour cette
   *      variable dans c. Si elle existe,
   *      tester si celle-ci contient le nom n. */
  public boolean containsVar(String n, Contexte c){
     if (this.name.equals(n))   return true;
     Expr en = c.get(name);
     if (en == null)
        return false;
     else
        return en.containsVar(n, c);
   }
```