Récurrence

Ecriture de fonctions et d'actions récursives

Réalisé dans le cadre de l'UNRRA avec le soutien financier de la Région Rhône-Alpes

Rappel : preuve par récurrence

Soit P une propriété à vérifier $\forall\ x \geq 0$

1. On vérifie la propriété pour le(s) cas de base : P(0) est vrai, P(1) vrai, …
2. Hypothèse de récurrence :
   - On suppose P vraie au rang k: P(k) vrai, k > 0
3. On démontre que si P est vraie au rang k, alors P est vraie au rang k+1 :

   $P(k)$ vrai $\Rightarrow$ $P(k+1)$ vrai
4. Conclusion :

   (1) et (3) $\Rightarrow$ P(x) vrai, $\forall\ x \geq 0$

Seconde forme de preuve par récurrence

Soit P une propriété à vérifier $\forall\ x \geq 0$

1. On vérifie la propriété pour le(s) cas de base : P(0) est vrai, P(1) vrai, …
2. Hypothèse de récurrence :
   - On suppose P vraie $\forall\ x \leq k,\ k > 0$
3. On démontre que si P est vraie jusqu'au rang k, alors P est vraie au rang k+1 :

   $P(X)$ vrai $\forall\ x \leq k \Rightarrow P(k+1)$ vrai
4. Conclusion :

   (1) et (3) $\Rightarrow$ P(x) vrai, $\forall\ x \geq 0$

Définition d'un ensemble par récurrence

- Définition d'un ensemble en extension

  E = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

- Définition d'un ensemble en compréhension

  Par exemple, l'ensemble I des nombre impairs :

  $x \in I$ ssi $\exists k \in \mathbf{Z}$ tel que : $x = 2k + 1$

- Définition d'un ensemble par récurrence
  1. Base : on donne la liste les éléments de base qui vont servir à la construction des autres éléments
  2. Récurrence : formulation récurrente d'une règle de construction d'un élément de l'ensemble à partir d'éléments déjà construits
  3. Conclusion : l'ensemble est formé des seuls éléments de base, et des éléments construits en appliquant la règle de construction énoncée

Définition d'un ensemble par récurrence

Exemple : l'ensemble I des nombres impairs

1. Base : $1 \in I$
2. Récurrence, règles de construction:

   si $x \in I$, alors $(x+2) \in I$

   si $x \in I$, alors $(x-2) \in I$

1. Conclusion : I est formé de 1 et des entiers obtenus en appliquant la règle un nombre quelconque de fois

Définition d'un ensemble par récurrence

Exemple : l'ensemble L* des chaînes formées de lettres majuscules

Soit L l'ensemble des lettres majuscules

L = { 'A', 'B', 'C', 'D', … 'Z' }

- Base : $"" \in L^*$
- Récurrence : $ch \in L^*$ et $c \in L$ $\Rightarrow ch \bullet c \in L^*$
- Conclusion : $x \in L^*$ si x = "" ou si x est construit en appliquant une ou plusieurs fois la règle de construction

Fonction récursive

A partir de la définition récurrente d'une fonction, on peut écrire la fonction récursive correspondante.

Exemple : calcul de la factorielle

profil : fact : N → N*

base : fact(0) = 1

réc. : fact(n) = fact(n-1) * n, ∀ n ∈ N*

fonction **fact**(x : entier ≥ 0) → entier > 0

// fact(x) renvoie factorielle de x : x!

lexique de fact

x : entier ≥ 0 // paramètre : valeur dont on calcule la factorielle

f : entier ≥ 0 // valeur calculée: x!

algorithme de fact

selon x

x = 0 : f ← 1

x > 0 : f ← fact(x-1) * x

fselon

renvoyer(f)

Fonction récursive

Autre écriture possible de la fonction fact :

fonction **fact**(x : entier ≥ 0) → entier > 0
// fact(n) renvoie factorielle de n : n!
lexique de fact
  x : entier ≥ 0   // paramètre : valeur dont on calcule la factorielle
algorithme de fact
 selon x
    x ≤ 1 : renvoyer(1)
    x > 1 : renvoyer(fact(x-1) * x)
 fselon

Démonstration de la correction de la fonction récursive: par récurrence

```
fonction fact(x : entier ≥ 0) → entier > 0
// fact(n) renvoie factorielle de n : n!
lexique de fact
  x : entier ≥ 0    // paramètre : valeur dont on calcule la factorielle
  f : entier ≥ 0    // valeur calculée: x!
algorithme de fact
 selon x
      x = 0 : f ← 1
      x > 0 : f ← fact(x-1) * x
 fselon
 renvoyer(f)
```

Base: fact(0) = 1, fact(1) = 1

Réc.: Hypothèse de récurrence : soit k > 0, on a : fact(k) = k!

Calculons fact(k+1) :

fact(k+1) = fact(k) * (k+1) = k! * (k+1) = (k+1)!

Conclusion:

si fact(k) renvoie k!, alors fact(k+1) renvoie (k+1)!

et comme fact renvoie la valeur correcte pour le cas de base,

fact(x) renvoie x! $\forall$ x ≥ 0

Fonctionnement d'une fonction récursive

Pour comprendre l'exécution d'un algorithme récursif, on réécrit l'algorithme en remplaçant l'appel de la fonction par le texte algorithmique qui lui correspond:

fact(4) : f ← fact(3) * 4

fact(3) : f ← fact(2) * 3

fact(2) : f ← fact(1) * 2

fact(1) : f ← fact(0) * 1

fact(0): renvoyer(1)

Fonctionnement d'une fonction récursive

Pour comprendre l'exécution d'un algorithme récursif, on réécrit l'algorithme en remplaçant l'appel de la fonction par le texte algorithmique qui lui correspond:

fact(4) : f ← 6 * 4 ; renvoyer(24)

fact(3) : f ← 2 * 3 ; renvoyer(6)

fact(2) : f ← 1 * 2 ; renvoyer(2)

fact(1) : f ← 1 * 1 ; renvoyer(1)

fact(0): renvoyer(1)

Etapes de construction d'un algorithme récursif

1. Spécification mathématique
   - Définition récursive du domaine de définition
   - Description des relations de récurrence de la fonction
2. Ecriture de la fonction récursive
3. Preuve de la correction et de la terminaison
   - Vérification de la cohérence des appels
   - Preuve par récurrence de la correction et de la terminaison
4. Evaluation de l'algorithme en termes
   - Nombre d'opérations
   - Nombre d'appels engendrés

Evaluation de l'algorithme de fact

```
fonction fact(x : entier ≥ 0) → entier > 0
// fact(n) renvoie factorielle de n : n!
lexique de fact
  x : entier ≥ 0   // paramètre : valeur dont on calcule la factorielle
algorithme de fact
 selon x
     x ≤ 1 : renvoyer(1)
     x > 1 : renvoyer(fact(x-1) * x)
 fselon
```

- Nombre d'appels engendrés par fact(n) :
  - si n ≤ 1, pas d'appel engendré
  - si n > 1, (n-1) appels engendrés
- Nombre d'opérations pour fact(n) :
  - si n ≤ 1, une comparaison
  - si n > 1, (n-1) soustractions et multiplications
    n comparaisons

Exercice : nombre de A

Ecrire la fonction récursive nba qui renvoie le nombre de 'A' d'une chaine de caractères

1. Spécification mathématique
   - Définition récursive du domaine de définition, C* l'ensemble des chaines de caractères
     - Base : "" $\in$ C*
     - Réc. : x $\in$ C, ch $\in$ C* $\Rightarrow$ x $\circ$ ch $\in$ C*
   - Description des relations de récurrence de la fonction nba: on se base sur la définition récurrente du domaine de définition

     profil: nba : C* $\rightarrow$ N

     base : nba("") = 0

     rec. : nba(x $\circ$ ch) = nba(ch) si x $\neq$ 'A'

     nba(x $\circ$ ch) = nba(ch)+1 si x = 'A'

Exercice : nombre de A

2. Ecriture de la fonction récursive

```
fonction nba(x : chaine) → entier ≥ 0
// nba(x) renvoie le nombre de 'A' contenus dans x
lexique de nba
  x : chaine  // paramètre : chaine examinée
algorithme de nba
 selon x
    x = "" : renvoyer(0)
    x ≠ "" : si pre(x) = 'A'
                alors  renvoyer(nba(fin(x)) + 1)
                sinon renvoyer(nba(fin(x))
               fsi
 fselon
```

Exercice : nombre de A

3. Preuve de la correction et de la terminaison
   - Vérification de la cohérence des appels
   - Preuve par récurrence de la correction et de la terminaison

```
fonction nba(x : chaine) → entier ≥ 0
// nba(x) renvoie le nombre de 'A' contenus dans x
lexique de nba
  x : chaine    // paramètre : chaine examinée
algorithme de nba
 selon x
    x = "" : renvoyer(0)
    x ≠ "" : si pre(x) = 'A'
                alors  renvoyer(nba(fin(x)) + 1)
                sinon renvoyer(nba(fin(x)))
                fsi
 fselon
```

Exercice : nombre de A

4. Evaluation de l'algorithme en termes
   - Nombre d'opérations
   - Nombre d'appels engendrés

```
fonction nba(x : chaine) → entier ≥ 0
// nba(x) renvoie le nombre de 'A' contenus dans x
lexique de nba
  x : chaine    // paramètre : chaine examinée
algorithme de nba
 selon x
    x = "" : renvoyer(0)
    x ≠ "" : si pre(x) = 'A'
                  alors  renvoyer(nba(fin(x)) + 1)
                  sinon renvoyer(nba(fin(x))
                  fsi
 fselon
```

Nombre de A dans un tableau

- On veut écrire une version de la fonction **nba** qui calcule le nombre de A présents dans un tableau de caractères.
- La fonction récursive s'applique à une partie du tableau (sous-tableau) caractérisé par sa borne inférieure et sa borne supérieure
- La fonction récursive doit donc comporter 3 paramètres :
  - Le tableau dans lequel on compte le nombre de 'A'
  - La borne inférieure du sous-tableau
  - La borne supérieure du sous-tableau

Nombre de A dans un tableau
version 1

```
fonction nba(t : tableau sur [0..Lmax-1] de caractères,
             bi, bs : entiers sur 0..Lmax-1) → entier ≥ 0
// nba(t, bi, bs) renvoie le nombre de 'A' contenus dans t[bi..bs]
// t[bi..bs] est vide si bi>bs
lexique de nba
 t : tableau sur [0..Lmax-1] de caractères // paramètre : séq. Examinée
 bi : entiers sur 0..Lmax // paramètres : borne inf. de l'intervalle de t
 bs : entiers sur 0..Lmax-1 // paramètres : borne sup. de l'intervalle de t
algorithme de nba
 selon bi, bs
  bi > bs : renvoyer(0)
  bi ≤ bs : si t[bi] = 'A'
              alors  renvoyer(nba(t, bi+1, bs) + 1)
              sinon renvoyer(nba(t, bi+1, bs))
             fsi
 fselon
```

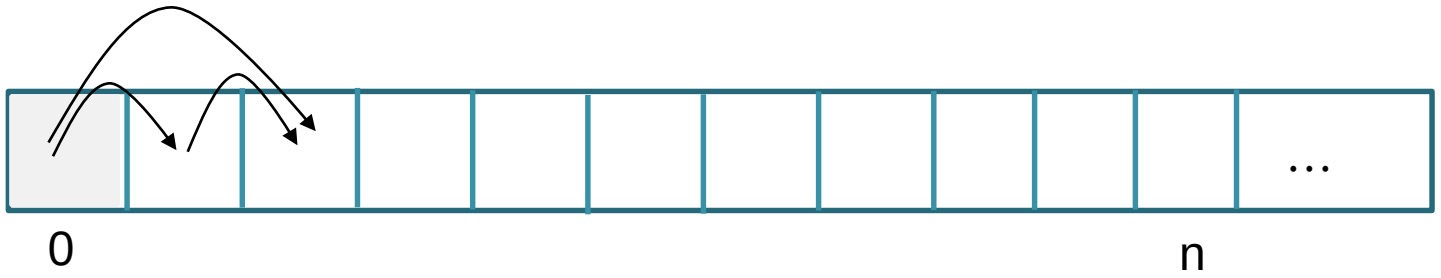
Nombre de A dans un tableau
version 2

```
fonction nba(t : tableau sur [0..Lmax-1] de caractères,
                 bs : entier sur -1..Lmax-1) → entier ≥ 0
// nba(t, bs) renvoie le nombre de 'A' contenus dans t[0..bs]
// t[0..bs] est vide si bs = -1
lexique de nba
 t : tableau sur [0..Lmax-1] de caractères // paramètre : séq. Examinée
 bs : entiers sur -1..Lmax-1 // paramètre : borne sup. de l'intervalle de t
algorithme de nba
 selon bs
  bs < 0 : renvoyer(0)
  bs ≥ 0 : si t[bs] = 'A'  alors  renvoyer(nba(t, bs-1) + 1)
                          sinon renvoyer(nba(t, bs-1))
           fsi
 fselon
```

Exercice 2

Ecrire une fonction récursive qui détermine si deux chaines de caractères sont égales

La Marelle



Des enfants jouent à la marelle : ils partent de la case n°0 et sautent jusqu'à la case n. De la case où ils se trouvent (soit k cette case) ils peuvent sauter

- soit à la case suivante (k+1),
- Soit à la case située juste après la case suivante (k+2)

Question : *De combien de manières différentes peuvent-ils atteindre la case n en partant de la case n°0 ?*

Soit nbc la fonction récursive qui calcule le nombre de chemins différents pour atteindre la case n.

Exercice : **Trouver l'expression récursive de la fonction nbc**

Expression récursive de la fonction nbc

nbc : $\mathbf{N^* \rightarrow N^*}$
nbc(n) = nombre de chemins différents pour atteindre la case n

nbc(0) = 1

nbc(1) = 1

nbc(2) = 2

nbc(3) = 3

nbc(4) = 5

nbc(5) = 8

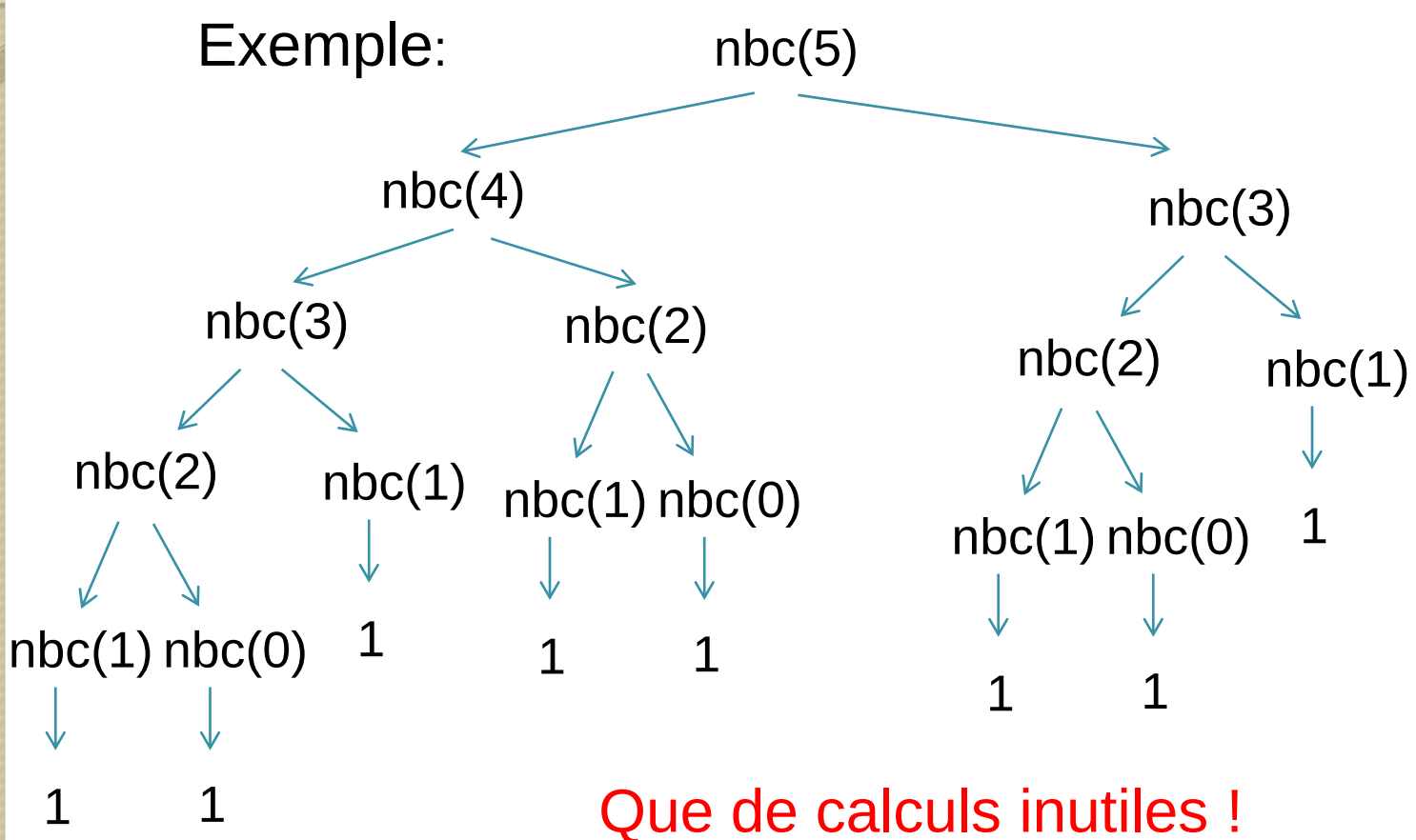
nbc(n) = nbc(n-1) + nbc(n-2)

nbc est la fonction de Fibonacci

Ecriture de la fonction nbc

```
fonction nbc(n : entier ≥ 0) → entier > 0
// nbc(n) renvoie le nombre de chemins différents pour atteindre la case n
lexique de nbc
  n : entier ≥ 0   // paramètre : n° de la case à atteindre
algorithme de nbc
 selon n
    n ≤ 1 : renvoyer (1)
    n > 1 : renvoyer ( nbc(n-1) + nbc(n-2) )
fselon
```

Nombre d'appels engendrés pour l'appel nbc(n)



Que de calculs inutiles !

Optimisation

Pour éviter de faire des calculs inutiles nous définissons la fonction nbc2 qui renvoie un doublet d'entiers : nbc2(n) = < nbc(n), nbc(n-1) >

Lexique partagé

Doublet : type agrégat

p, s : entiers // p est le premier entier, s le second

fagregat

nbc2 : N* → N* x N*

base : nbc2(1) = < 1, 1 >

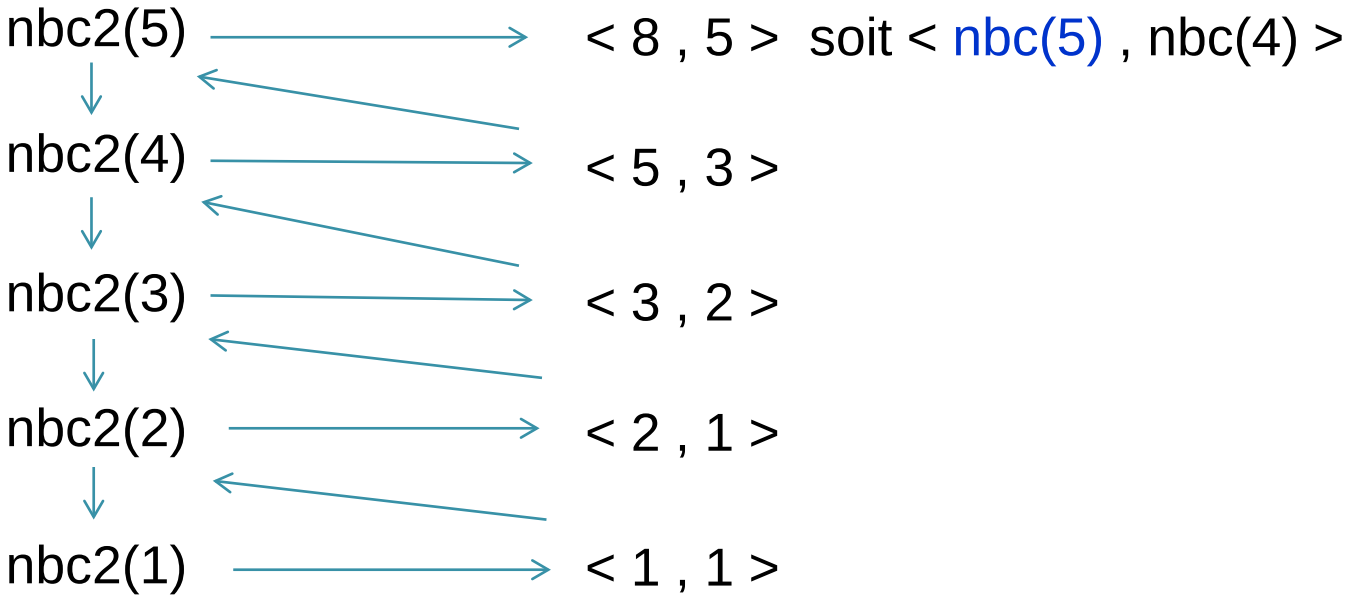
réc. : nbc2(n+1) = < nbc2(n).p + nbc2(n).s , nbc2(n).p >

Réalisation de la fonction nbc2

```
fonction nbc2(n : entier > 0) → Doublet
// nbc2(n) renvoie le doublet formé de nbc(n) et nbc(n-1)
lexique de nbc
  n : entier ≥ 0        // paramètre : n° de la case à atteindre
  dr : Doublet          // doublet renvoyé
  di : Doublet          // intermédiaire : résultat de l'appel récursif nbc2(n-1)
algorithme de nbc
 selon n
    n = 1 : dr.p ← 1 ; dr.s ← 1
    n > 1 : di ← nbc2(n-1)   // di = < nbc(n-1), nbc(n-2) >
                dr.p ← di.p + di.s ; dr.s ← di.p
fselon
renvoyer(dr)
```

Nombre d'appels engendrés pour l'appel nbc2(n)

Exemple:



Nouvelle version de la fonction nbc

```
fonction nbc(n : entier ≥ 0) → entier > 0
// nbc(n) renvoie le nombre de chemins différents pour atteindre la case n
lexique de nbc
  n : entier ≥ 0      // paramètre : n° de la case à atteindre
  d : Doublet         // intermédiaire : valeur renvoyée par nbc2(n)
  fonction utilisée : nbc2(n : entier > 0) → Doublet
algorithme de nbc
 selon n
    n ≤ 1 : renvoyer (1)
    n > 1 : d ← nbc2(n)   // d = < nbc(n), nbc(n-1) >
            renvoyer(d.p)
fselon
```

Nbc n'est pas récursive, mais fait appel à la fonction récursive nbc2

- Ajouter chEgales (en caché pas dans le poly)
- Programmer fibo/nbc, les deux versions en java
- Ajouter la fonction fact qui diverge et la programmer en Java + Python

Actions récursives

Actions récursives

Principe généraux de l'analyse récurrente

- Etant donné un problème P traitant une information d'un certain type, on cherche à réduire le problème en exprimant le traitement en termes du même traitement appliqué à une information de même type de taille plus petite.
- Il faut trouver un moyen de « découper » l'information en sous-information de même type
- Une analyse par cas doit ensuite faire apparaître le(s) cas où ce découpage n'est plus significatif : cas de base.

Exemple d'action récursive

Ecrire une action récursive afficherChaine qui affiche une chaine de caractères, sachant que l'on ne dispose que d'une action d'affichage d'un seul caractère nommée afficherCar :

```
action afficherCar (consulté x : caractère)
// Effet : affiche x sur l'écran
// E.I. : écran indifférent, x = c
// E.F. : le caractère c est affiché, x = c
```

Pour construire cette action on se base sur la définition récursive des chaînes :

- Base : $"" \in C^*$
- Récurrence : $ch \in C^*$ et $c \in C \Rightarrow ch \bullet c \in C^*$

Réalisation de afficherChaine

```
action afficherChaine(consulté x : chaine , modifié e : écran)
// effet : affiche la chaine x sur l'écran e
lexique de afficherChaine
  x : chaine          // paramètre : chaine à afficher
  e : écran           // paramètre : écran d'affichage
algorithme de afficherChaine
 si x ≠ ""
 alors
       afficherChaine(deb(x), e)
       e.afficherCar(der(x))
 fsi
```

afficherChaine("abcd", e)

afficherChaine("abc", e) ; e.afficherCar('d')

afficherChaine("ab", e) ; e.afficherCar('c')

afficherChaine("a", e) ; e.afficherCar('b')

afficherChaine("", e) ; e.afficherCar('a')

Autre solution pour afficherChaine

On se base sur la seconde définition récursive des chaînes :

- Base : "" $\in$ C*
- Réc. : x $\in$ C, ch $\in$ C* $\Rightarrow$ x $\circ$ ch $\in$ C*

```
action afficherChaine(consulté x : chaine , modifié e : écran)
// effet : affiche la chaine x sur l'écran e
lexique de afficherChaine
  x : chaine        // paramètre : chaine à afficher
  e : écran         // paramètre : écran d'affichage
algorithme de afficherChaine
 si x ≠ ""
 alors
       e.afficherCar(pre(x))
       afficherChaine(fin(x), e)
 fsi
```

Exemple 2

Ecrire une action récursive afficherEntier qui affiche un entier positif ou nul, sachant que l'on ne dispose que de l'action d'affichage d'un seul caractère afficherCar.

Pour construire cette action on se base sur une définition récursive des entiers naturels N:

- Base : {0,1,2,3,4,5,6,7,8,9} $\in$ N
- Récurrence :

  si x $\in$ N, si c $\in$ {0,1,2,3,4,5,6,7,8,9}

  alors (x*10+c) $\in$ N

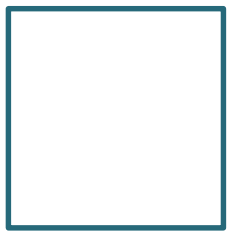
Le « découpage » consiste à décomposer un entier en un entier représentant les chiffres de poids fort, et le chiffre des unités

Réalisation de l'action afficherEntier

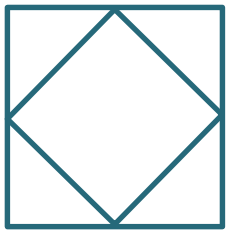
```
action afficherEntier(consulté x : entier ≥ 0 , modifié e : écran)
// effet : affiche l'entier x sur l'écran e
lexique de afficherChaine
  x : entier ≥ 0      // paramètre : entier à afficher
  e : écran           // paramètre : écran d'affichage
  fonction utilisée : convChiffreCar(c : entier entre 0 et 9) → caractère
algorithme de afficherEntier
 selon x
   x < 10 : e.afficherCar(convChiffreCar(x))
   x ≥ 10 : afficherEntier(x div 10, e) ;
            e.afficherCar(convChiffreCar(x mod 10))
fselon
```

Tracé d'une figure définie par récurrence à l'aide de la machine-tracés

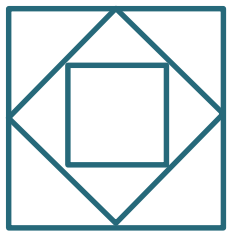
- Ecrire une action récursive carrésImbriqués qui trace la figure suivante à l'ordre **n**



n = 1

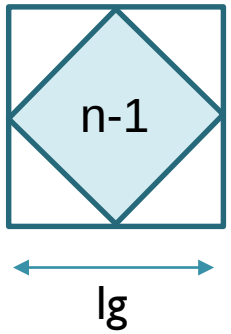


n = 2



n = 3

- Récurrence :
  1. Dessiner le carré extérieur
  2. Dessiner la figure intérieure à l'ordre n-1, de côté lg /$\sqrt{2}$



Réalisation de l'action carrésImbriqués

```
action carrésImbriqués(consulté n : entier > 0 , consulté lg : réel > 0, modifié m : machine-tracés)
// Effet : trace la figure à l'ordre n, avec une taille de côté lg, à partir de la position de la plume
// E.I. pp = s0 l'un des sommets du carré extérieur, pe = basse,
        cap = a0, direction du 1er côté du carré extérieur
// E.F. : figure à l'ordre N tracée, pp = s0, pe = basse, cap = a0
lexique de carrésImbriqués
  n : entier > 0            // paramètre : ordre de la figure à tracer
  lg : réel > 0             // paramètre : taille du côté du carré extérieur de la figure
  m : machine-tracés        // paramètre : machine-tracés utilisée
algorithme de carrésImbriqués
répéter 4 fois
   m.AV(lg) ; m.GA(90)
frépéter
// carré extérieur tracé, pp = s0, pe = basse, cap = a0
si n > 1
alors
     m.AV(lg/2); m.GA(45) ;    // pp = milieu du  1er côté, pe = basse, cap = a0 + 45
     carrésImbriqués(n-1, lg/racine(2), m)   // figure intérieure tracée, pp = milieu du  1er côté,
                                             // pe = basse, cap = a0+45
     m.DR(45) ; m.RE(lg/2)  // pp = s0, pe = basse, cap = a0
fsi
// figure tracée à l'ordre n, pp = s0, pe = basse, cap = a0
```

Utilisation de l'action carrésImbriqués

Lexique principal

```
  n : entier > 0  // donnée : ordre de la figure à tracer
  lg : réel > 0   // donnée : taille du côté du carré extérieur de la figure
  a : angle       // donnée : orientation du 1er côté du carré extérieur de la figure
  p : point       // donnée : premier sommet du carré extérieur de la figure
  m : machine-tracés   // machine-tracés utilisée
  cl : clavier         // pour la saisie des données
```

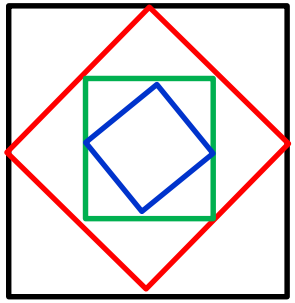
Algorithme principal

```
// saisie des données
cl.saisir(n, lg, a, p)   // n = n0, lg = l0, a = a0, p = s0
// positionnement de la plume plume au point p, direction a, position basse :
m.vider                  // pp = <0,0>, cap = 0, pe = haute
m.pos(p)                 // pp = s0
m.diriger(a)             // cap = a0
m.baisser                // pp = s0 , cap = a0 , pe = basse
// appel de l'action de tracé de la figure de carrés imbriqués, à l'ordre n
carrésImbriqués(n, lg, m)
// figure tracée à l'ordre n,  pp = s0 , cap = a0 , pe = basse
```

Exercice

- A l'aide de 4 crayons de couleur ☺ dessinez :
  - la partie de la figure dessinée quand N vaut 1
  - la partie de la figure dessinée quand N vaut 2
  - la partie de la figure dessinée quand N vaut 3
  - la partie de la figure dessinée quand N vaut 4

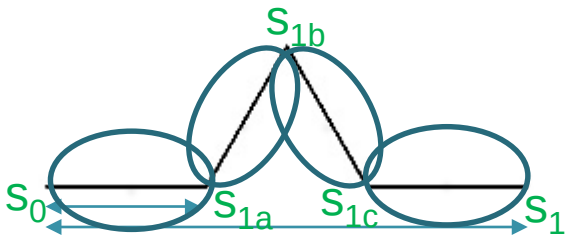
N = 4



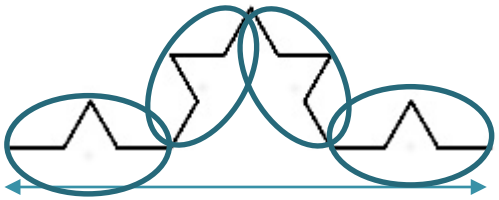
Courbe de Koch

- Ecrire l'action récursive Koch qui trace la figure suivante à l'ordre N

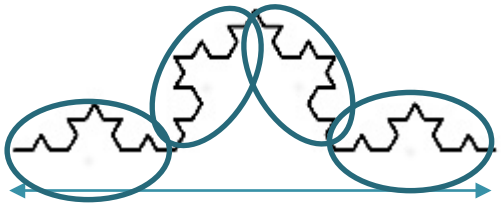
N = 1



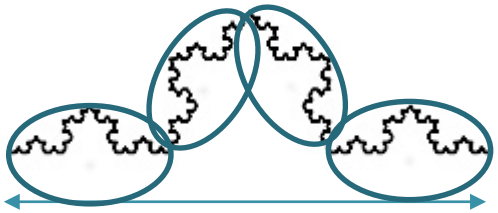
N = 2



N = 3



N = 4



N = 5

On a 4 appels récursifs

Action de tracé de la courbe de Koch à l'ordre n

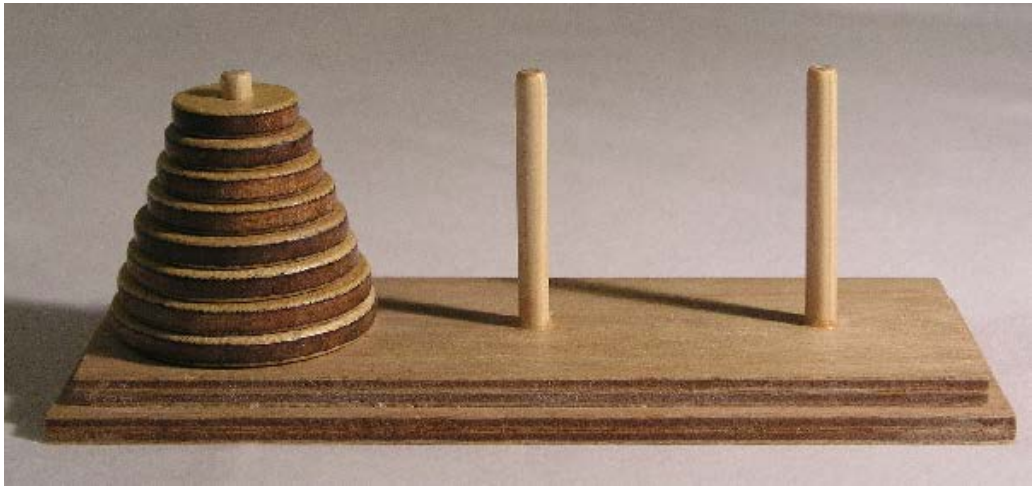
action **Koch** (consulté n : entier > 0 , consulté lg : réel > 0, modifié m : machine-tracés)
// effet : trace la courbe de Koch de longueur lg, à l'ordre n
// E.I. pp = s_0 extrémité gauche de la courbe, pe = basse, cap = a_0 direction de tracé
// E.F. : figure à l'ordre n tracée, pp = s_1 extrémité droite, pe = basse, cap = a_0
lexique de Koch
  n : entier > 0 // paramètre : ordre de la figure à tracer
  lg : réel > 0 // paramètre : longueur à l'horizontale de la courbe tracée
  m : machine-tracés // paramètre : machine-tracés utilisée
algorithme de Koch
  selon N
   n = 1 : m.AV(lg) ; // figure tracée à l'ordre 1, pp = s_1 , pe = basse, cap = a_0
   n > 1 : Koch (n-1, lg/3, m) // 1ère courbe tracée à l'ordre n-1, pp = s_{1a}
           m.GA(60) // cap = a_0 + 60
           Koch (n-1, lg/3, m) // 2ème courbe tracée à l'ordre n-1, pp = s_{1b} ,cap = a_0 + 60
           m.DR(120) // cap = a_0 - 60
           Koch (n-1, lg/3, m) // 3ème courbe tracée à l'ordre n-1, pp = s_{1c} ,cap = a_0 - 60
           m.GA(60) // cap = a_0
           Koch (n-1, lg/3, m) // 4ème courbe tracée à l'ordre n-1, pp = s_1 ,cap = a_0
           // figure tracée à l'ordre n, pp = s_1 , pe = basse, cap = a_0
  fselon

Exercice

N = 3

- A l'aide de 3 crayons de couleur ☺ dessinez :
  - la partie de la figure dessinée quand N vaut 1
  - la partie de la figure dessinée quand N vaut 2
  - la partie de la figure dessinée quand N vaut 3

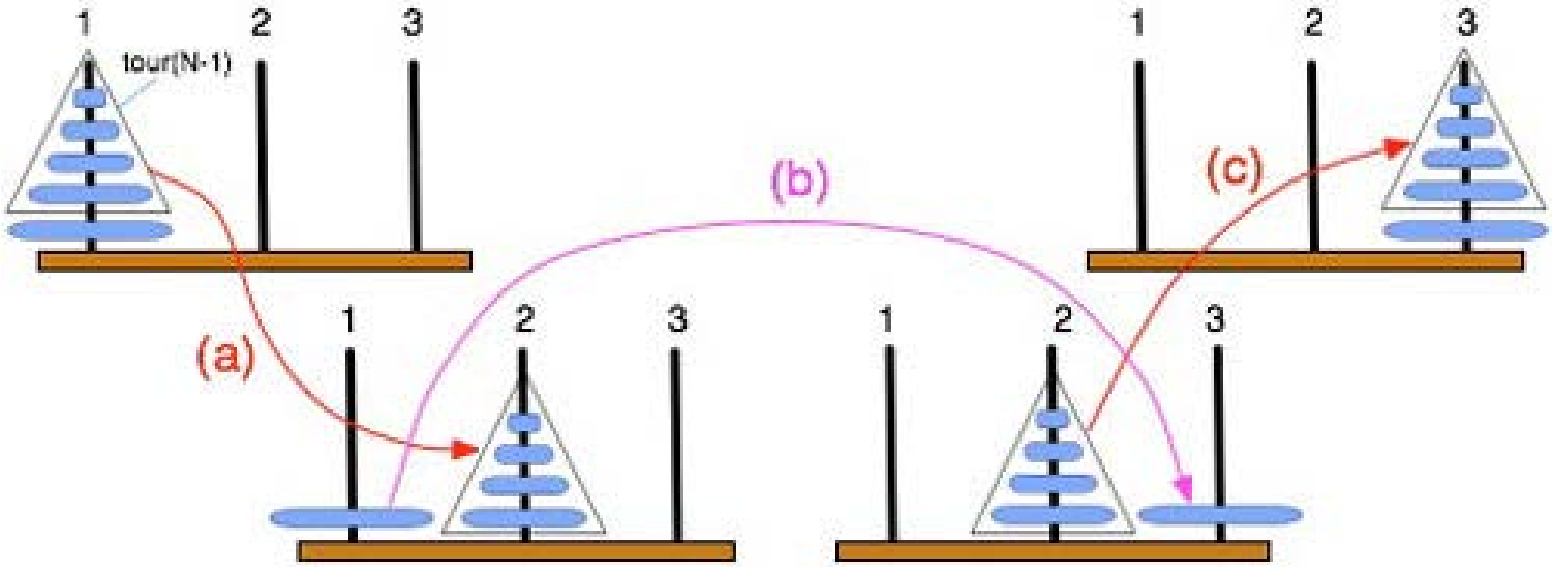
Les tours de Hanoï



Une légende orientale dit que la fin du monde surviendra quand sera achevé le déplacement d'une tour de 64 disques d'or, installée lors de la création du monde. À raison d'un mouvement par seconde, il y faudrait 584 milliards d'années…

Les tours de Hanoï : analyse

- On a un plot de départ, un plot d'arrivée et un plot intermédiaire
- Le problème pour une tour de N disques peut être résolu simplement si on sait le résoudre pour une tour de (N-1) disques. On transporte alors cette tour(N-1) du plot 1 au plot 2 avec le plot 3 comme intermédiaire (opération **a**). On déplace ensuite le disque restant du plot 1 au plot 3 (opération **b**). Enfin, on transporte la tour(N-1) du plot 2 au plot 3 avec le plot 1 comme intermédiaire (opération **c**), ce qui termine le travail.
- La figure ci-dessous schématise ces opérations.



Les tours de Hanoï
écriture de l'action récursive

On utilise une action élémentaire dep ayant pour objet de déplacer un disque d'un plot à un autre. Les plots sont désignés par des entiers : plot : type entier entre 1 et 3

```
action Hanoï (consulté n : entier > 0, consultés d, a, i : plots)
// Effet : déplace une tour de n disques du plot d au plot a
//          en utilisant le plot intermédiaire i
// E.I. le plot d contient n disques
// E.F. le plot a contient les n disques qui se trouvaient sur d
lexique de Hanoï
  n : entier > 0      // paramètre : hauteur de la tour à déplacer
  d, a, i : plots     // paramètres : plots de départ, d'arrivée et intermédiaire
  action utilisée : dep(x,y : plots)  // déplace un disque du plot x vers le plot y
algorithme de Hanoï
  selon n
   n = 1 : dep(d,a)
   n > 1 : Hanoï(n-1, d, i, a)          (a)
           dep(d,a)                     (b)
           Hanoï(n-1, i, a, d)          (c)
  fselon
```

Les tours de Hanoï
(version trouvée souvent dans la littérature)

action **Hanoï** (consulté N : entier ≥ 0, consultés D, A, I : plots)
// Effet : déplace une tour de N disques du plot D au plot A
// en utilisant le plot intermédiaire I
// E.I. le plot D contient N disques
// E.F. le plot A contient les N disques qui se trouvaient sur D
lexique de Hanoï
  N : entier > 0 // paramètre : hauteur de la tour à déplacer
  D, A, I : plots // paramètres : plots de départ, d'arrivée et intermédiaire
  action utilisée : dep(x,y : plots) // déplace un disque du plot x vers le plot y

algorithme de Hanoï
  si n > 0
  alors
    Hanoï(N-1, D, I, A) (a)
    dep(D,A) (b)
    Hanoï(N-1, I, A, D) (c)
  fsi