Langage fonctionnel et Récursivité

Programmation Fonctionnelle
Master 2 I2L apprentissage

Sébastien Verel
verel@lisic.univ-littoral.fr
http://www-lisic.univ-littoral.fr/~verel

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

Septembre 2018

Objectifs de la séance 02

- Savoir les principes de la programmation fonctionnelle
- Ecrire un algorithme récursif avec un seul test
- Etablir le lien entre définition par récurrence et algorithme récursif

Question principale du jour :

Comment écrire ce que l'on ne connait pas encore ?

Plan

Bibliographie

Basée sur le livre et les considérations de
Jean-Paul Roy
Université Nice Sophia Antipolis

Spécialiste de la programmation fonctionnelle :

`http://deptinfo.unice.fr/~roy/`

"Premiers cours de programmation avec Scheme - Du fonctionnel pur aux objets avec DrRacket", Ellipse, 432 pages, 2010.

Paradigmes de programmation

- Paradigme procédural ou impératif (1955) :
  - Machine à état,
  - Ordres (procédures) à la machine
- Paradigme fonctionnel (1958) :
  - Fonction, structure d'arbre
  - Composition de fonction, principe de récurrence
- Paradigme à objets (1967) :
  - Division du monde en classe, objets
  - Envoie de messages

- Styles complémentaires,
- Langages rarement bornés à un style (souplesse) et évoluent constamment
- Un informaticien jongle avec ces styles selon les besoins

Paradigme procédural, impératif

Le plus ancien, Fortran (1955) puis Pascal, C.

Principe

Programme = suite d'instructions pilotant un ordinateur

cf. Machine de Turing, machine à états.

Effets de bord

- Suppose de connaitre l'état de la machine,
- Modification zone de mémoire partagée : **effet de bord**
  Pensez aux applications distribuées où il y a un partage de ressource
- Source de nombreux bugs (mauvaise allocation, utilisation imprévue de variable, état imprévus, etc.)

- Monde virtuel de la mémoire ne cesse d'être modifiée (débutants !)
- Technique de réduction des interactions
  (programmation structurée, variable locale, encapsulation)

Paradigme procédural, impératif

Outil principal

Itération pour le calcul répétitif

- Programmation structurée : Pascal
- C est incontournable pour les professionnels au plus prés de l'architecture et de l'OS
- Considéré comme langage machine de haut niveau (nombreux compilateurs se contentent de traduire en C !)
- Evolution du C vers l'objet (C++, objective C, C♯)

Paradigme à objets

- Emergence avec Smalltalk (milieu 1970),
- Popularisé Java et C++ (années 90)
- Basé sur le besoin de Réutilisation du logicielle et d'Extensibilité

Principe

Notion de classes et d'objets

cf. math, ensembles et éléments

- Objets ont des capacités définies par des méthodes (fonctions),
- Exécuter par envoie de "message".
- Calcul décentralisé dans les objets.

Tous les langages populaires sont objets

Paradigme fonctionnel

- Réfuter la notion d'instruction,
- Pas de notion de temps,
- Pas de modification (comme en mathématique)

Principe

Eléments de base = fonctions

cf. lambda calcul (Alonzo Church, année 1930)

Paradigme fonctionnel

- S'appuie sur des mécanismes mentaux primitifs :
  Définition par récurrence
- Pas d'effet de bord, pas d'opérations d'affectation :
  En interne, pile pour stocker les informations temporaires (*notion d'environnement*)
- Calcul consiste en l'évaluation d'une fonction pour éviter toute modification "d'états"
- Résultat dépend seulement des entrées et non pas de l'état du programme
- *transparence référentielle* : remplacement des entrées par des fonctions qui ont les mêmes valeurs (voir ex. tableau)
- *Auto-référent* : capable de parler de lui-même
  Fonction, type de base
  Construction au cours de l'exécution

Paradigme fonctionnel

- Ancètre LISP (John McCarthy, 1958)
- Fortran (55) : besoin numérique militaire, les gestionnaires
- LISP : problèmes symbolique IA
  Démonstration automatique, planification jeu d'échec, compréhension langage naturel, calcul formel, mécanismes cognitifs

Exemples : emacs, autocad, PathFinder, certains jeux et les application d'erlang (telecom), langage Xquery/XSLT, scala avec le "big data", etc.

Paradigme fonctionnel

Différent impératif / fonctionnel

Impératif : $\{I_1 \ ; \ I_2\}$
Fonctionnel : $f_1 \ o \ f_2$

Approximativement

- Programmation fonctionnelle :
  Exécuter un programme : évaluer un arbre,
  et le résultat du programme est la valeur de l'arbre.
  Sens au programme = valeur
- Programmation impérative :
  Exécuter un programme : évaluer un arbre, dont la valeur n'est pas importante.
  Au fur et à mesure de l'évaluation de l'arbre, des actions sont produites sur le monde extérieur.
  Sens au programme = actions

Exemple du calcul du pgcd

Algorithme PGCD(a, b : entier) : entier
début
  si $b = 0$ **alors**
    retourner a
  sinon
    $c \leftarrow a$ modulo b
    retourner PGCD(b, c)
  fin si
fin

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$
5. $c = 70$

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$
5. $c = 70$
6. PGCD(462, 70)

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$
5. $c = 70$
6. PGCD(462, 70)
2. $b \neq 0$
5. $c = 42$
6. PGCD(70, 42)

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$
5. $c = 70$
6. PGCD(462, 70)
2. $b \neq 0$
5. $c = 42$
6. PGCD(70, 42)
2. $b \neq 0$
5. $c = 28$

Exécution de l'algorithme

Pour $a = 70$ et $b = 462$

1. PGCD(70, 462)
2. $b \neq 0$
5. $c = 70$
6. PGCD(462, 70)
2. $b \neq 0$
5. $c = 42$
6. PGCD(70, 42)
2. $b \neq 0$
5. $c = 28$
6. PGCD(42, 28)
2. $b \neq 0$
5. $c = 14$

Exécution de l'algorithme

Pour $a = 462$ et $b = 70$

6. PGCD(28, 14)
2. $b \neq 0$
5. $c = 0$

Exécution de l'algorithme

Pour $a = 462$ et $b = 70$

6. PGCD(28, 14)
2. $b \neq 0$
5. $c = 0$
6. PGCD(14, 0)
2. $b = 0$
3. PGCD $= 14$

Elements de l'algorithme

- Base : initialisation de la récurrence

  si $b = 0$ **alors**
  retourner a
  sinon
  ...
  fin si

Elements de l'algorithme

- Base : initialisation de la récurrence

  si $b = 0$ **alors**
  retourner a
  sinon
  ...
  fin si

- Hérédité : calcul à partir de paramètres plus "petits"

  si $b = 0$ **alors**
  ...
  sinon
  ...
  retourner PGCD(b, c)
  fin si

fin

Définition (informelle)

Algorithmes récursifs

Un algorithme récursif est un algorithme qui fait appel à lui-même dans le corps de sa propre définition.

Définition (informelle)

Algorithmes récursifs

Un algorithme récursif est un algorithme qui fait appel à lui-même dans le corps de sa propre définition.

Il existe deux types d'algorithmes récursifs :

- les algorithmes récursifs qui se terminent :
  au bout d'un nombre fini d'opérations, l'algorithme s'arrête.

Définition (informelle)

Algorithmes récursifs

Un algorithme récursif est un algorithme qui fait appel à lui-même dans le corps de sa propre définition.

Il existe deux types d'algorithmes récursifs :

- les algorithmes récursifs qui se terminent :
  au bout d'un nombre fini d'opérations, l'algorithme s'arrête.
- les algorithmes récursifs qui ne se terminent pas :
  on peut imaginer que l'algorithme continue "éternellement" de calculer.

Exemples

Algorithme suiteU(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteV$(n-1) + 2$
  fin si
fin

Exemples

Algorithme suiteU(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteV$(n-1) + 2$
  fin si
fin

suiteU n'est pas un algorithme résursif

Exemples

Algorithme suiteV(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteV(n-1) +2
  fin si
fin

Exemples

Algorithme suiteV(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteV(n-1) +2
  fin si
fin

suiteV est un algorithme résursif qui se termine

Exemples

Algorithme suiteW(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteW(n-3)+2
  fin si
fin

Exemples

Algorithme suiteW(n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 2
  sinon
    retourner $\frac{1}{2}$ suiteW(n-3)+2
  fin si
fin

suiteW(n) est un algorithme résursif :

- si n est un multiple de 3, suiteW se termine
- sinon suiteW ne se termine pas

Exemple d'exécution

Calcul de la puissance

Définition mathématique :

$$a^n = \begin{cases} 1, & \text{si } n = 0 \\ a^{n-1}.a, & \text{sinon} \end{cases}$$

Exemple d'exécution

Calcul de la puissance

Définition mathématique :

$$a^n = \begin{cases} 1, & \text{si } n = 0 \\ a^{n-1}.a, & \text{sinon} \end{cases}$$

Algorithme puissance(a : réel, n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 1
  sinon
    retourner puissance(a, n-1) * a
  fin si
fin

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)
3. ————-> puissance(2.5, 1)

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)
3. ————-> puissance(2.5, 1)
4. ——————-> puissance(2.5, 0) = 1

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)
3. ————-> puissance(2.5, 1)
4. ——————-> puissance(2.5, 0) = 1
5. ————-> puissance(2.5, 1) = 1 * 2.5 = 2.5

Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)
3. ————-> puissance(2.5, 1)
4. ——————-> puissance(2.5, 0) = 1
5. ————-> puissance(2.5, 1) = 1 * 2.5 = 2.5
6. ——> puissance(2.5, 2) = 2.5 * 2.5 = 6.25

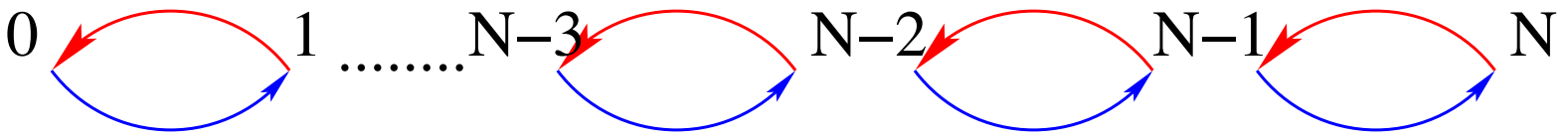
Exemple d'exécution

Calcul de la puissance

Calcul de $(2.5)^3$: $a = 2.5$ et $n = 3$.

1. puissance(2.5, 3)
2. ——> puissance(2.5, 2)
3. ————-> puissance(2.5, 1)
4. ——————-> puissance(2.5, 0) = 1
5. ————-> puissance(2.5, 1) = 1 * 2.5 = 2.5
6. ——> puissance(2.5, 2) = 2.5 * 2.5 = 6.25
7. puissance(2.5, 3) = 6.25 * 2.5 = 15.625

Principe d'exécution



- en rouge : parcours "aller"
- en bleu : parcours "retour"

En Haskell

En Haskell

```
{-------------------------------------
* calcul de la puissance d'un nombre
*
* arguments :
*   - a : nombre
*   - n : puissance
*
* resultat :
*   - a^n
-------------------------------------}
puissance :: (Fractional a, Integral b) => a -> b -> a
puissance x 0 = 1
puissance x n = puissance x (n - 1) * x
```

En Erlang

Encore mieux

```
{-------------------------------------
* calcul de la puissance d'un nombre
*
* arguments :
*   - a : nombre
*   - n : puissance
*
* resultat :
*   - a^n
-------------------------------------}
puissance :: (Fractional a, Integral b) => a -> b -> a
puissance x 0 = 1
puissance x n | n > 0 = puissance x (n - 1) * x
              | n < 0 = 1 / puissance x (-n)
```

PGCD en Haskell

PGCD en Haskell

```
{-------------------------------------
* calcul du pgcd de a et b
*
* arguments :
*   - a, b : nombres entiers
*
* resultat :
*   - pgcd(a, b)
-------------------------------------}
pgcd :: Integral p => p -> p -> p
pgcd a 0 = a
pgcd a b = pgcd b (mod a b)
```

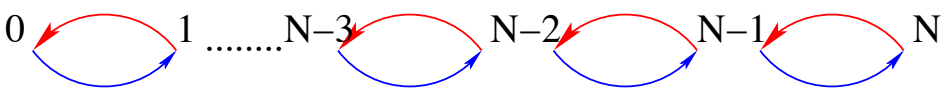
Intérêts

- bien adapté à la résolution de certains problèmes (et pas seulement mathématiques !)
- algorithmes souvent moins "laborieux" à écrire :
  moins de variables, beaucoup moins de boucles.
- une résolution par algorithme récursif nécessite souvent de prendre du recul pour résoudre le problème (avantage !)

A ne pas oublier !

- Hérédité : calcul à partir de paramètres plus "petits"

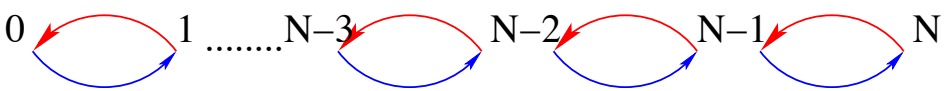
si $b = 0$ **alors**
...
sinon
...
retourner PGCD(b, $b \mod a$)
fin si
fin



A ne pas oublier !

- Base : initialisation de la récurrence

si $b = 0$ **alors**
  retourner a
sinon
  ...
fin si



Parallèle entre principe de récurrence et algorithme récursif

définition mathématique par récurrence
très proche
définition d'un algorithme récursif (cf. puissance)

Parallèle entre principe de récurrence et algorithme récursif

définition mathématique par récurrence
très proche
définition d'un algorithme récursif (cf. puissance)

Modes de calcul proches :

$$a^3 = a.a^2 = a.a.a^1 = a.a.a$$

Parallèle entre principe de récurrence et algorithme récursif

définition mathématique par récurrence
très proche
définition d'un algorithme récursif (cf. puissance)

Modes de calcul proches :

$$a^3 = a.a^2 = a.a.a^1 = a.a.a$$

Souvent, définition mathématique valide lorsque algorithme récursif associé se termine.

Correction du petit jeu "plus petit, plus grand"

Analyse

Correction du petit jeu "plus petit, plus grand"

Analyse

```
{----------------------------------------
 * Lance le jeu
 ----------------------------------------}
play :: IO ()
play = tour (rnd 100) 5

{----------------------------------------
 * Tirage d'un nombre aléatoire entre 1 et m
 *
 * arguments :
 *   - aucun
 *
 * resultat :
 *   - nombre entier pseudo-aléatoire
 ----------------------------------------}
rnd :: Int -> Int
rnd m = 42
    -- randomIO (1, m)
```

Résolution "plus petit, plus grand"

```
{-------------------------------------
 * tour: Permet de jouer un tour de jeu
 *
 * arguments :
 *   - x : nombre a deviner
 *   - n : nombre de tours possibles
 *
 * resultat :
 *   - aucun
 -------------------------------------}
tour :: Int -> Int -> IO ()
tour x 0 =
   putStrLn $ "Perdu loser. Le nombre était " ++ (show x)
```

Résolution "plus petit, plus grand"

```
{-------------------------------------
 * tour: Permet de jouer un tour de jeu
 *
 * arguments :
 *   - x : nombre a deviner
 *   - n : nombre de tours possibles
 *
 * resultat :
 *   - aucun
 -------------------------------------}
tour :: Int -> Int -> IO ()
tour x 0 =
   putStrLn $ "Perdu loser. Le nombre était " ++ (show x)

tour x n = do
   p <- askNumber
   if p == x
      then putStrLn "Ouais Youpi c'est gagné."
```

Résolution "plus petit, plus grand"

```
tour x n = do
   p <- askNumber
   if p == x
      then putStrLn "Ouais Youpi c'est gagné."
      else do
        if p < x
             then putStrLn "le nombre proposé est trop petit."
             else putStrLn "le nombre proposé est trop grand."
        tour x (n-1)
```

Résolution "plus petit, plus grand"

```
{-------------------------------------
 * Demande un nombre sur l'entrée standard
 *
 * arguments :
 *   - aucun
 *
 * resultat :
 *   - nombre entier dans un IO
 -------------------------------------}
askNumber :: IO Int
askNumber = do
   putStr "Proposez votre nombre : "
   pString <- getLine
   return (read pString :: Int)
```

Peux-t-on écrire le calcul de la puissance avec le même principe ?

```
{-------------------------------------
* calcul de la puissance d'un nombre
*
* arguments :
*   - a : nombre
*   - n : puissance
*
* resultat :
*   - a^n
-------------------------------------}
puissance :: (Fractional a, Integral b) => a -> b -> a
puissance x 0 = 1
puissance x n | n > 0 = puissance x (n - 1) * x
              | n < 0 = 1 / puissance x (-n)
```

Récursivité terminale : calcul de la puissance

Algorithme puissanceTerminale(a : réel, n : entier, acc : réel) : réel
début
  si $n = 0$ **alors**
    retourner acc
  sinon
    retourner puissanceTerminale(a, $n - 1$, $acc * a$)
  fin si
fin

Comment s'exécute cet algorithme ? puissanceTerminale(2.5, 3, 1)

- récursivité terminale : équivalent à une itération

Peux-t-on écrire le calcul de la puissance avec le même principe ?

Peux-t-on écrire le calcul de la puissance avec le même principe ?

```
{-------------------------------------
* calcul de la puissance d'un nombre
*
* arguments :
*   - a : nombre
*   - n : puissance
*   - accumulateur
*
* resultat :
*   - a^n
-------------------------------------}
puissanceTerm :: (Fractional a, Integral b) => a -> b -> a -> a
puissanceTerm x 0 acc = acc
puissanceTerm x n acc | n > 0 = puissanceTerm x (n - 1) (acc * x
                      | n < 0 = 1 / puissanceTerm x (-n) acc
```

Récursivité terminale

Comment s'exécute l'algorithme "puissanceTerminale(2.5, 3, 1)" ?

Récursivité terminale

Comment s'exécute l'algorithme "puissanceTerminale(2.5, 3, 1)" ?

Récursivité terminale

- Utilisation de paramètres auxiliaires (accumulateurs),
- Equivalent à une itération (dérécursivation),
- Meilleur complexité (moins de temps de calcul)
  Certains compilateurs compilent les récursivités en récursivité terminale
- Style "moins naturel"

Calcul de la factorielle

Factorielle

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n.(n-1)!, & \text{sinon} \end{cases}$$

Calcul de la factorielle

Factorielle

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n.(n-1)!, & \text{sinon} \end{cases}$$

Algorithme factorielle(n : entier) : entier
début
  si $n = 0$ **alors**
    retourner 1
  sinon
    retourner n * factorielle(n-1)
  fin si
fin

Factorielle en Haskell

Factorielle en Haskell

```
{-------------------------------------
 * calcul de la factorielle
 *
 * Argument :
 *   - n : nombre
 *
 * Resultat :
 *   - n!
-------------------------------------}
factorielle :: Integral a => a -> a
factorielle 0 = 1
factorielle n = factorielle (n-1) * n
```

Exécution de l'algorithme

Calcul de 3!.

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)
3. ————-> factorielle(1) = 1 * factorielle(0)

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)
3. ————-> factorielle(1) = 1 * factorielle(0)
4. ——————-> factorielle(0) = 1

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)
3. ————-> factorielle(1) = 1 * factorielle(0)
4. ——————-> factorielle(0) = 1
5. ————-> factorielle(1) = 1 * 1 = 1

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)
3. ————-> factorielle(1) = 1 * factorielle(0)
4. ——————-> factorielle(0) = 1
5. ————-> factorielle(1) = 1 * 1 = 1
6. ——> factorielle(2) = 2 * 1 = 2

Exécution de l'algorithme

Calcul de 3!.

1. factorielle(3) = 3 * factorielle(2)
2. ——> factorielle(2) = 2 * factorielle(1)
3. ————-> factorielle(1) = 1 * factorielle(0)
4. ——————-> factorielle(0) = 1
5. ————-> factorielle(1) = 1 * 1 = 1
6. ——> factorielle(2) = 2 * 1 = 2
7. factorielle(3) = 3 * 2 = 6

Quand utiliser un algorithme récursif ?

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?

Si oui, oui, oui
alors la résolution par un algorithme récursif est à envisager.

Tours de Hanoï (Édouard Lucas 1842 - 1891)

Le problème des tours de Hanoï consiste à déplacer N disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Comment résoudre ce problème ?

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?

Comment résoudre ce problème ?

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre de disques.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?

Comment résoudre ce problème ?

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre de disques.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui lorsque le nombre de disque est 1.
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?

Comment résoudre ce problème ?

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre de disques.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui lorsque le nombre de disque est 1.
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?
  $\longrightarrow$ Oui...

Algorithme récursif

Algorithme hanoi(n : entier, A : caractère, B : caractère, C : caractère) : rien

Algorithme récursif

Algorithme hanoi(n : entier, A : caractère, B : caractère, C : caractère) : rien
début
  si $n = 1$ **alors**
    écrire("déplacer ", A, " vers ", C)
  sinon
    hanoi(n-1, A, C, B) ;
    écrire("déplacer ", A, " vers ", C)
    hanoi(n-1, B, A, C) ;
  fin si
fin

En Haskell

Codage d'un mouvement

Couple de char :

```
('A', 'B')
```

Déplacement de la tige A vers la tige B

Ajouter un élément à une liste

```
Element : UneListe
```

Concaténer deux listes

```
List1 ++ List2
```

Hanoi en Haskell

```
{-------------------------------------
 * resolution des tours de Hanoi
 * Arguments :
 *   - n : nombre de disques du probleme
 *   - a : pic initial
 *   - b : pic intermediaire
 *   - c : pic final
 * Resultat :
 *    list des mouvements (couple)
-------------------------------------}
hanoi :: Int -> Char -> Char -> Char -> [ (Char, Char) ]
hanoi 0 _ _ _ = [ ]
hanoi n a b c = hanoi (n-1) a c b ++ ( (a,c) : hanoi (n-1) b a c )
```

Exécution de l'algorithme

- hanoi(2, a, b, c).
- hanoi(3, a, b, c)
- hanoi(4, a, b, c)

Quel est le nombre de déplacements en fonction de n ?

Exécution de l'algorithme

- hanoi(2, a, b, c).
- hanoi(3, a, b, c)
- hanoi(4, a, b, c)

Quel est le nombre de déplacements en fonction de n ?

Pour tout entier $n \geq 1$, $C_n = 2^n - 1$. A démontrer par récurrence...

Pour $n = 64$, les moines d'Hanoi y sont encore...

Temps de diffusion

Calculer $T_a(n) = a + 2a + 3a + \ldots + n.a$

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?

Temps de diffusion

Calculer $T_a(n) = a + 2a + 3a + \ldots + n.a$

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui n.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?

Temps de diffusion

Calculer $T_a(n) = a + 2a + 3a + \ldots + n.a$

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui n.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui pour $n = 0$ ou $n = 1$
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?

Temps de diffusion

Calculer $T_a(n) = a + 2a + 3a + \ldots + n.a$

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui n.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui pour $n = 0$ ou $n = 1$
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?
  $\longrightarrow$ Oui, $T_a(n) = T_a(n-1) + n.a$

Temps de diffusion

Algorithme

Algorithme diffusion(a : réel, n : entier) : réel
début
  si $n = 0$ **alors**
    retourner 0
  sinon
    retourner diffusion(a, n-1) $+n.a$
  fin si
fin

En Haskell

En Haskell

```
{-------------------------------------
 * Calcul du temps de diffusion
 *
 * Arguments :
 *   - n : nombre de particule a diffuser
 *   - a : temps pour diffusion 1 particule sur le bord
 *
 * Resultat :
 *   - temps total
-------------------------------------}
diffusion a 0 = 0
diffusion a n = (diffusion a (n-1)) + n * a
```

Exécution de l'algorithme

Calcul de $T_4(3)$

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)
3. ————-> diffusion(4, 1)

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)
3. ————-> diffusion(4, 1)
4. ——————-> diffusion(4, 0) = 0

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)
3. ————-> diffusion(4, 1)
4. ——————-> diffusion(4, 0) = 0
5. ————-> diffusion(4, 1) = 0 + 4 = 4

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)
3. ————-> diffusion(4, 1)
4. ——————-> diffusion(4, 0) = 0
5. ————-> diffusion(4, 1) = 0 + 4 = 4
6. ——> diffusion(4, 2) = 4 + 2.4 = 12

Exécution de l'algorithme

Calcul de $T_4(3)$

1. diffusion(4, 3)
2. ——> diffusion(4, 2)
3. ————-> diffusion(4, 1)
4. ——————-> diffusion(4, 0) = 0
5. ————-> diffusion(4, 1) = 0 + 4 = 4
6. ——> diffusion(4, 2) = 4 + 2.4 = 12
7. diffusion(4, 3) = 12 + 3.4 = 24

Complexité temporelle

Premier contact

Approximativement : nombre d'opérations élémentaires pour exécuter l'algorithme

Comparer les complexités des algorithmes hanoi et diffusion.

Régionnement du plan

Etant donné un nombre n de droites, calculer le nombre R_n maximum de régions du plan obtenus

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?

Régionnement du plan

Etant donné un nombre n de droites, calculer le nombre R_n maximum de régions du plan obtenus

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre n de droites.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?

Régionnement du plan

Etant donné un nombre n de droites, calculer le nombre R_n maximum de régions du plan obtenus

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre n de droites.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui pour $n = 0$ ou $n = 1$
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?

Régionnement du plan

Etant donné un nombre n de droites, calculer le nombre R_n maximum de régions du plan obtenus

- Est-ce que le problème dépend d'un (ou plusieurs) paramètre(s) ?
  $\longrightarrow$ Oui le nombre n de droites.
- Est-il possible de résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)" ?
  $\longrightarrow$ Oui pour $n = 0$ ou $n = 1$
- Est-il possible de résoudre le problème à l'aide de la résolution du problème portant sur une (des) "plus petite(s)" valeur(s) du paramètre ?
  $\longrightarrow$ Oui, en comptant le nombre régions ajoutées lorsqu'on ajoute une droite à $n - 1$ droites : une région supplémentaire par droite coupée, plus une dernière région.

Régionnement du plan

Algorithme

Algorithme region(n : entier) : entier
début
  si $n = 0$ **alors**
    retourner 1
  sinon
    retourner region(n-1) $+n$
  fin si
fin

En Haskell

En Haskell

```
{-------------------------------------
 * Calcul du nombre de region du plan
 *
 * Arguments :
 *   - n : nombre de droite
 *
 * Resultat :
 *   - nombre de region
-------------------------------------}
region :: (Integral a) => a -> a
region 0 = 1
region n = (region (n-1)) + n
```