CSI1502
Introduction au génie logiciel

Chapitre 11:
Récursion

Chapitre 11: Récursivité

- La récursivité est une technique fondamentale de programmation qui permet de résoudre élégamment certains types de problème.
- Objectifs du cours:
  - Apprendre à penser de manière récursive.
  - Apprendre à programmer de manière récursive.
  - Distinguer les cas d'utilisation de la récursivité et de l'itération.
  - Comprendre les exemples utilisant la récursivité.
  - Avoir un aperçu des fractales.

Qu'est-ce que la récursivité? Penser récursivement

- La *récursivité* est une technique de programmation dans laquelle une fonction s'appelle elle-même pour résoudre un problème.
- Avant d'appliquer la récusivité en programmation, il faut s'entraîner à penser récursivement.
- Considérons la liste d'entiers suivante:
  - `24, 88, 40, 37`
- Une liste peut être définie récursivement. Une idée ?

Définition récursive d'une liste

- Une liste peut être définie récursivement par:

```
A LIST is a:  number
      or a:   number  comma  LIST
```

- Ainsi un objet LIST peut être défini par un nombre ou par un nombre suivi d'une virgule et d'un autre objet LIST.
- Le concept d'objet LIST est utilisé dans la définition d'un objet LIST.

Définitions récursives

- La partie récursive de l'objet LIST est utilisée plusieurs fois; la partie non récursive terminant la définition de l'objet.
- Exemple:

```
number comma LIST
  24     ,   88, 40, 37

            number comma LIST
              88     ,   40, 37

                        number comma LIST
                          40     ,   37

                                    number
                                      37
```


Eviter les appels récursifs infinis: la condition d'arrêt

- Toutes les définitions récursives possèdent une condition d'arrêt i.e. une partie non récursive.
- Si cette dernière manque, une fonction récursive ne se terminera jamais (boucle de récursivité infinie).
- Le code d'une méthode récursive doit pouvoir traiter les deux cas:
  - Le cas correspondant à la condition d'arrêt.
  - Le cas récursif.
- *Quelle est la condition d'arrêt dans le cas LIST?*

Définitions récursives de fonctions mathématiques

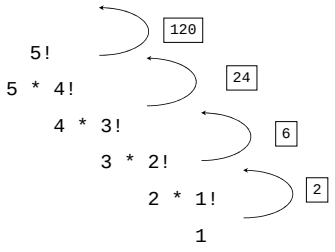
- N!, pour tout entier positif est défini comme étant le produit de tous les entiers de 1 à N inclus.
- Cette fonction peut être exprimée récursivement par:

```
1!  =  1
N!  =  N * (N-1)!
```

- Le concept de factorielle est défini à l'aide d'une autre factorielle **jusqu'à ce que le cas limite de 1! soit atteint**.

7

Définitions récursives: N!



8

Programmation récursive: autre exemple

- Considérez le problème consistant à calculer la somme des entiers de 1 à N inclus.
- Ce problème peut être exprimé récursivement par:

$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

$$= \text{etc.}$$

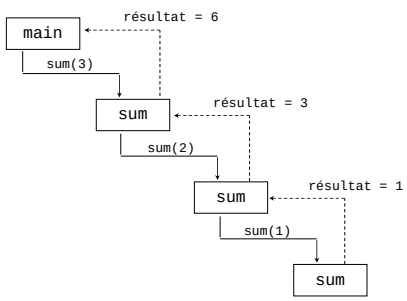
9

Programmtion récursive, calcul de la somme:

```
public int sum (int num)
  {
     int result;
     if (num == 1)
          result = 1;
     else
          result = num + sum (num - 1);
     return result;
  }
```

10

Programmtion récursive : Exécution du programme



11

Récursivité contre itération

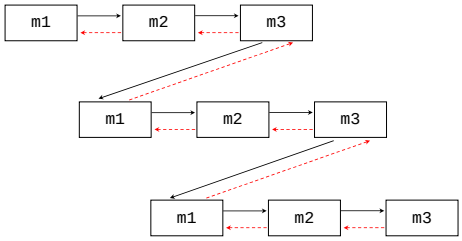
- Pouvoir résoudre récursivement un problème n'est pas toujours la panacée.
- Par exemple la somme (ou le produit) d'entiers de 1 à N peut être calculée à l'aide d'une boucle *for.*
- Vous devez être capables de déterminer quand une approche récursive est **appropriée** pour résoudre un problème.

12

Récursivité indirecte

- Une méthode s'appelant elle-même est considérée comme étant directement récursive.
- Cependant une autre possible existe: une méthode peut appelant une autre méthode qui elle-même en appellera une autre, ainsi de suite,... la méthode initiale étant finalement appelée.
- Par exemple une méthode `m1` peut appeler une méthode `m2`, qui appellera `m3`, cette dernière rappelant m1 jusqu'à ce qu'une condition de terminaison soit atteinte.
- Ceci est appelé une récursivité *indirecte* et demande autant de soins qu'une récursivité directe.
- De plus une récursivité *indirecte* est souvent plus dure à suivre et à déboguer.

Récursivité indirecte



Un exemple de récursivité: le parcours d'un labyrinthe.

- On peut utiliser la récursivité pour trouver un chemin dans un labyrinthe; un chemin pouvant être trouvé à partir d'une location arbitraire si l'on connaît les chemins menant aux locations voisines.
- Chaque location rencontrée sera marquée comme ayant été *visitée* et nous essayerons de trouver un chemin jusqu'à ses voisines non encore visitées.
- La récursivité garde en mémoire le chemin parcouru à travers le labyrinthe. Les conditions d'arrêt sont **déplacement interdit** et **destination finale atteinte**.

Parcours de labyrinthe: Grille labyrinthique et sortie

- La grille

```
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 0 0 1
0 0 0 0 1 0 1 0 1 0 1 0 0
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1
```

- Le résultat:

The maze was successfully traversed!

```
7 7 7 0 1 1 0 0 0 1 1 1 1
3 0 7 7 7 0 7 7 7 1 0 0 1
0 0 0 0 7 0 7 0 7 0 3 0 0
7 7 7 0 7 7 7 0 7 0 3 3 3
7 0 7 0 0 0 0 7 7 3 0 0 3
7 0 7 7 7 7 7 7 0 3 3 3 3
7 0 0 0 0 0 0 0 0 0 0 0 0
7 7 7 7 7 7 7 7 7 7 7 7 7
```


Traversée labyrinthique: MazeSearch.java

```
public class MazeSearch
{
   public static void main (String[] args)
   {
      Maze labyrinth = new Maze();

      System.out.println (labyrinth);

      if (labyrinth.traverse (0, 0))
         System.out.println ("The maze was successfully traversed!");
      else
         System.out.println ("There is no possible path.");

      System.out.println (labyrinth);
   }
}
```


Traversée labyrinthique : Maze.java

```
public class Maze
{
      private final int TRIED = 3;
      private final int PATH = 7;

      private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                               {1,0,1,1,1,0,1,1,1,1,0,0,1},
                               {0,0,0,0,1,0,1,0,1,0,1,0,0},
                               {1,1,1,0,1,1,1,0,1,0,1,1,1},
                               {1,0,1,0,0,0,0,1,1,1,0,0,1},
                               {1,0,1,1,1,1,1,1,0,1,1,1,1},
                               {1,0,0,0,0,0,0,0,0,0,0,0,0},
                               {1,1,1,1,1,1,1,1,1,1,1,1,1} };

Continued...
```

Traversée labyrinthique : Maze.java (suite.)

```
public boolean traverse (int row, int column)
{
        boolean done = false;

        if (valid (row, column))
        {
          grid[row][column] = TRIED;  // this cell has been tried

          if (row == grid.length-1 && column == grid[0].length-1)
            done = true;  // the maze is solved
          else
          {
            done = traverse (row+1, column);     // down
            if (!done)
              done = traverse (row, column+1);  // right
            if (!done)
              done = traverse (row-1, column);  // up
            if (!done)
              done = traverse (row, column-1);  // left
          }

          if (done)  // this location is part of the final path
            grid[row][column] = PATH;
        }
        return done;   }                                    suite...
```


Traversée labyrinthique : Maze.java (suite.)

```
  //-----------------------------------------------------------------
  //  Détermine si une location donnée est valide.
  //-----------------------------------------------------------------
  private boolean valid (int row, int column)
  {
    boolean result = false;

    // check if cell is in the bounds of the matrix
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[row].length)

      //  check if cell is not blocked and not previously tried
      if (grid[row][column] == 1)
        result = true;

    return result;
  }
                                                          suite...
```


Traversée labyrinthique : Maze.java

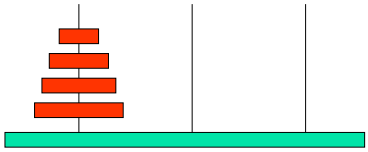
```
  //-----------------------------------------------------------------
  //  retourne le lqbyrinthe comme chaîne de caractères.
  //-----------------------------------------------------------------
  public String toString ()
  {
    String result = "\n";

    for (int row=0; row < grid.length; row++)
    {
      for (int column=0; column < grid[row].length; column++)
        result += grid[row][column] + "";
      result += "\n";
    }

    return result;
  }
}
```


Problème récursif classique: Tours de Hanoi

- Les tours de Hanoi sont un casse-tête composé de trois tiges verticales et de différents disques pouvant être insérés dessus.
- Le but est de déplacer tous les disques d'une tige à une autre en suivant les règles suivantes:
  - Ne bouger qu'un disque à la fois.
  - On ne peut placer un disque plus large sur un moins large.
  - Tous les disques doivent être autour de tiges sauf le disque manipulé.

Tours de Hanoi

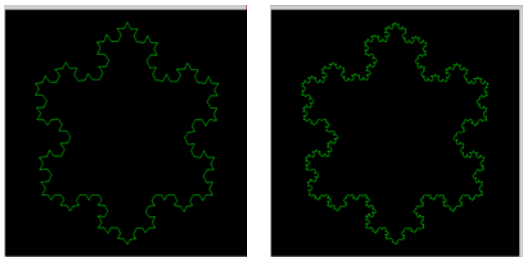
- Nous utilisons 3 tiges pour accomplir cette tâche.
- Voir p. 616 du livre de cours.



Récursivité en dessin: les fractales

- Une *fractale* est une forme géométrique consistant d'un motif se répétant à diverses échelles et orientations
- Le *flocon de Koch* est une fractale particulière commençant à partir d'un triangle équilatéral.
- Pour passer à l'ordre supérieur, le milieu de chaque arête est remplacé par 2 segments formant un angle prédéfini.

Fractales: Modélisation du chaos et caetera



25

Sommaire: Chapitre 11

- Objectifs du cours:
  - Apprendre à penser de manière récursive.
  - Apprendre à programmer de manière récursive.
  - Distinguer les cas d'utilisation de la récursivité et de l'itération.
  - Comprendre les exemples utilisant la récursivité.
  - Avoir un aperçu des fractales.

26