Chapitre 13

Récursivité

13.1 La notion de récursivité

13.1.1 La décomposition en sous-problèmes

Le processus d'analyse permet de décomposer un problème en sous-problèmes "plus simples". A leur tour, ces sous-problèmes seront décomposés jusqu'à un niveau d'opérations "élémentaires", faciles à réaliser.

Exemple : afficher un rapport avec les élèves d'une classe, groupés par la note obtenue à un examen, en ordre décroissant des notes et en ordre alphabétique pour chaque note.

Une décomposition possible de ce problème en sous-problèmes :
- introduction des données (noms des élèves et notes)
- tri des élèves en ordre décroissant des notes
- pour chaque note obtenue, en ordre décroissant, répéter
  - extraction des noms d'élèves qui ont obtenu cette note
  - calcul du nombre d'élèves qui ont obtenu cette note
  - tri de ces élèves en ordre alphabétique
  - affichage de la note, du nombre d'élèves qui ont cette note et des noms de ces élèves

Chacun de ces sous-problèmes pourra être décomposé à son tour en sous-sous-problèmes, etc. En programmation, la décomposition en sous-problèmes correspond au découpage d'un programme en sous-programmes ; a chaque sous-problème correspond un sous-programme.

La décomposition ci-dessus décrit *l'algorithme* de résolution du problème en utilisant l'appel aux sous-programmes qui traitent les sous-problèmes.

13.1.2 Décomposition récursive

Dans certains cas, le sous-problème est une illustration du problème initial, mais pour un cas "plus simple". Par conséquent, *la solution du problème s'exprime par rapport à elle-même* ! On appelle ce phénomène **récursivité**.

Exemple : calcul de la factorielle d'une valeur entière positive n (n ! = 1 * 2 * .... * n)

Mais, n ! = (1 * 2 * .... * (n-1)) * n, donc **n ! = (n-1) ! * n**.

Pour calculer la valeur de n !, il suffit donc de savoir calculer (n-1) ! et ensuite de multiplier cette valeur par n. Le sous-problème du calcul de (n-1) ! est le même que le problème initial, mais pour un cas "plus simple", car $n - 1 < n$.

Conclusion : le problème a été réduit à lui-même, mais pour un cas plus simple.

En programmation, le sous-programme qui traite le problème fait un appel à lui-même (!) pour traiter le cas plus simple, ce qui revient à un appel avec des paramètres différents ("plus simples"). On appelle cela *un appel récursif*.

La fonction `factorielle` aura alors la forme suivante :

Listing 13.1 – (lien vers le code brut)

```
1 int factorielle (int n){
2   int sous_resultat = factorielle (n-1); //appel recursif
3   int resultat = sous_resultat * n;
4   return resultat;
5 }
```

Remarque très importante : un appel récursif peut produire lui-même un autre appel récursif, etc, ce qui peut mener à une suite infinie d'appels. En l'occurrence, la fonction ci-dessus *est incorrecte*. Il faut arrêter la suite d'appels au moment où le sous-problème peut être résolu directement. Dans la fonction précédente, il faut s'arrêter (ne pas faire d'appel récursif) si n = 1, car dans ce cas on connaît le résultat (1 ! = 1).

Conclusion : ***dans tout sous-programme récursif il faut une condition d'arrêt***.

La version correcte de la fonction `factorielle` est la suivante :

Listing 13.2 – (lien vers le code brut)

```
1 int factorielle (int n){
2   int resultat;
3   if(n==1) resultat = 1;
4   else {
5     int sous_resultat = factorielle (n-1); //appel recursif
6     resultat = sous_resultat * n;
7   }
8   return resultat;
9 }
```

Remarque : la fonction factorielle peut être écrite d'une manière plus compacte, mais nous avons préféré cette version pour mieux illustrer les sections suivantes. Normalement, on n'a pas besoin des variables locales et on peut écrire directement :

Listing 13.3 – (lien vers le code brut)

```
1 int factorielle (int n){
2   if(n==1) return 1;
3   else return factorielle(n-1) * n;
4 }
```

Remarque : l'existence d'une condition d'arrêt ne signifie pas que l'appel récursif s'arrête grâce à celle-ci. Prenons l'exemple de l'appel `factorielle(-1)` : celà produit un appel à `factorielle(-2)`, qui produit un appel à `factorielle(-3)`, etc. La condition d'arrêt (n=1) n'est jamais atteinte et on obtient une suite infinie d'appels.

Conclusion : dans un sous-programme récursif, *il faut s'assurer que la condition d'arrêt est atteinte après un nombre fini d'appels.*

Remarque : la condition d'arrêt doit être choisie avec soin. Elle doit correspondre en principe au cas "le plus simple" qu'on veut traiter, sinon certains cas ne seront pas couverts par le sous-programme. Dans notre cas, la fonction `factorielle` ne sait pas traiter le cas n=0, qui est pourtant bien défini (0 ! = 1) et qui respecte la relation de décomposition récursive (1 ! = 0 ! * 1).

Le programme complet qui utilise la fonction `factorielle`, modifiée pour tenir compte des remarques ci-dessus, est le suivant :

Listing 13.4 – (lien vers le code brut)

```
1  public class TestFactorielle{
2    static int factorielle (int n){
3      int resultat;
4      if(n<0) throw new MauvaisParametre();
5      else if(n==0) resultat = 1;
6      else {
7        int sous_resultat = factorielle (n-1); //appel recursif
8        resultat = sous_resultat * n;
9      }
10     return resultat;
11   }
12
13   public static void main(String[] args){
14     Terminal.ecrireString("Entrez un entier positif : ");
15     int x = Terminal.lireInt();
16     Terminal.ecrireStringln(x + "! = " + factorielle(x));
17   }
18 }
19
20 class MauvaisParametre extends Error{}
```

13.1.3 Récursivité directe et indirecte

Quand un sous-programme fait appel à lui même, comme dans le cas de la factorielle, on appelle cela *récursivité directe*. Parfois, il est possible qu'un sous-programme A fasse appel à un sous-programme B, qui lui même fait appel au sous-programme A. Donc l'appel qui part de A atteint de nouveau A après être passé par B. On appelle cela *récursivité indirecte* (en fait, la suite d'appels qui part de A peut passer par plusieurs autres sous-programmes avant d'arriver de nouveau à A !).

Exemple : un exemple simple de récursivité indirecte est la définition récursive des nombres pairs et impairs. Un nombre n positif est pair si n-1 est impair ; un nombre n positif est impair si n-1 est pair. Les conditions d'arrêt sont données par les valeurs n=0, qui est paire et n=1, qui est impaire.

Voici le programme complet qui traite ce problème :

Listing 13.5 – (lien vers le code brut)

```
1  public class TestPairImpair{
```

```
2    static boolean pair(int n){
3      if(n<0) throw new MauvaisParametre();
4      else if(n==0) return true;
5      else if(n==1) return false;
6      else return impair(n-1);
7    }
8
9    static boolean impair(int n){
10     if(n<0) throw new MauvaisParametre();
11     else if(n==0) return false;
12     else if(n==1) return true;
13     else return pair(n-1);
14   }
15
16   public static void main(String[] args){
17     Terminal.ecrireString("Entrez un entier positif : ");
18     int x = Terminal.lireInt();
19     if(pair(x)) Terminal.ecrireStringln("nombre pair");
20     else Terminal.ecrireStringln("nombre impair");
21   }
22 }
23
24 class MauvaisParametre extends Error{}
```

La récursivité indirecte est produite par les fonctions `pair` et `impair` : `pair` appelle `impair` et `impair` appelle à son tour `pair`.

Un autre exemple, qui combine les deux types de récursivité, est présenté ci-dessous :

Exemple : calculer les suites de valeurs données par les relations suivantes :

$x_0 = 1$; $x_n = 2*y_{n-1} + x_{n-1}$

$y_0 = -1$; $y_n = 2*x_{n-1} + y_{n-1}$

Les fonctions récursives qui calculent les valeurs des suites x et y sont présentées ci-dessous. On retrouve dans chaque fonction à la fois de la récursivité directe (`x` fait appel à `x` et `y` à `y`) et de la récursivité indirecte (`x` fait appel à `y`, qui fait appel à `x`, etc). L'exemple ne traite pas les situations de mauvaises valeurs de paramètres (çàd $n < 0$).

Listing 13.6 – (lien vers le code brut)

```
1  int x (int n){
2    if(n==0) return 1;
3    else return 2*y(n-1) + x(n-1);
4  }
5
6  int y (int n){
7    if(n==0) return -1;
8    else return 2*x(n-1) + y(n-1);
9  }
```

13.2 Evaluation d'un appel récursif

Comment est-il possible d'appeler un sous-programme pendant que celui-ci est en train de s'exécuter ? Comment se fait-il que les données gérées par ces différents appels du même sous-programme ne se

"mélangent" pas ? Pour répondre à ces questions il faut d'abord comprendre le modèle de mémoire dans l'exécution des sous-programmes en Java.

13.2.1 Modèle de mémoire

Chaque sous-programme Java (le programme principal "main" aussi) utilise une zone de mémoire pour stocker *ses paramètres* et *ses variables locales*. De plus, une fonction réserve aussi dans sa zone de mémoire une place pour le résultat retourné. Prenons pour exemple la fonction suivante :

Listing 13.7 – (lien vers le code brut)

```
1  boolean exemple (int x, double y){
2    int [] t = new int[3];
3    char c;
4    ...
5  }
```

La zone de mémoire occupée par cette fonction est illustrée dans la figure 13.1

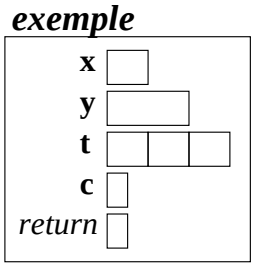

FIG. 13.1 – Zone de mémoire occupée par la fonction `exemple`

L'allocation de cette zone de mémoire se fait *au moment de l'appel du sous-programme*, dans une zone de mémoire spéciale du programme, appelé **la pile**. Les zones de mémoire des sous-programmes sont empilées suivant l'ordre des appels et dépilées dès que le sous-programme se termine. Par conséquent, la zone de mémoire d'un sous-programme n'existe physiquement *que pendant que le sous-programme est en cours d'exécution*.

Supposons que le programme principal qui appelle la fonction exemple a la forme suivante :

Listing 13.8 – (lien vers le code brut)

```
1  public class Principal{
2    static boolean exemple (int x, double y){ ... }
3    public static void main(String[] args){
4      double x;
5      int n;
6      boolean c;
7      .........
8      c = exemple(n, x);
9      .........
10   }
11 }
```

Le contenu de la pile pendant l'appel de la fonction `exemple` est présenté dans la figure 13.2.

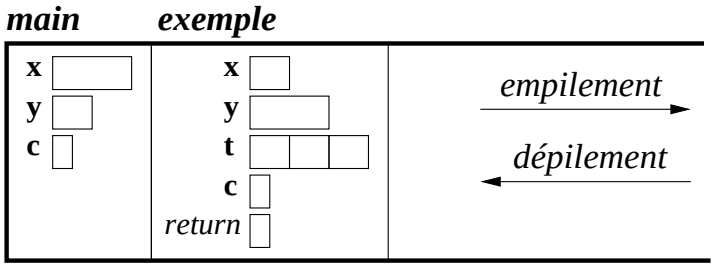

FIG. 13.2 – La pile pendant l'appel de fonction `exemple`

Avant l'appel, tout comme après la fin de l'exécution de la fonction `exemple`, la pile ne contient que la zone de `main`. La place libérée par `exemple` sera occupée par un éventuel appel ultérieur d'un autre sous-programme (peut-être même `exemple`, s'il est appelé plusieurs fois par `main` !).

Remarque : Il y a une séparation nette entre les variables locales des différents sous-programmes, car elles occupent des zones de mémoire distinctes (*ex.* les variables x, y et c).

Conclusion : la pile contient à un moment donné les zones de mémoire de l'enchaînement de sous-programmes en cours d'exécution, qui part du programme principal.

13.2.2 Déroulement des appels récursifs

Dans le cas des programmes récursifs, la pile est remplie par l'appel d'un même sous-programme (qui s'appelle soi-même). Prenons le cas du calcul de `factorielle(2)` : le programme principal appelle `factorielle(2)`, qui appelle `factorielle(1)`, qui appelle `factorielle(0)`, qui peut calculer sa valeur de retour sans autre appel récursif. La valeur retournée par `factorielle(0)` permet à `factorielle(1)` de calculer son résultat, ce qui permet à `factorielle(2)` d'en calculer le sien et de le retourner au programme principal.

L'enchaînement des appels et des calculs est illustré dans la figure 13.3.

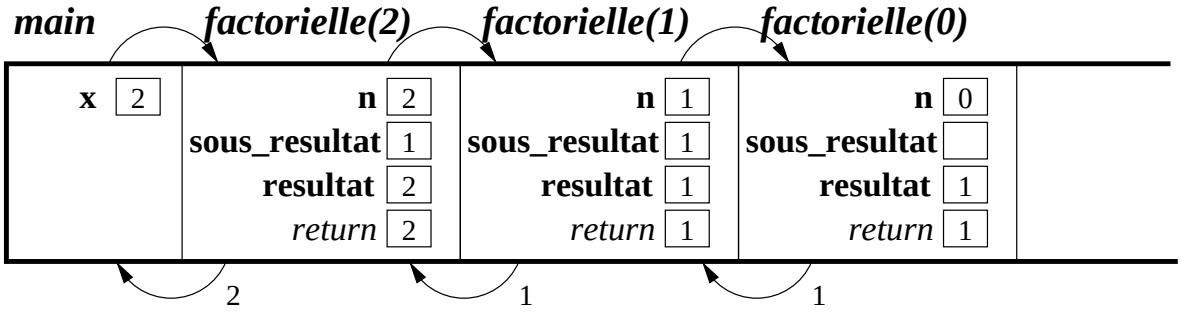

FIG. 13.3 – Enchaînement des appels récursifs pour `factorielle(2)`

Chaque instance de `factorielle` récupère le résultat de l'appel récursif suivant dans la variable *`sous_resultat`*, ce qui lui permet de calculer resultat et de le retourner à son appelant.

13.3 Comment concevoir un sous-programme récursif ?

Dans l'écriture des programmes récursifs on retrouve généralement les étapes suivantes :

1. *Trouver une décomposition récursive du problème*
   (a) Trouver l'élément de récursivité qui permet de définir les cas plus simples (*ex.* une valeur numérique qui décroît, une taille de données qui diminue).
   (b) Exprimer la solution dans le cas général en fonction de la solution pour le cas plus simple.
2. *Trouver la condition d'arrêt de récursivité et la solution dans ce cas*
   – Vérifier que la condition d'arrêt est atteinte après un nombre fini d'appels récursifs dans tous les cas
3. *Réunir les deux étapes précédentes dans un seul programme*

Exemple : calcul du nombre d'occurrences n d'un caractère donné c dans une chaîne de caractères donnée s.

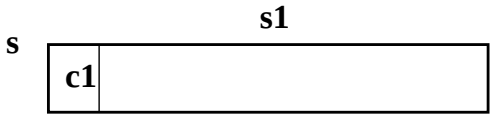

FIG. 13.4 – Décomposition récursive d'une chaîne `s` en un premier caractère `c1` et le reste de la chaîne `s1`

1. Décomposition récursive
   (a) Elément de récursivité : la chaîne, dont la taille diminue. Le cas "plus simple" est la chaîne sans son premier caractère (notons-la $s1$) - voir la figure 13.4.
   (b) Soit $n1$ le nombre d'occurrences de c dans $s1$ (la solution du problème plus simple).
   Soit $c1$ le premier caractère de la chaîne initiale s.
   Si $c1 = c$, alors le résultat est $n = n1 + 1$, sinon $n = n1$.
2. Condition d'arrêt : si s= "" (la chaîne vide) alors n = 0.
   La condition d'arrêt est toujours atteinte, car toute chaîne a une longueur positive et en diminuant de 1 la taille à chaque appel récursif on arrive forcément à la taille 0.
3. La fonction sera donc la suivante :

Listing 13.9 – (lien vers le code brut)

```
1  int nbOccurrences (char c, String s){
2    if(s.length() == 0) return 0;  //condition d'arret: chaine vide
3    else{
4      int sous_resultat = nbOccurrences(c, s.substring(1, s.length()-1));
5      if(s.charAt(0) == c) return 1 + sous_resultat;
6      else return sous_resultat;
7    }
8  }
```

13.3.1 Sous-programmes récursifs qui ne retournent pas de résultat

Les exemples précédents de récursivité sont des *calculs récursifs*, représentés par des fonctions qui retournent le résultat de ce calcul. La décomposition récursive montre comment le résultat du calcul dans le cas "plus simple" sert à obtenir le résultat final.

La relation entre la solution du problème et la solution du cas “plus simple” est donc *une relation de calcul* (entre valeurs).

Toutefois, dans certains cas le problème ne consiste pas en un calcul, mais en une *action récursive* sur les données (affichage, modification de la valeur). Dans ce cas, l'action dans le cas “plus simple” représente une partie de l'action à réaliser dans le cas général. Il s'agit donc d'une *relation de composition entre actions*.

Exemple : affichage des éléments d'un tableau t en ordre inverse à celui du tableau.

Cet exemple permet d'illustrer aussi la méthode typique de décomposer récursivement un tableau Java. La différence entre un tableau et une chaîne de caractères en Java est qu'on ne dispose pas de fonctions d'extraction de sous-tableaux (comme la méthode `substring` pour les chaînes de caractères). On peut programmer soi-même une telle fonction, mais souvent on peut éviter cela, comme expliqué ci-dessous.

La méthode la plus simple est de considérer comme élément de récursivité non pas le tableau, mais *l'indice du dernier élément du tableau*. Ainsi, le cas “plus simple” sera non pas un sous-tableau, mais un indice de dernier élément plus petit - voir la figure 13.5.

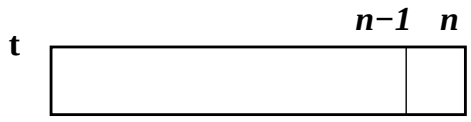

FIG. 13.5 – Décomposition récursive d'un tableau `t` en utlisant l'indice `n` du dernier élément

1. Décomposition récursive
   (a) Elément de récursivité : l'indice n du dernier élément du tableau t.
   Le cas “plus simple” est l'indice du dernier élément n-1, ce qui correspond au tableau t sans son dernier élément.
   (b) L'action pour le tableau t est décomposée de la manière suivante :
   1. Affichage de $t(n)$
   2. Affichage récursif pour t sans le dernier élément.
2. Condition d'arrêt : si n= 0 (un seul élément dans t) alors l'action est d'afficher cet élément ($t(0)$).
   La condition d'arrêt est toujours atteinte, car tout tableau a au moins un élément, donc $n >= 0$.
   En diminuant n de 1 à chaque appel récursif, on arrive forcément à la valeur 0.
3. La fonction sera donc la suivante :

Listing 13.10 – (lien vers le code brut)

```
1 void affichageInverse(int[] t, int n){
2   if(n==0) Terminal.ecrireIntln(t(0)); //condition d'arret: un seul element
3   else{
4     Terminal.ecrireIntln(t(n));
5     affichageInverse(t, n-1);
6   }
7 }
```

13.4 Récursivité et itération

Par l'appel répété d'un même sous-programme, la récursivité permet de réaliser des traitements répétitifs. Jusqu'ici, pour réaliser des répétitions nous avons utilisé *les boucles (itération)*. Suivant le type de problème, la solution s'exprime plus naturellement par récursivité ou par itération. Souvent il est aussi simple d'exprimer les deux types de solution.

Exemple : la solution itérative pour le calcul du nombre d'occurrences d'un caractère dans une chaîne est comparable en termes de simplicité avec la solution récursive.

Listing 13.11 – (lien vers le code brut)

```
1 int nbOccurrences (char c, String s){
2   int accu = 0;
3   for(int i=0; i<s.length(); i++)
4     if(s.charAt(i)==c) accu++;
5   return accu;
6 }
```

En principe tout programme récursif peut être écrit à l'aide de boucles (par itération), sans récursivité. Inversement, chaque type de boucle (while, do...while, for) peut être simulé par récursivité. Il s'agit en fait de deux manières de programmer différentes. Utiliser l'une ou l'autre est souvent une question de goût et de style de programmation - cependant chacune peut s'avérer mieux adaptée que l'autre à certaines classes de problèmes.

13.4.1 Utilisation de l'itération

Un avantage important de l'itération est *l'efficacité*. Un programme qui utilise des boucles décrit précisément chaque action à réaliser - ce style de programmation est appelé *impératif* ou *procédural*. Le code compilé d'un tel programme est une image assez fidèle des actions décrites par le programme, traduites en code machine. Les choses sont différentes pour un programme récursif.

Conclusion : si on recherche l'efficacité (une exécution rapide) et le programme peut être écrit sans trop de difficultés en style itératif, on préférera l'itération.

13.4.2 Utilisation de la récursivité

L'avantage de la récursivité est qu'elle se situe à un *niveau d'abstraction supérieur* par rapport à l'itération. Une solution récursive décrit comment calculer la solution à partir d'un cas plus simple - ce style de programmation est appelé *déclaratif*. Au lieu de préciser chaque action à réaliser, on décrit ce qu'on veut obtenir - c'est ensuite au système de réaliser les actions nécessaires pour obtenir le résultat demandé.

Souvent la solution récursive d'un problème est *plus intuitive* que celle itérative et le programme à écrire est *plus court* et *plus lisible*. Néanmoins, quelqu'un habitué aux boucles et au style impératif peut avoir des difficultés à utiliser la récursivité, car elle correspond à un type de raisonnement particulier.

Le style déclaratif produit souvent du code moins efficace, car entre le programme descriptif et le code compilé il y a un espace d'interprétation rempli par le système, difficile à optimiser dans tous les cas. Cependant, les langages déclaratifs offrent un confort nettement supérieur au programmeur (comparez SQL avec un langage de bases de données avec accès enregistrement par enregistrement).

Dans le cas précis de la récursivité, celle-ci est moins efficace que l'itération, car la répétition est réalisée par des appels successifs de fonctions, ce qui est plus lent que l'incrémentation d'un compteur de boucle.

Dans les langages (comme Java) qui offrent à la fois l'itération et la récursivité, on va préférer la récursivité surtout dans les situations où la solution itérative est difficile à obtenir, par exemple :

- si les structures de données manipulées sont récursives (*ex*. les arbres).
- si le raisonnement lui même est récursif.

13.4.3 Exemple de raisonnement récursif : les tours de Hanoi

Un exemple très connu de raisonnement récursif apparaît dans le problème des tours de Hanoi. Il s'agit de n disques de tailles différentes, trouées au centre, qui peuvent être empilés sur trois piliers. Au début, tous les disques sont empilés sur le pilier de gauche, en ordre croissant de la taille, comme dans la figure suivante. Le but du jeu est de déplacer les disques un par un pour reconstituer la tour initiale sur le pilier de droite. Il y a deux règles :

1. on peut seulement déplacer un disque qui se trouve au sommet d'une pile (non couvert par un autre disque) ;
2. un disque ne doit jamais être placé sur un autre plus petit.

Le jeu est illustré dans la figure 13.6.

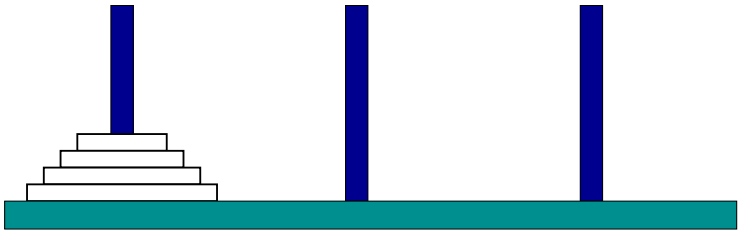

FIG. 13.6 – Les tours de Hanoi

La solution s'exprime très facilement par récursivité. On veut déplacer une tour de n disques du pilier de gauche vers le pilier de droite, en utilisant le pilier du milieu comme position intermédiaire.

1. Il faut d'abord déplacer vers le pilier du milieu la tour de n-1 disques du dessus du pilier de gauche (en utilisant le pilier de droite comme intermédiaire).
2. Il reste sur le pilier de gauche seul le grand disque à la base. On déplace ce disque sur le pilier de droite.
3. Enfin on déplace la tour de n-1 disques du pilier du milieu vers le pilier de droite, au-dessus du grand disque déjà placé (en utilisant le pilier de gauche comme intermédiaire).

La procédure récursive `deplaceTour` réalise cet algorithme et utilise la procédure `deplaceUnDisque` pour afficher le déplacement de chaque disque individuel.

Remarque : Il est difficile d'écrire un algorithme itératif pour ce problème. Essayez comme exercice d'écrire le même programme avec des boucles !

Listing 13.12 – (lien vers le code brut)

```
1 public class Hanoi{
2   static void deplaceUnDisque (String source, String dest){
```

```
3       Terminal.ecrireStringln(source + " - " + dest);
4     }
5
6     static void deplaceTour(int taille, String source, String dest, String interm){
7       if(taille==1) deplaceUnDisque(source, dest);
8       else{
9         deplaceTour(taille-1, source, interm, dest);
10        deplaceUnDisque(source, dest);
11        deplaceTour(taille-1, interm, dest, source);
12      }
13    }
14
15    public static void main(String[] args){
16      Terminal.ecrireString("Combien de disques ? ");
17      int n = Terminal.lireInt();
18      deplaceTour(n, "gauche", "droite", "milieu");
19    }
20  }
```

L'exécution de ce programme produit le résultat suivant :

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