Introduction à la Programmation 1

Séance 9 de cours/TD

Université Paris-Diderot

Objectifs:

- Comprendre ce qu'est la factorisation de code.
- Décrire précisément le comportement des fonctions.
- Spécifier le comportement d'une fonction.
- Réaliser une spécification.

1 Factoriser du code

Factorisation de code [Cours]

Dans la communication, écrite en particulier, la concision est une vertu qui va souvent de paire avec la clarté et l'élégance. Par exemple, on préfère écrire :

$7 \times (x + y + z) \times w$

plutôt que :

$w \times x \times 7 + w \times 7 \times y + 7 \times z \times w$.

La deuxième expression est équivalente à la première, mais plus difficile à comprendre.

Dans la pratique de la programmation, produire du code concis, clair, facilement réutilisable et documenté est très important. En effet, la programmation est certes un exercice de communication avec la machine mais c'est aussi un exercice de communication avec l'humain qui essaiera de comprendre votre programme (pour l'améliorer, pour corriger une erreur ou encore pour le réutiliser dans un autre contexte par exemple).

Il arrive souvent que l'on écrive un premier programme qui fonctionne mais n'est pas très bien écrit. On cherche alors le *réécrire en un programme équivalent* mais plus concis. Cela s'appelle **factoriser** du code.

Il existe bien sûr une infinité de façon de factoriser du code. S'il n'y a pas de recette universelle pour écrire du code concis et élégant, il existe des transformations récurrentes comme par exemple :

- **Déplacer** des instructions pour éviter de les répéter plusieurs fois.
- **Réécrire** des conditions pour regrouper des cas.
- **Factoriser** plusieurs instructions répétées à l'aide une boucle.
- **Découper** une longue série d'instructions en plusieurs fonctions ou procédures.
- **Généraliser** plusieurs blocs d'instructions similaires en introduisant des fonctions ou des procédures.

Exemple : un code très verbeux [Cours]

Voici un programme verbeux formé d'une série d'instructions conditionnelles difficiles à comprendre :

```
]if (dir == 1) {
    x = x - 1;
    y = y + 1;
```

```
4       drawPosition (x, y);
5   }
6   else if (dir == 2) {
7       y = y + 1;
8       drawPosition (x, y);
9   }
10  else if (dir == 3) {
11      x = x + 1;
12      y = y + 1;
13      drawPosition (x, y);
14  }
15  else if (dir == 4) {
16      x = x - 1;
17      drawPosition (x, y);
18  }
19  else if (dir == 5) {
20      x = x + 1;
21      drawPosition (x, y);
22  }
23  else if (dir == 6) {
24      x = x - 1;
25      y = y - 1;
26      drawPosition (x, y);
27  }
28  else if (dir == 7) {
29      y = y - 1;
30      drawPosition (x, y);
31  }
32  else if (dir == 8) {
33      x = x + 1;
34      y = y - 1;
35      drawPosition (x, y);
36  }
```

Listing 1 – Exemple de code verbeux

Dans cet exemple, dir est une variable de type int qui représente une direction dans le plan :

1	2	3
4	X	5
6	7	8

L'objectif de ce fragment de programme est d'afficher un point à la position (x, y) une fois qu'il a été déplacé dans la direction dir.

Ce programme a plusieurs défauts :

1. C'est une longue série de tests : comment se convaincre qu'aucun test n'a été oublié ?
2. Les instructions de chaque branche se ressemblent sans être exactement les mêmes : comment se convaincre que ces différences sont bien justifiées ?
3. Le programme fait trois choses en même temps : (i) il calcule un déplacement en x et y en fonction de la valeur de dir ; (ii) il met à jour la valeur de x et y ; (iii) il affiche le point à la nouvelle position. Pour être plus compréhensible, ce programme devrait expliciter ces trois étapes.

Déplacement d'instructions [Cours]

Dans le programme précédent, l'appel de fonction drawPosition (x, y) apparaît huit fois en dernière position de la séquence d'instructions de chaque branche. On peut sans risque *déplacer cette instruction après la série des tests pour n'avoir à l'écrire qu'une fois*.

```
if (dir == 1) {
    x = x - 1;
    y = y + 1;
} else if (dir == 2) {
    y = y + 1;
} else if (dir == 3) {
    x = x + 1;
    y = y + 1;
} else if (dir == 4) {
    x = x - 1;
} else if (dir == 5) {
    x = x + 1;
} else if (dir == 6) {
    x = x - 1;
    y = y - 1;
} else if (dir == 7) {
    y = y - 1;
} else if (dir == 8) {
    x = x + 1;
    y = y - 1;
}
drawPosition (x, y);
```

Listing 2 – Après déplacement d'une instruction.

Explicitation de variables [Cours]

On peut faire apparaître les trois étapes du programme en introduisant des variables dx et dy qui représentent le déplacement de x et y en fonction de la valeur de dir.

```
int dx = 0;
int dy = 0;
if (dir == 1) {
    dx = - 1;
    dy = 1;
} else if (dir == 2) {
    dy = 1;
} else if (dir == 3) {
    dx = 1;
    dy = 1;
} else if (dir == 4) {
    dx = -1;
} else if (dir == 5) {
    dx = 1;
} else if (dir == 6) {
    dx = -1;
    dy = -1;
} else if (dir == 7) {
    dy = -1;
} else if (dir == 8) {
    dx = 1;
```

```
22      dy = -1;
23  }
24  x = x + dx;
25  y = y + dy;
26  drawPosition (x, y);
```

Listing 3 – Après introduction de deux variables.

Regroupement de tests [Cours]

Dans beaucoup de cas, les calculs de dx et de dy sont les mêmes. On peut restructurer les tests comme suit :

```
1   int dx = 0;
2   int dy = 0;
3   if (dir == 1 || dir == 2 || dir == 3) {
4       dy = 1;
5   }
6   if (dir == 6 || dir == 7 || dir == 8) {
7       dy = -1;
8   }
9   if (dir == 3 || dir == 5 || dir == 8) {
10      dx = 1;
11  }
12  if (dir == 1 || dir == 4 || dir == 6) {
13      dx = -1;
14  }
15  x = x + dx;
16  y = y + dy;
17  drawPosition (x, y);
```

Listing 4 – Après regroupement des tests.

Introduction de fonctions [Cours]

Le programme précédent est concis mais on peut encore améliorer sa lisibilité en introduisant du *vocabulaire* permettant de le décrire en des termes plus abstraits qu'avec les instructions basiques de Java. Pour introduire du vocabulaire, on peut se donner des fonctions aux noms bien choisis.

```
1   public static int dyFromDirection (int dir) {
2     if (dir == 1 || dir == 2 || dir == 3) {
3         return 1;
4     }
5     if (dir == 6 || dir == 7 || dir == 8) {
6         return -1;
7     return 0;
8   }
9
10  public static int dxFromDirection (int dir) {
11    if (dir == 3 || dir == 5 || dir == 8) {
12        return 1;
13    }
14    if (dir == 1 || dir == 4 || dir == 6) {
15        return -1;
16    }
17    return 0;
18  }
19
```

```
20 public static void moveTowards (int dir) {
21   x = x + dxFromDirection (dir);
22   y = y + dyFromDirection (dir);
23 }
```

Listing 5 – Des fonctions, du nouveau vocabulaire.

Dès lors le code final se réduit à deux lignes, aisément compréhensibles :

```
1 moveTowards (dir);
2 drawPosition (x, y);
```

Listing 6 – Le programme final.

Exercice 1 (Bien choisir les tests, ⋆⋆)

Réecrire le fragment de code suivant, de manière concise :

```
1  if (x < 0) {
2      y = y + 1;
3      z = z - 1;
4  } else {
5      if (1 <= x && x < 5) {
6          y = y - 1;
7          z = z - 1;
8      } else {
9          if (6 <= x && x < 10) {
10             y = y + 1;
11             z = z - 1;
12         } else {
13             y = y - 1;
14             z = z - 1;
15         }
16     }
17 }
```

□

Factorisation à l'aide d'une boucle [Cours]

Il est toujours préférable de remplacer par une boucle une série d'instructions répétées un grand nombre de fois. Cela peut augmenter le nombre de lignes de code mais la boucle obtenue est plus compréhensible. Par ailleurs, lorsque le programmeur doit taper une longue série d'instructions très similaires, il est plus probable qu'il fasse une erreur que lorsqu'il écrit une seule boucle.

Ainsi, prendre la moyenne d'un tableau d'entiers array peut s'écrire ainsi :

```
1 int mean = (array[0] + array[1] + array[2] + array[3] + array[4]
2           + array[5] + array[6] + array[7] + array[8] + array[9]) / 10;
```

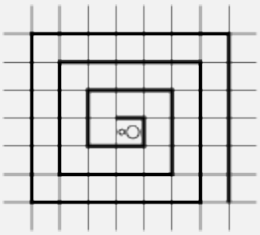
Listing 7 – Calcul de moyenne verbeux

mais on préférera l'usage d'une boucle :

```
1 int mean = 0;
2 for (int i = 0; i < 10; i++) {
3     mean = mean + array[i];
4 }
5 mean = mean / 10;
```

Listing 8 – Calcul de moyenne concis

Parfois le caractère répétitif de la tâche à accomplir est évident, mais bien paramétrer les boucles censées factoriser le code ne l'est pas. Considérez le labyrinthe :



```
west();
north();
east();
east();
south();
south();
west();
west();
west();
north();
north();
north();
east();
east();
east();
east();
south();
south();
south();
south();
west();
west();
west();
west();
west();
north();
north();
north();
north();
north();
east();
east();
east();
east();
east();
east();
south();
south();
south();
south();
south();
south();
```

Listing 9 – Sortie compliquée du labyrinthe.

En utilisant des boucles :

```
for (int i = 1; i < 4; i++) {
    for (int j = 1 ; j <= (2 * i - 1); j++) {
```

```
        west();
    }
    for (int j = 1; j <= (2 * i - 1); j++) {
        north();
    }
    for (int j = 1; j <= (2 * i); j++) {
        east();
    }
    for (int j = 1; j <= (2 * i); j++) {
        south();
    }
}
```

Listing 10 – Sortie claire du labyrinthe.

[Cours]

Souvent, quand on passe du code verbeux au code concis, on cerne mieux la nature du problème à résoudre : le code verbeux focalise sur *une instance* du problème, le code concis sur son *essence*. À partir de là, on peut définir un outil qui résout toutes les instances du problème en question, c'est à dire une fonction (ou une procédure), qui prend comme les paramètres décrivant l'instance du problème et renvoie le résultat escompté (ou exécute la tâche escomptée). Dans l'exemple de la moyenne d'un tableau d'entiers, cela donne :

```
public static int average (int[] a) {
    int acc = 0;
    int l = intArrayLength (a);
    for (int i = 0; int < l; i++) {
        acc = acc + a[i];
    }
    return (acc / l);
}
```

Listing 11 – Calcul de la moyenne par une fonction

ensuite, quand on veut affecter à la variable entiere `mean` la (partie entière de la) moyenne du tableau d'entiers `array`, on appelle la fonction comme suit :

```
mean = average (array);
```

Dans le cas du labyrinthe :

```
public static void exitLabySpiral (int size) {
    for (int i = 1; i <= size / 2; i++) {
        for (int j = 1; j <= 2 * i - 1; j++)
            west ();
        for (int j = 1; j <= 2 * i - 1; j++)
            north ();
        for (int j = 1; j <= 2 * i; j++)
            east ();
        for (int j = 1; j <= 2 * i; j++)
            south ();
    }
}
```

Listing 12 – Sortie du labyrinthe par une procédure

ensuite, on peut utiliser cette procédure pour sortir d'un labyrinthe à spirale de `n` lignes, en l'appelant comme suit :

```
exitLabySpiral (n);
```

Dans le cas de la sortie du labyrinthe à spirale, il est possible de factoriser ultérieurement le code en définissant une procédure qui permet à la fourmi d'avancer d'un nombre de cases donné dans une direction donnée. Le type du premier paramètre, le nombre de cases, est `int`. Pour le deuxième, la direction, on pourrait aussi choisir `int`, à condition de choisir un code entier pour chaque point cardinal. Ce deuxième choix donne le code suivant :

```
/*
   Procédure qui fait avancer la fourmi d'un nombre de cases p
   dans la direction d. La direction est codée comme suit
            nord = 1, est = 2, sud = 3, ouest = 4
*/
public static void goForward (int p, int d) {
  for (int i = 1; i <= p; i++) {
    if (d == 1)
      north ();
    else if (d == 2)
      east ();
    else if (d == 3)
      south ();
    else if (d == 4)
      west ()
  }
}
```

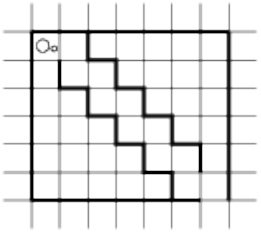
on peut à présent utiliser `goForward` pour écrire une deuxième version de `exitLabySpyral` :

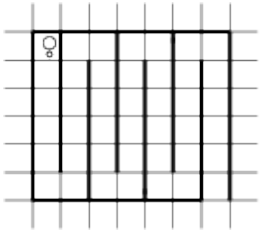
```
public static void exitLabySpiral2 (int size) {
    for (int i = 1; i <= size / 2; i++) {
      goForward (2 * i - 1, 4);
      goForward (2 * i - 1, 1);
      goForward (2 * i, 2);
      goForward (2 * i, 3);
    }
}
```

Exercice 2 (Labyrinthes, ⋆⋆)

On considère le problème de sortir des labyrinthes ci-dessous. (On dispose des procédures `north()`, `south()`, `east()`, `west()` *pour déplacer la fourmi.) Donnez pour chacun :*

1. *le code verbeux,*
2. *le code concis en utilisant une boucle,*
3. *une fonction qui prend en paramètre la taille du labyrinthe et résout le problème.*





□

2 Approfondissements sur les fonctions

2.1 État de la mémoire avant et après exécution d'un appel de fonction.

[Cours]

Un appel de fonction sert à calculer une valeur de retour, à partir des valeurs d'un certain nombre de paramètres. Ce calcul peut être complexe, et peut nécessiter l'usage d'un certain nombre de *variables locales* à la fonction. Après le renvoi du résultat, c'est à dire après l'exécution de l'instruction `return`, les variables locales de la fonction, tout comme ses paramètres, sont effacés de la mémoire. Par exemple :

```
public static int f (int a, int b){
    int c = 0;
    /* a, b, c sont visibles ici, les variables de main ne le sont pas. */
    ...
    return ...;
}

public static void main(String[] vars){
    ...
    int d = f (3, 7);
    /* Les parametres a et b et la variable locale c de f
       ne sont plus visibles ici. */
    ...
}
```

De plus, il n'y a aucune interférence entre paramètres et variables locales d'une fonction d'un côté, et variables visibles après le retour d'un appel de cette fonction de l'autre. Par exemple :

```
public static int f (int a, int b) {
    int c = 0;
    ...
    return ...;
}

public static void main (String[] vars) {
    ...
    int a = 1;
    int c = 1;
    int d = f (3, 7);
    /*
       Les variables a et c portent le même nom qu'un parametre et
       une variable locale de f, mais désignent d'autres emplacements
       en mémoire.

       Elles valent 1 avant l'appel, et conservent donc leur valeur.
    */
    ...
}
```

Il est possible de résumer ce qu'on vient de voir en un slogan : *Les variables locales et les paramètres d'une fonction ne sont visibles que dans celle-ci, et ne sont allouées que pendant l'exécution de celle-ci.*

Néanmoins, l'exécution d'un appel de fonction peut avoir des effets autres que le simple renvoi de la valeur de retour. Ceci ne constitue pas une exception à la règle de localité, énoncée dans le slogan ci-dessus, mais dépend du type du paramètre.

Les exemples suivants illustrent le passage d'une valeur de type `int` et de type `int[]` respectivement :

```java
public static int f (int a) {
    a = a + 1;
    return a;
}

public static void main (String[] vars) {
    ...
    int b = 0;
    int c = f (b);
    /* Ici, c vaut 1 et b vaut 0. */
    ...
}
```

Cet exemple montre que lorsqu'on passe une variable entière en argument à une fonction, c'est la *valeur* de cette variable qui est copiée dans le paramètre. Les modifications du paramètre dans la fonction n'affectent pas la variable b.

```java
public static int f (int[] a) {
    a[0] = a[0] + 1;
    return a[0];
}

public static void main (String[] vars) {
    ...
    int[] b = {0, 0};
    int c = f (b);
    /* Ici, c vaut 1 et b[0] vaut 1. */
    ...
}
```

Cet exemple montre que lorsqu'on passe un tableau à une fonction, c'est la *référence* à ce tableau dans le tas qui est copiée dans le paramètre. En effet, la *valeur* contenu dans une valeur de type tableau est une référence. C'est donc cette valeur qui est copiée et non pas l'ensemble de toutes les valeurs contenues dans le tableau.

Les modification éventuelles des paramètres de type tableau pendant l'exécution du corps d'une fonction sont donc pérennes puisqu'elles travaillent sur le même tableau (*via* la même référence) que celui qui est accessible par l'appelant de la fonction.

2.2 Documenter et commenter son code

[Cours]

Un programme peu ou mal documenté est difficile à corriger, modifier et réutiliser. Il est donc important de documenter et de commenter systématiquement son code. Pour chaque fonction, avant l'en-tête, on place quelques lignes de commentaires, au format suivant :

- une ligne pour chaque paramètre d'entrée, indiquant son type, ce que ce paramètre représente et une propriété attendue sur l'entrée, que l'on appelle **précondition** ;
- une ligne pour la valeur renvoyée, son type, ce qu'elle représente et la propriété promise sur cette sortie, que l'on appelle **postcondition**.

Pour les procédures, on indiquera **l'effet qui sera observé** après l'exécution de la procédure à la place de la **postcondition**, qui n'a pas lieu d'être puisqu'aucune valeur n'est renvoyée par une procédure.

Des commentaires supplémentaires liés à l'implémentation de la fonction seront placés dans son corps, comme dans l'exemple suivant :

```
/*
   Vérification de la correction d'une date.

   Entrée : d un entier désignant le jour.
   Entrée : m un entier désignant le mois.
   Entrée : y un entier désignant l'année.
   Sortie : true si la date est correcte, false sinon.
*/
public static boolean checkDate (int d, int m, int y) {
    if (m >= 1 && m <= 12 && d >= 1 && d <= 31) {
        if (m == 4 || m == 6 || m == 9 || m == 11) {
            return (d <= 30);
        } else {
            if (m == 2) { /* Est-ce une année bissextile? */
                if ((y%4 == 0 && y%100 !=0 ) || y%400 == 0) {
                    return (d <= 29);
                } else {
                    return (d <= 28);
                }
            } else {
                return true;
            }
        }
    } else {
        return false;
    }
}
```

La spécification des fonctions ainsi fournie doit permettre de les utiliser sans avoir à connaître leur code.

Exercice 3 (Réaliser des spécifications, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
/*
   Somme des entiers d'un intervalle.

   Entrée: low un entier designant le début de l'intervalle.
   Entrée: high un entier designant la fin de l'intervalle.
   Sortie: la somme des entiers compris au sens large entre low et high.
*/
```

□

Exercice 4 (Utiliser des spécifications, ⋆⋆)

La conjecture de Goldbach *dit que tout nombre pair plus grand que 2 peut être écrit comme la somme de deux nombres premiers.*

1. *Écrire une fonction* boolean isPrime(int p) *qui implémente la spécification suivante :*

```
/*
    Test si un entier est premier.

```

```
    Entrée : un entier p plus grand que 1.
    Sortie : true si et seulement si n est premier, false sinon.
*/
```

2. *Écrire la spécification de la fonction* boolean isGoldbach(int n) *qui renvoie* true *si et seulement si la conjecture est vérifiée pour l'entier* n. *(Si* n *n'est pas pair ou est strictement plus petit que 2 la fonction renvoie* true*.)*
3. *Donner le code de la fonction* boolean isGoldbach(int n)*. (On utilisera la fonction précédemment définie.)*
4. *Écrire une fonction* boolean goldbach(int p) *qui implémente la spécification suivante. (On utilisera la fonction précédemment définie.) :*

```
/*
   Vérifie la conjecture de Goldbach jusqu'à un certain rang.

   Entrée : un entier n.
   Sortie : true si et seulement si tout entier inférieur ou égal à p
            vérifie la conjecture de Goldbach.
*/
```

□

3 Fonctions utilisées

```
/*
 * Calcule le nombre de cases d'un tableau d'entiers
 * t est le tableau dont on veut connaitre la taille
 * Retourne la taille du tableau
 */
public static int intArrayLength(int [] t) {
   return t.length;
}
```

4 DIY

Dans les exercices suivants, vous vous appliquerez à bien factoriser votre code.

Exercice 5 (Inclusion de tableaux, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
/*
    Inclusion d'un tableau dans un autre.

    Entrée : un tableau d'entiers a
    Entrée : un tableau d'entiers b
    Sortie : true si et seulement si tous les éléments du tableau a sont
             contenus dans le tableau b en prenant en compte les
             répétitions.
*/
```

□

Exercice 6 (Séquences dans un tableau, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
/*
    Entrée : un tableau d'entiers a.
    Sortie : true si et seulement si a contient deux
             séquences consécutives identiques. Par exemple, sur
             {1,2,3,2,3,7} la valeur à renvoyer est true.
*/
```

□

Exercice 7 (Chaîne correspondante a un tableau, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
/*
    Entrée : un tableau de tableaux d'entiers t.
    Sortie : une chaine de caractère le représentant.

    Par exemple, si t vaut {{1,2},{3},{2,3,7}}, la chaine de caractère sera
    "{{1,2},{3},{2,3,7}}".
*/
```

□

Exercice 8 (Tri d'un tableau, ⋆⋆)

1. *Écrire une procédure* swap *qui implémente la spécification suivante :*

```
/*
    Echange le contenu de deux cases d'un tableau.

    Entrée : un tableau d'entiers t.
    Entrée : un entier i compris dans les bornes du tableau.
    Entrée : un entier j compris dans les bornes du tableau.

    Après avoir exécuté cette procédure les contenus des cases i et j
    du tableau t sont échangés.
*/
```

2. *En utilisant la procédure précédente, écrire une procédure* findmin *de spécification suivante :*

```
/*
    Echange le contenu de la case i avec la case j du tableau t qui
    contient le plus petit entier des cases d'indice compris entre
    i et l'indice de fin du tableau.

    Entrée : un tableau d'entiers t.
    Entrée : un entier i compris dans les bornes du tableau.

*/
```

3. *En utilisant la procédure précédente, écrire une procédure* sort *qui implémente la spécification suivante :*

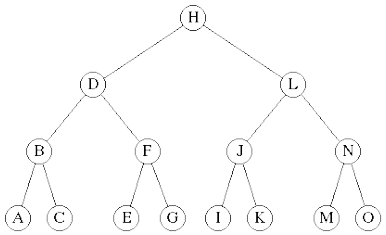
```
/*
    Tri le tableau.

```

```
    Entrée : un tableau d'entiers t.
*/
```

□

Exercice 9 (Tournoi, ⋆⋆⋆)

On cherche à représenter les différents matchs d'un tournoi dans un tableau. Un tournoi peut se représenter dans un arbre binaire. (Les nœuds de l'arbre contiennent toujours deux fils.)



Pour simplifier le problème, le nombre de participant est supposé être une puissance de 2. On suppose aussi que chaque partipant est représenté par un entier unique. Cet entier est le niveau du participant, si il joue contre un autre participant dont l'entier est plus petit, alors il gagne sinon il perd.

Un arbre est représenté par un tableau en décidant que :

- *la racine de l'arbre est à la position 0 ;*
- *le fils gauche d'un nœud situé dans la case i est situé dans la case $2 * i + 1$;*
- *le fils droit d'un nœud situé dans la case i est situé dans la case $2 * i + 2$.*

Ainsi, l'arbre précédent peut être représenté par le tableau de taille 15 suivant :

```
{ H, D, L, B, F, J, N, A, C, E, G, I, K, M, O }
```

Pour représenter un tournoi avec 2^n participants, il faut $2^{n+1} - 1$ noeuds dans l'arbre. (Montrez-le par récurrence pour vous en convaincre.) Il faut donc créer un tableau de taille $2^{n+1} - 1$ et on l'initialise avec la valeur -1 pour indiquer qu'aucun match n'a encore eu lieu.

Pour commencer le tournoi, on place les participants sur les feuilles de l'arbre. Tant qu'il ne reste pas qu'un seul participant, on fait un tour supplémentaire de jeu. Un tour de jeu consiste à faire jouer tous les matchs du niveau le plus bas de l'arbre dont les matchs ne sont pas encore joués. On fait alors "remonter" le gagnant de chaque match joué pour ce tour au niveau suivant, ce qui a pour effet de préparer le tour suivant, joué au niveau du dessus dans l'arbre.

1. *Après avoir spécifié et implémenté des fonctions utiles pour :*
   - *construire l'arbre nécessaire à la représentation d'un tournoi entre 2^n participants représentés par un tableau ;*
   - *représenter l'indice du père d'un nœud dans l'arbre ;*
   - *faire jouer deux participants en faisant remonter le gagnant dans l'arbre ;*

   Écrivez une fonction tournament *de spécification suivante :*

```
/*

    Entrée : un entier n.
    Entrée : un tableau d'entiers t de taille 2 puissance n.
    Sortie : un tableau représentant le tournoi des participants de t.

*/
```

2. *Comment généraliser la fonction précédente à un nombre de participants qui n'est pas une puissance de 2 ?*

3. *Après avoir spécifié et implémenté une fonction qui calcule l'ensemble des participants qui ont perdu contre le vainqueur d'un tournoi, écrivez une fonction* kbest *de spécification suivante :*

```
/*
    Entrée : un tableau d'entiers t.
    Entrée : un entier k compris entre 1 et la taille du tableau.
    Sortie : un tableau représentant les k meilleurs participants de t.
*/
```

□