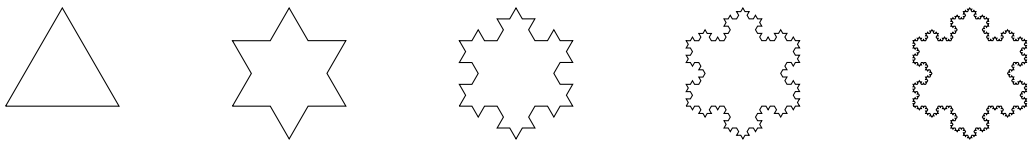
1 La récursivité

1.1 Généralités

En informatique, la **récursivité** est une approche algorithmique qui consiste à faire référence au problème ou à l'objet étudié à un moment du processus. En d'autres termes, c'est une démarche dont la description mène à la répétition d'une même règle. Les cas suivants constituent des cas concrets de récursivité :

1. décrire un processus dépendant de données en faisant appel à ce même processus sur d'autres données plus « simples » : $\texttt{PGCD}(1320, 210) = \texttt{PGCD}(210, 60) = \texttt{PGCD}(60, 30) = \texttt{PGCD}(30, 0) = 30$.
2. construire/décrire une image fractale (le flocon de Von Koch par exemple) :

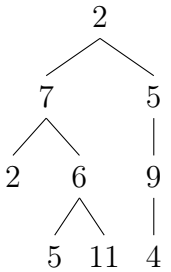


3. écrire un algorithme qui s'invoque lui-même ;

Algorithme 1 : `compte_a_rebours(n : entier naturel)`

```
1 Procedure compte_a_rebours(n : entier naturel) :
2 début
3     si n = 0 alors
4         Afficher(0)
5     sinon
6         Afficher(n)
7         compte_a_rebours(n-1)
```

4. définir une structure de données à partir de l'une au moins de ses sous-structures ; « un *arbre binaire* est soit vide, soit un nœud portant deux *arbres binaires* plus petits » :



Un arbre binaire est une structure de données récursive. L'*arbre* représenté ci-contre, contenant les neuf éléments $\{2, 7, 2, 6, 5, 11, 5, 9, 4\}$, est constitué d'un nœud 2 portant à gauche un *arbre* à cinq éléments $\{7, 2, 6, 5, 11\}$ et à droite un *arbre* à trois éléments $\{5, 9, 4\}$. Ce dernier est constitué d'un nœud 5 portant à gauche un *arbre* vide, et à droite un *arbre* à deux éléments $\{9, 4\}$. Ce dernier est constitué d'un nœud 9 portant à gauche un *arbre* à un élément $\{4\}$, et à droite un *arbre* vide. L'*arbre* à un élément $\{4\}$ est constitué d'un nœud 4 portant deux *arbres* vides.

Une fonction (informatique) renvoyant une valeur, il est possible de créer une fonction qui s'appelle elle-même en passant en paramètre le résultat d'un traitement effectué par (une instance d'elle-même. Ce second appel pourra lui-même appeler la fonction une troisième fois, et ainsi de suite. Nous obtenons ainsi

un empilement d'appels, chacun réalisant une étape du traitement à effectuer. Lorsque le processus arrive au bout du traitement, la dernière fonction appelée (fonction *fille*) retourne une valeur qui se propagera jusqu'à la première fonction appelée (fonction *mère*) par dépilement des appels. C'est ainsi qu'une fonction récursive se termine. Il est important de retenir deux points caractérisant la récursivité :

1. la pile mémoire est abondamment utilisée par la récursivité : la plupart des erreurs de programmation récursive génèrent un dépassement de pile (**stack overflow** en anglais) ;
2. une fonction récursive doit impérativement posséder une condition de fin qui provoquera le dépilement.

Le calcul de la factorielle

En mathématiques, la **factorielle** d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. Elle est notée $n!$ et est définie comme suit : pour tout $n \in \mathbb{N}$,

$$n! = \begin{cases} 1 & \text{si } n \in \{0,1\}, \\ n \times (n-1) \times \ldots 3 \times 2 & \text{sinon.} \end{cases} \tag{1.1}$$

De cette définition, nous déduisons « naturellement » un premier algorithme et un premier programme `C` pour calculer la factorielle d'un entier naturel :

Algorithme 2 : `factoriel d'un entier naturel (version itérative)`

```
1 Fonction f_I(n : entier naturel) : entier naturel
2 début
      Données : p, i : entiers naturels
3     p ← 1
4     pour i allant de 2 à n faire
5         p ← p × i
6     retourner p
```

```c
unsigned int f_I(unsigned int n){
  unsigned int p,i ;
  p = 1;
  for(i = 0 ; i <= p ; i = i +1){
    p = p*i;
  }
  return p;
}
```

La fonction mathématique `factorielle` peut également être définie à partir d'une suite récurrente : soit $(f_R(n))_{n\in\mathbb{N}}$ la suite d'entiers naturels telle que $f_R(n) = n!$. Alors, $f_R(0) = f_R(1) = 1$ et $f_R(n) = n \times f_R(n-1)$ pour tout $n \in \mathbb{N}^\star$.

1. Le **profil de la fonction** `f_R` est alors `f_R : entier_naturel` $\longrightarrow$ `entier_naturel`
2. Deux exemples de la trace d'un calcul sont :
   [1] `f_R(0)` $= 1$,
   [2] `f_R(3)` $= 3 \times$ `f_R(2)` $= 3 \times 2 \times$ `f_R(1)` $= 3 \times 2 \times 1 \times$ `f_R(0)` $= 3 \times 2 \times 1 \times 1 = 6$
3. Pus généralement, la fonction `f` satisfait les deux **axiomes** suivants :
   [1] `f_R(0)` $= 1$

[2] `f_R(n)` = `n` × `f_R(n − 1)` si $n > 0$

4. À partir des axiomes précédents, nous déduisons l'**algorithme récursif** suivant :

Algorithme 3 : `factoriel d'un entier naturel`

```
1 Fonction f_R(n : entier naturel) : entier naturel
2 début
3     si n = 0 alors
4         retourner 1
5     sinon
6         retourner n × f_R(n-1)
```

5. Enfin, nous traduisons cet algorithme en un programme `C` :

```c
unsigned int f_R(unsigned int n){
  if (n == 0).   // <---- condition de base
    return 1;
  return n*f_R(n-1); // <---- appel récursif
}
```

La programmation récursive permet d'écrire des programmes généralement plus courts et plus simples que dans leur version itérative. Par exemple, pour le calcul de la factorielle d'un entier naturel, la version récursive ne fait pas appel explicitement à une boucle `pour` ou `tant que` ni à une variable locale contrairement à la version itérative.

Démarche algorithmique

Dans la suite de ce cours, sauf mention explicite du contraire, nous définirons les opérations/algorithmes/programmes selon la démarche algorithmique suivante :

[1] Profil de l'opération : `<nom_de_l_opération>` : `<type_1>`×`<type_2>`× ... ×`<type_n>` ⟶ `<type>`

[2] Quelques exemples détaillés ;

[3] Ensemble d'axiomes à partir des exemples ;

[4] Traduction de l'ensemble des axiomes en un algorithme récursif ;

[5] Écriture d'un algorithme itératif.

[6] Traduction de l'algorithme récursif en un programme `C`.

[7] Traduction de l'algorithme itératif en un programme `C`.

[8] Écriture des procédures de test.

1.2 Exercices

1.2.1 Sur les entiers

Exercice 1

On considère les opérations suivantes :

1. `puissance` est l'opération qui à deux entiers naturels x et n associe x^n.
2. `somme_de_0_a_n` est l'opération qui à un entier naturel n associe la somme des entiers naturels i inférieurs ou égaux à n.
3. `binaire` est l'opération qui pour un entier naturel n affiche l'écriture de n en binaire.
4. `somme_des_chiffres` est l'opération qui à deux entiers naturels n et $b > 1$ associe la somme des chiffres de l'écriture de n en base b.
5. `nombre_de_chiffres` est l'opération qui à deux entiers naturel n et $b > 0$ associe le nombre de chiffres dans l'écriture de n en base b.
6. `pgcd` est l'opération qui à deux entiers naturels a et b associe leur plus grand diviseur commun.
7. `est_premier` est l'opération qui détermine si un entier naturel passé en paramètre est premier ou non.
8. `est_un_carré` est l'opération qui détermine si un entier naturel passé en paramètre est un carré parfait.

Pour chacune d'elle, il est demandé de suivre les cinq premiers points de la démarche algorithmique présentée ci-dessus.

Exercice 2

La **suite de Fibonacci** est la suite $(f_n)_{n\in\mathbb{N}}$ d'entiers naturels définie par $f_0 = f_1 = 1$ et pour tout entier $n > 1$, $f_n = f_{n-1} + f_{n-2}$. L'entier f_n est appelé le n^{e} **nombre de Fibonacci**. On considère l'opération `fibonacci` qui à tout entier naturel n associe le n^{e} nombre de Fibonacci.

1. Suivre les 4 premiers points de la démarche algorithmique page 3.
2. Combien la fonction `fibonacci` produit-elle d'appels récursifs pour le calcul de `fibonacci(5)` ?
3. Soit A_n le nombre d'appels à la fonction `fibonacci` pour le calcul de `fibonacci(n)` pour tout entier n. Soit S_n le nombre d'additions nécessaires dans le calcul de `fibonacci(n)`.
   a) Donnez une formule de récurrence pour A_n et S_n. (**Indication** : considérez les suites $a_n = A_n + 2$ et $s_n = S_n + 1$).
   b) Comparez A_n, S_n et f_n.
4. Écrire un algorithme itératif pour l'opération `fibonacci`.

Exercice 3

Étant donné deux entiers naturels $0 \leq p \leq n$, le coefficient binomial $\binom{n}{p}$ est défini par $\binom{n}{p} = \binom{n+1}{p+1} = \dfrac{n \times (n-1) \dots (n-p+1)}{p!}$.

Pour tous entiers $0 < p < n$, on a :

$$\binom{n+1}{p+1} = \binom{n}{p} + \binom{n}{p+1}$$

1. En mathématiques, il est d'usage de définir $\binom{n}{p}$ de la manière suivante : $\binom{n}{p} = \frac{n!}{p!(n-p)!}$. Citez des avantages et des inconvénients de chacune des définitions du coefficient binomial.
2. Donnez le profil de la fonction `binom` qui pour tous entiers naturels n et p renvoie 0 si $n < p$ et renvoie $\binom{n}{p}$ sinon.
3. Ecrire un algorithme récursif réalisant la fonction `binom` n'utilisant que des additions d'entiers naturels.
4. Donnez la trace mémoire du calcul de `binom`$(5, 2)$ par l'algorithme précédent.

Exercice 4

On considère l'opération `puissance` définie dans l'exercice 1

1. Combien d'appels récursifs et de multiplications sont nécessaires au calcul de x^n (en fonction de n) ?
2. Un autre moyen pour calculer x^n est donné par la méthode suivante, appelée « square-and-multiply », et est basé sur les relations de récurrence suivantes : soit $x \in \mathbb{R}$ et $n \in \mathbb{N}^\star$. Montrez que :

$$x^n = \begin{cases} (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x(x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases} \tag{1.2}$$

   a) Calculez 2^{26} en utilisant les relations de récurrence (1.2).

   b) Combien de multiplications sont-elle nécessaires pour ce calcul ?

   c) `square_and_multiply` est l'opération qui à un réel x et un entier n associe x^n. Écrire un algorithme récursif pour l'opération `square_and_multiply` qui utilise les relations de récurrence (1.2).

   d) Combien de multiplications sont nécessaires pour le calcul de x^n (en fonction de n) ?

1.2.2 Sur les chaînes de caractères

Exercice 5

On considère les opérations suivantes :

1. `longueur` est l'opération qui à une chaîne de caractères associe le nombre de caractères qui la composent.
2. `premiere_occurrence` est l'opération qui à une chaîne de caractères `s` et un caractère `c` associe l'indice de la première occurrence de `c` dans `s`. Si `c` ne possède aucune d'occurence dans `s`, alors

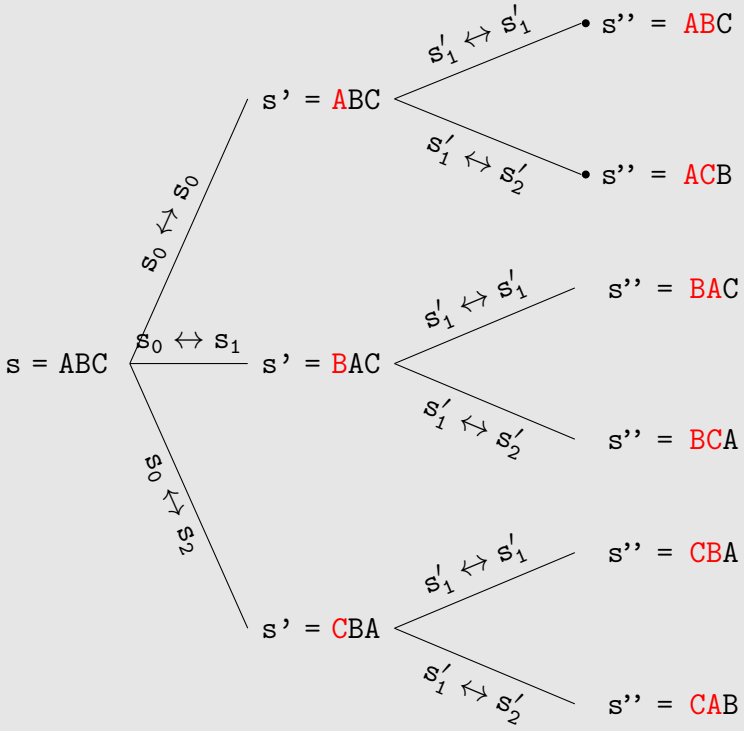
l'opération `premiere_occurrence` renvoie -1.

3. `nb_occurrences` est l'opération qui à une chaîne de caractères `s` et un caractère `c` associe le nombre d'occurrences de `c` dans `s`. Exemple : `nb_occurrences(babc, b) = 2`.
4. `inverser` est l'opération qui à une chaîne de caractères associe la chaîne de caractères inversées. Exemple : `inverser(aabc) = bcaa`.
5. `palindrome` est l'opération qui à une chaîne de caractères associe `vraie` si et seulement si la chaîne `s` est un palindrome.

Pour chacune d'elle, il est demandé de suivre les cinq premiers points de la démarche algorithmique présentée ci-dessus.

Exercice 6 [*]

On représente l'ensemble des permutations de la chaîne de caractères `ABC` par l'arbre suivant :



L'ensemble des permutations de la chaîne `ABC` sont les étiquettes des feuilles de cet arbre.

1. Complétez cet arbre pour générer l'ensemble des permutations de la chaîne de caractères ABCD.
2. Écrivez une procédure récursive prenant en entrée une chaîne de caractères et affichant l'ensemble de ses permutations.

1.2.3 Sur les tableaux

Exercice 7

On considère les opérations suivantes :

1. `maximum` est l'opération qui à un tableau de nombres associe son plus grand élément.
2. `somme` est l'opération qui à un tableau de nombres associe la somme de ses éléments.
3. `est_trié` est l'opération qui renvoie `vraie` si et seulement si le tableau passé en entré est trié.

Pour chacune d'elle, il est demandé de suivre les cinq premiers points de la démarche algorithmique présentée ci-dessus.

Exercice 8

Soit `T` un tableau trié en ordre croissant. On effectue une recherche **dichotomique** d'une valeur x comme suit :

— `g` l'indice de gauche initialisé à 0.

— `d` l'indice de droite initialisé à $n-1$.

— `trouve` le booléen de la recherche initialisé à `Faux`.

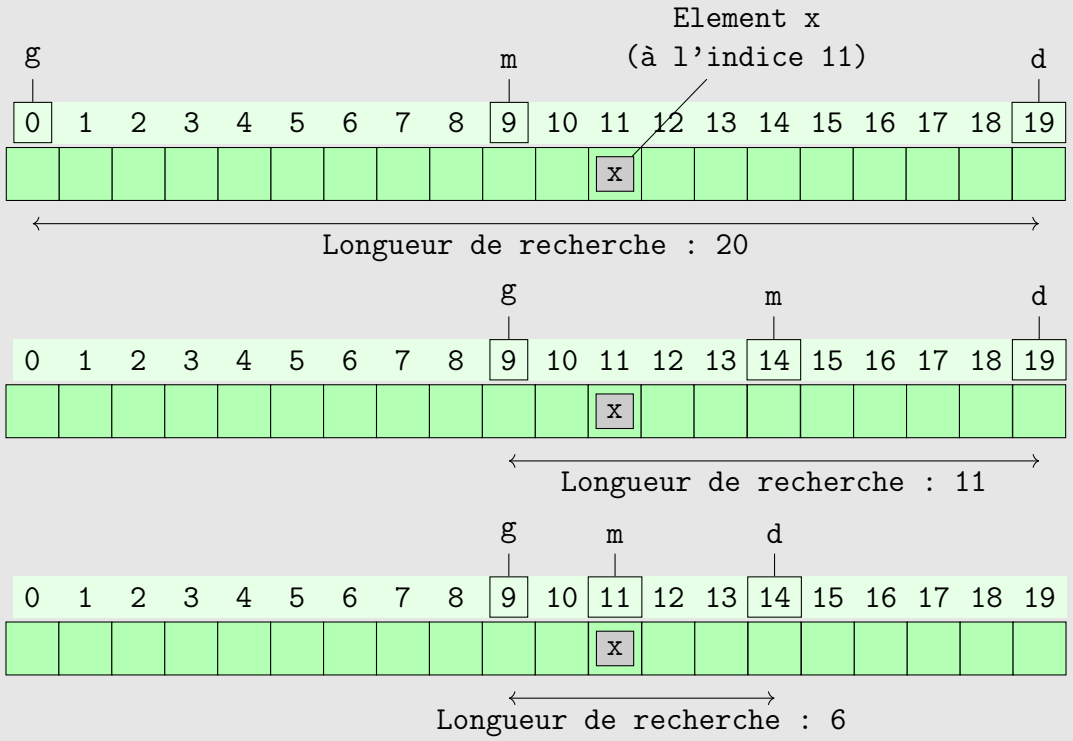
— `m` l'indice milieu de l'intervalle $[\![\texttt{g} \dots \texttt{d}]\!]$.

Alors

[1] Si `T[m]=x` alors on fixe la variable `trouve` à `Vrai`.

[2] Sinon si `T[m]` $<$ `x` alors la position de x est forcément après `m` et on fixe `g` à `m+1`. Dans le cas contraire, on fixe d en $m-1$ (car `T[m]` $>$ `x`).

[3] On répète les opérations (1) et (2) jusqu'à ce que `trouve` soit `Vrai` ou que `g` soit strictement supérieur à `d`.



`recherche_Dichotomique` est l'opération qui à partir d'un tableau `T` trié de taille `n` et un élément `x` effectue une recherche dichotomique de `x` dans `T` et renvoie l'indice d'une occurrence de `x` dans `T`.

1. Suivez les points [1] à [4] de la démarche algorithmique page 3. On suppose pour cette question que l'élément `x` appartient toujours au tableau `T`).
2. Comment faut-il si l'on n'est pas certains que l'élément `x` appartient à `T` ? Si `x` n'appartient pas à `T` votre fonction renverra `-1`.

Exercice 9

Le **tri fusion**, ou **tri dichotomique**, est un algorithme de tri qui met en œuvre le paradigme *diviser pour régner* comme suit :

Algorithme 4 : Tri fusion d'un tableau d'entiers

```
1 Procedure tri_fusion(T : un tableau entiers, n : un entier naturel)
2 début
      // n : longueur du tableau T
3     si n = 1 alors
4         // Le tableau est déjà trié, il n'y a rien à faire
5     sinon
6         Décomposer le tableau T en deux sous-tableaux de même longueur (±1)
7         Appliquer la procédure tri_fusion aux deux sous-tableaux
8         Fusionner les deux tableaux triés en un seul tableau trié
```

1. Appliquer « à-la-main » la procédure `tri_fusion` au tableau suivant :

39	27	43	3	9	82	10

2. Écrire une procédure `fusion` prenant en entrée deux sous-tableaux triés et qui les fusionne « en place » en un (sous-)tableau trié du tableau initial.
3. Écrire une procédure `tri_fusion` qui exécute le tri fusion.

1.2.4 Récursivité croisée

On parle de **récursivité croisée** lorsque deux opérations s'appellent l'une et l'autre récursivement.

Exercice 10

On considère les trois suites $(u_n)_{n\in\mathbb{N}}$, $(v_n)_{n\in\mathbb{N}}$ et $(w_n)_{n\in\mathbb{N}}$ définies par : $u_0 = 1$, $v_0 = 2$, $w_0 = 3$ et pour tout $n \in \mathbb{N}$,

$$\begin{aligned} u_{n+1} &= 2u_n + 3v_n + w_n \\ v_{n+1} &= u_n + v_n + 2w_n \\ w_{n+1} &= u_n + 4v_n + w_n \end{aligned}$$

1. Écrire un algorithme itératif, qui pour tout entier naturel $n \in \mathbb{N}$, affiche les valeurs de u_n, v_n et

w_n.

2. Écrire un algorithme récursif, qui pour tout entier naturel $n \in \mathbb{N}$, affiche les valeurs de u_n, v_n et w_n.
3. Traduire ces algorithmes en C

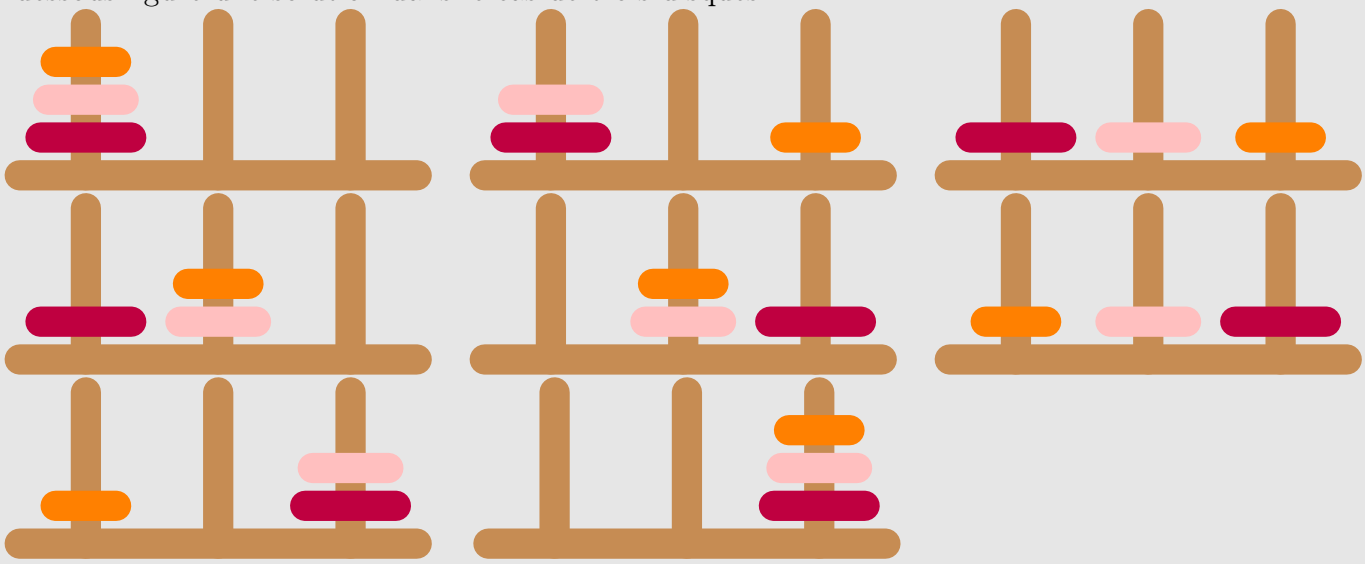
1.2.5 Dans un jeu

Exercice 11

Le jeu des Tours de Hanoï est constitué de trois piquets A, B et C, placés verticalement, et de n disques de taille décroissante. Chacun des disques est percé en son centre pour être mis autour de l'un ou l'autre des trois piquets. Les n disques sont initialement placés par taille décroissante autour du piquet A (celui de gauche), formant ainsi une tour. Le but du jeu consiste à déplacer les disques jusqu'à parvenir à la situation finale dans laquelle tous les disques se retrouvent autour du piquet C par ordre de taille décroissante. Les disques peuvent aller et venir librement sur les piquets, en suivant deux règles :

1. on ne déplace qu'un seul disque à la fois ;
2. un disque ne peut jamais être posé sur un disque plus petit.

Ci-dessous figure une solution dans le cas de trois disques.



1. Effectuez quelques essais à la main avec 4, 5 et 6 disques. Pour chaque cas, quel est le nombre minimum de déplacements de disques à effectuer ?
2. Soit n un entier naturel non nul. Montrez que si l'on sait résoudre ce jeu pour n disques, alors on sait le résoudre pour $n+1$ disques.
3. En déduire, un algorithme récursif qui affiche chaque étape de la résolution du jeu pour n disque.