Chapitre 4 : Récursivité

1 Introduction

Nous avons déjà écrit de nombreuses fonctions *récursives*, c'est-à-dire des fonctions intervenant dans leur propre définition, comme par exemple la version suivante de la fonction factorielle :

[1]:
```
let rec fact n = match n with
    | 0 -> 1
    | _ -> n * fact (n-1)
;;
```

[1]:
```
val fact : int -> int = <fun>
```

Se pose alors la question de la *terminaison* : pour une valeur donnée de `n`, l'appel `fact n` termine-t-il ? Ceci n'est pas évident, comme le prouve l'exemple suivant :

[2]:
```
let rec inarretable n = match n with
    | 0 -> 1
    | n -> n * inarretable n
;;
```

[2]:
```
val inarretable : int -> int = <fun>
```

Dans le cas de la fonction `fact`, nous pouvons appliquer le raisonnement suivant : pour un entier naturel non nul `n`, la suite des arguments des appels récursifs à la fonction `fact` est une suite strictement décroissante d'entiers naturels, donc ne peut pas être infinie ; par conséquent, `fact n` termine pour tout entier naturel `n`.

On voudrait aussi s'assurer que la fonction renvoie bien le résultat attendu : c'est le problème de la *correction*. Dans le cas de la fonction `fact`, on peut raisonner par récurrence : si n vaut 0, la fonction renvoie bien $n!$, et pour tout entier naturel n, si `fact` n renvoie bien $n!$ alors `fact` $(n+1)$ renvoie $(n+1)!$. Comme nous allons le voir, la notion de récursivité est proche de celle de récurrence.

L'objectif des deux premières parties de ce chapitre est de généraliser ce raisonnement à des cas où l'on ne peut pas (ou en tout cas pas facilement) raisonner sur des entiers naturels.

Il faut néanmoins garder en tête qu'il n'existe pas d'algorithme général permettant de déterminer à l'avance si l'appel à une fonction termine. C'est un résultat d'Alan Turing, qui exprime que *le problème de l'arrêt est indécidable*. Dans le cas contraire, on pourrait en effet disposer d'une

fonction `termine : (int -> int) -> int -> bool` qui prendrait en argument une fonction `f : int -> int` et un entier `n` et qui renverrait `true` si l'appel `f n` termine, `false` sinon. En définissant la fonction :

[3]:
```
let rec absurde n =
  if termine absurde n
  then absurde n
  else 0
;;
```

```
File "[3]", line 2, characters 5-12:
2 |   if termine absurde n
        ^^^^^^^
Error: Unbound value termine
```

On obtient alors une fonction qui termine si et seulement si elle ne termine pas. L'algorithme suivant est un exemple classique d'algorithme dont on suspecte la terminaison, mais dont la terminaison n'a jamais été démontrée. C'est la *conjecture de Syracuse*.

[4]:
```
let rec syracuse n = match n with
    | 0 -> 1
    | 1 -> 1
    | n when n mod 2 = 0 -> syracuse (n / 2)
    | n -> syracuse (3 * n + 1)
;;
```

[4]:
```
val syracuse : int -> int = <fun>
```

2 Ordres bien fondés

Pour démontrer la terminaison de la fonction `fact`, nous avons utilisé un raisonnement par récurrence simple. Cela ne peut pas suffire pour certaines fonctions récursives. Considérons par exemple le calcul des termes de la suite de Fibonacci :

[5]:
```
let rec fibo n = match n with
    | 0 | 1 -> 1
    | _ -> fibo (n-1) + fibo (n-2)
;;
```

[5]:
```
val fibo : int -> int = <fun>
```

Il est clair que dans ce cas il faudra utiliser une récurrence double ou forte. Mais les choses peuvent être encore plus compliquées, avec des fonctions récursives de deux variables par exemple :

[6]:
```
let rec ackermann m n = match m, n with
    | 0, _ -> n + 1
    | _, 0 -> ackermann (m - 1) 1
    | n, p -> ackermann (m - 1) (ackermann m (n - 1))
;;
```

[6]:
```
val ackermann : int -> int -> int = <fun>
```

On ne peut plus utiliser ici, dans $\mathbb{N}^2$, la notion de *prédecesseur* d'un entier. Pour étudier ce type de fonctions, nous allons devoir utiliser des résultats sur les relations d'ordre qui ne figurent pas au programme de mathématiques.

2.1 Rappels sur les relations d'ordre

Définition :
Une relation binaire $\preccurlyeq$ sur un ensemble E est appelée une relation d'ordre sur E lorsqu'elle est :

- réflexive : $\forall x \in E, x \preccurlyeq x$;
- antisymétrique : $\forall (x, y) \in E^2, (x \preccurlyeq y \text{ et } y \preccurlyeq x) \Rightarrow x = y$
- et transitive : $\forall (x, y, z) \in E^3, (x \preccurlyeq y \text{ et } y \preccurlyeq z) \Rightarrow x \preccurlyeq z$.

Une relation d'ordre $\preccurlyeq$ est dite relation d'ordre **total** lorsque deux éléments sont toujours comparables :

$$\forall (x, y) \in E^2, x \preccurlyeq y \text{ ou } y \preccurlyeq x$$

On dit alors que $(E, \preccurlyeq)$ est un ensemble totalement ordonné.

Dans le cas contraire, on dit que la relation d'ordre est **partielle** $\preccurlyeq$ et que $(E, \preccurlyeq)$ est un ensemble partiellement ordonné.

Proposition : Soient $(E, \preccurlyeq_E)$ et $(F, \preccurlyeq_F)$ deux ensembles ordonnés. Alors l'*ordre lexicographique* défini sur $E \times F$ par :

$$\forall (a, b), (c, d) \in E \times F, (a, b) \preccurlyeq (c, d) \Leftrightarrow (a \prec_E c \text{ ou } (a = c \text{ et } b \preccurlyeq_F d))$$

où $\prec_E$ désigne la relation d'ordre strict associé à $\preccurlyeq_E$, est une relation d'ordre sur $E \times F$

Exemple : Ordre lexicographique sur $\mathbb{N}^2$.

2.2 Ordre bien fondé

Définition : Soit E un ensemble. Un ordre $\preccurlyeq$ sur E est dit *bien fondé* lorsqu'il n'existe pas de suite strictement décroissante d'éléments de E pour cet ordre. On dit alors que $(E, \preccurlyeq)$ est un ensemble *bien fondé*.

Remarque : $\preccurlyeq$ est bien fondé si et seulement si toute suite décroissante pour $\preccurlyeq$ est stationnaire.

Proposition : Soit $(E, \preccurlyeq)$ un ensemble bien fondé et $F \subset E$ non vide. Alors $(F, \preccurlyeq)$ est bien fondé.

Définition : Soit $(E, \preccurlyeq)$ un ensemble ordonné et $A \subset E$. On dit que :

- $m \in A$ est un *élément minimal* de A lorsque $\forall a \in A, a \preccurlyeq m \Rightarrow a = m$.

- $m \in A$ est un *plus petit élément* de A lorsque $\forall a \in A, m \preccurlyeq a$.

Remarque :

- Si m est un plus petit élément de A, alors m est unique.
- Dans le cas d'un ordre total, si m est un élément minimal de A alors m est un plus petit élément de A.

Proposition : Un ensemble ordonné $(E, \preccurlyeq)$ est bien fondé si et seulement si toute partie non vide de E admet un élément minimal.

Démonstration :

- Soit $(E, \preccurlyeq)$ un ensemble bien fondé. Soit $A \subset E$ non vide, montrons par l'absurde que A admet un élément minimal. Si A n'admet pas d'élément minimal, alors $\forall m \in A, \exists a \in A, a \prec m$. Comme A est non vide, il existe un élément $u_0 \in A$ et on peut construire une suite strictement décroissante d'éléments de A : absurde. Donc A admet un élément minimal.
- Soit $(E, \preccurlyeq)$ un ensemble ordonné tel que toute partie non vide de E admet un élément minimal. Supposons qu'il existe une suite $(u_n)_{n\in\mathbb{N}}$ strictement décroissante. Soit $A = \{u_n | n \in \mathbb{N}\}$. Alors A admet un élément minimal m et il existe n_0 tel que $m = u_{n_0}$. Or $u_{n_0+1} \in A$ et $u_{n_0+1} \prec u_{n_0}$: absurde. Donc E est bien fondé.

Définition : Soit E un ensemble. On dit qu'un ordre $\preccurlyeq$ sur E est un *bon ordre* si toute partie non vide admet un plus petit élément. On dit alors que $(E, \preccurlyeq)$ est un ensemble *bien ordonné*.

Remarque : - un ensemble bien ordonné est totalement ordonné ; - la réciproque est fausse ; - un ensemble bien fondé et totalement ordonné est bien ordonné. *Exemple :* Considérons $\mathbb{N}^2$ muni de l'ordre lexicographique :

$$(a, b) \preccurlyeq (c, d) \Leftrightarrow (a < c) \text{ ou } (a = c \text{ et } b \leqslant d)$$

Soit $A \subset \mathbb{N}^2$ non vide.

Posons $a_0 = \min\{a \in \mathbb{N} | \exists b \in \mathbb{N}, (a, b) \in A\}$ et $b_0 = \min\{b \in \mathbb{N} | (a\_0, b) \in A\}$. Alors pour tout $(a, b) \in A$, $a_0 \leqslant a$ et si $a = a_0$, alors $b_0 \leqslant b$ donc $(a_0, b_0) \preccurlyeq (a, b)$.

Par conséquent, $(a, b) \preccurlyeq (a_0, b_0) \Rightarrow (a, b) = (a_0, b_0)$.

Finalement, $(\mathbb{N}^2, \preccurlyeq)$ est bien fondé (et même bien ordonné puisque cet ordre est total).

2.3 Principe d'induction

Définition : On appelle *prédicat* sur un ensemble E une application de E dans l'ensemble des booléens.

Théorème : Principe d'induction
Soit $(E, \preccurlyeq)$ un ensemble bien fondé, soit $\mathcal{M}$ l'ensemble de ses éléments minimaux. Si un prédicat $\mathcal{P}$ sur E vérifie :

- $\forall x \in \mathcal{M}$, $\mathcal{P}(x)$
- $\forall x \in E \setminus \mathcal{M}$, $(\forall y \prec x, \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$

Alors pour tout $x \in E$, $\mathcal{P}(x)$.

Démonstration : Par l'absurde, supposons qu'il existe $x_0 \in E$ tel que $\mathcal{P}(x_0)$ soit faux. Alors x_0 n'appartient pas à $\mathcal{M}$, et il existe $x_1 \in E$ tel que $x_1 \prec x_0$ et tel que $\mathcal{P}(x_1)$ est faux. De même, x_1 ne peut pas appartenir à $\mathcal{M}$. On peut donc construire une suite infinie strictement décroissante, ce qui est en contradiction avec le fait que E est bien fondé.

Remarque : Pour $E = \mathbb{N}$ muni de l'ordre usuel, on retrouve le principe de récurrence forte.

3 Terminaison

On considère une fonction récursive f.

Théorème : Soit $\mathcal{A}$ l'ensemble des arguments de f. Soit φ une application de $\mathcal{A}$ vers un ensemble bien fondé E. Soit $\mathcal{M}_{\mathcal{A}}$ l'ensemble des arguments x tels que $\varphi(x)$ est un élément minimal de E.
Si :

- la fonction f termine pour tout $x \in \mathcal{M}_{\mathcal{A}}$;
- pour tout $x \in \mathcal{A} \setminus \mathcal{M}_{\mathcal{A}}$, le calcul de $f(x)$ ne nécessite qu'un nombre fini (éventuellement nul) d'appels à f, sur des arguments y tels que $\varphi(y) \prec \varphi(x)$, et la terminaison de ces appels entraîne celle de $f(x)$

alors la fonction f termine sur tout argument de $\mathcal{A}$.

Démonstration : On utilise le principe d'induction pour la propriété suivante sur $z \in E$: $\mathcal{P}(z)$ "les appels $f(x)$ avec $\varphi(x) = z$ terminent."

Définition : Les éléments x de $\mathcal{A}$ pour lesquels le calcul de $f(x)$ ne nécessite aucun appel à f sont appelés *cas de base* ou *cas terminaux*. En effet en informatique les calculs sont "descendants" et les cas terminaux sont les derniers à être calculés, et ceux qui déterminent que le calcul va s'arrêter. Lorsqu'un appel à f pour un argument x est effectué, on évalue la condition "x est-il un cas terminal ?" : cette condition est appelée *condition d'arrêt*.

Remarque : L'ensemble $\mathcal{A}$ sera souvent lui-même un ensemble bien fondé, et la fonction φ utilisée sera alors la fonction identité.

Dans d'autres cas, l'application φ est souvent une application à valeurs dans $\mathbb{N}$; par exemple, à une liste, on peut associer sa longueur.

Exemple : Considérons la fonction :

[7]:
```
let rec length l =
  match l with
  | [] -> 0
  | t::q -> 1 + length q
;;
```

[7]:
```
val length : 'a list -> int = <fun>
```

A une liste, on peut associer sa longueur : - La fonction `length` termine pour la liste de longueur 0 ; - Pour une liste de longueur au moins 1, un seul appel récursif est réalisé, et il porte sur une liste de longueur strictement inférieure.

Par conséquent, la fonction termine.

Exemple : La fonction d'Ackermann suivante termine :

[8]:
```
let rec ack n p =
  match (n, p) with
  | 0, _ -> p+1
  | _, 0 -> ack (n-1) 1
  | _, _ -> ack (n-1) (ack n (p-1))
;;
```

[8]:
```
val ack : int -> int -> int = <fun>
```

En effet, on considère ici $\mathbb{N}^2$ muni de l'ordre lexicographique. - Les cas de bases sont les couples dont la première coordonnée est nulle. Ils contiennent $(0,0)$, l'élément minimal de l'ordre lexicographique. - Pour $(n,0)$ avec $n \neq 0$, le seul appel récursif porte sur un couple strictement plus petit. - Pour (n,p) avec n et p tous les deux non nuls, il y a deux appels récursifs, dont les arguments sont $(n,p-1)$ et \$(n-1, \$ ack $(n,p-1)$) qui sont bien strictement plus petits que (n,p).

La fonction termine bien.

Remarque : Il n'est pas forcément facile de trouver un ensemble bien fondé et une application φ associée pour une fonction récursive donnée. On ne sait par exemple pas le faire pour la fonction de Syracuse.

4 Correction

Pour démontrer qu'une fonction récursive calcule ce qu'elle doit calculer, on peut raisonner comme pour la terminaison, en adaptant le prédicat à démontrer.

Théorème : Soit $\mathcal{A}$ l'ensemble des arguments de f. Soit φ une application de $\mathcal{A}$ vers un ensemble bien fondé E. Soit $\mathcal{M}_{\mathcal{A}}$ l'ensemble des arguments x tels que $\varphi(x)$ est un élément minimal de E. Pour $z \in E$, on considère le prédicat $\mathcal{P}(z)$: "Pour x tel que $\varphi(x) = z$, $f(x)$ a la bonne valeur". Si :

- pour tout $x \in \mathcal{M}_{\mathcal{A}}$, $\mathcal{P}(\varphi(x))$;
- pour tout $x \in \mathcal{A} \setminus \mathcal{M}_{\mathcal{A}}$, le calcul de $f(x)$ ne nécessite qu'un nombre fini (éventuellement nul) d'appels à f, sur des arguments y_1,..., y_N tels que pour tout $k \in [\![1, N]\!]$, $\varphi(y_k) \prec \varphi(x)$, et que $(\forall k \in [\![1, N]\!], \mathcal{P}(\varphi(y_k))) \Rightarrow \mathcal{P}(\varphi(x))$

alors pour tout $x \in \mathcal{A}$, $\mathcal{P}(\varphi(x)$, donc la valeur de $f(x)$ est correcte.

Démonstration : C'est le principe d'induction.

4.0.1 Exercice 1

Considérons les deux fonctions suivantes de calcul de $\binom{n}{p}$, rencontrées dans le chapitre 1 :

[9]:
```
let rec binome1 n p =
  match n, p with
    | 0, 0 -> 1
    | 0, _ -> 0
    | n, p  ->
       if  p < 0 || p > n
       then 0
       else binome1 (n-1) (p-1) + binome1 (n-1) p
    ;;

let rec binome2 n p =
  if p < 0 || p > n
  then 0
  else if p = 0
  then 1
  else n* binome2 (n-1) (p-1)/p
    ;;
```

[9]:
```
val binome1 : int -> int -> int = <fun>
```

[9]:
```
val binome2 : int -> int -> int = <fun>
```

Montrer que ces fonctions terminent et renvoient bien le résultat voulu.

5 Pratique de la récursivité

5.1 Récursivité croisée

On rappelle que and permet de définir simultanément plusieurs objets. Il est possible de définir simultanément plusieurs fonctions, qui s'appellent mutuellement (on parle de récursivité mutuelle ou croisée). Par exemple, nous pouvons écrire un algorithme calculant les termes successifs des deux suites adjacentes dont la limite commune est la moyenne arithmético-géométrique de deux réels `a` et `b`:

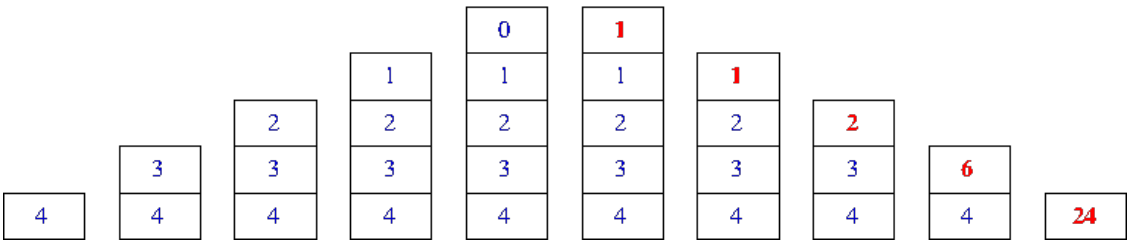
[10]:
```
let rec u a b n = match n with
    | 0 -> a
    | _ -> (u a b (n-1) +. v a b (n-1))/. 2.
and v a b n = match n with
    | 0 -> b
    | _ -> sqrt ((u a b (n-1)) *. (v a b (n-1)))
;;
```

[10]:
```
val u : float -> float -> int -> float = <fun>
val v : float -> float -> int -> float = <fun>
```

5.2 La pile d'appels

Si l'on revient à la fonction `fact`, et que l'on cherche par exemple à calculer `fact 4`, regardons ce qu'il se passe : un appel à `fact 4` est effectué. La condition d'arrêt n'étant pas vérifiée, ce calcul est mis en attente, et un appel à `fact 3` est effectué. Et ainsi de suite pour `fact 2`, `fact 1` et `fact 0`. 0 étant un cas terminal, on peut alors calculer `fact 0`, puis `fact 1`, ce qui permet de calculer `fact 2`, et ainsi de suite jusqu'à `fact 4`. Il existe une structure de données stockant les calculs à effectuer, et indiquant dans quel ordre les effectuer : c'est la *pile d'appels*, ou *pile d'éxécution*. Les calculs en attente y sont *empilés*. Une fois effectués, ils sont *dépilés*. Une fonction récursive étant une fonction qui s'appelle elle-même, ses appels s'empilent dans la pile d'appels ; une fois arrivé à un cas terminal, le nombre d'éléments de la pile se réduit. Enfin, la récursion s'arrête lorsque la pile est vide, c'est-à-dire lorsqu'on dépile l'élément correspondant au premier appel de la fonction.

Voici la représentation de la pile d'appels dans le cas de l'exemple précédent. En bleu figurent les arguments correspondant aux appels à `fact` mis en attente, et en rouge et gras les résultats de ces appels.



En OCaml, on peut vérifier ce comportement en *traçant* les appels de la fonction. Cette fonctionnalité n'est hélas pas disponible sur Capytale.

[11]:
```
#trace fact;;
```

[11]:
```
fact is now traced.
```

Remarque On peut cesser de tracer la fonction `fact` de la manière suivante :

[12]:
```
#untrace fact;;
```

[12]:
```
fact is no longer traced.
```

5.3 Capacité de la pile d'appels

Le nombre d'appels imbriqués par une fonction récursive peut être important. La pile d'appels a une capacité limitée ; si la pile est pleine, un appel supplémentaire produit un dépassement de capacité :

[13]:
```
let rec somme n =
  if n = 0
  then 0
  else n + somme (n-1);;
```

[13]:
```
val somme : int -> int = <fun>
```

[14]:
```
somme 1000000;;
```

```
Stack overflow during evaluation (looping recursion?).
Raised by primitive operation at somme in file "[13]", line 4, characters 11-22
Called from somme in file "[13]", line 4, characters 11-22
Called from somme in file "[13]", line 4, characters 11-22
Called from somme in file "[13]", line 4, characters 11-22
...
```

Un problème fréquent est celui où les appels se chevauchent : la pile d'appels garde en mémoire à plusieurs endroits des appels au même calcul. Cette situation survient dans la fonction `fibo` définie plus haut. Un appel à `fibo 5` déclenche un appel à `fibo 4` et un autre à `fibo 3`. Ces deux appels sont empilés dans la pile d'éxécution. Mais l'appel à `fibo 4`déclenche lui-même un second appel à `fibo 3`, qui va lui aussi être empilé. au bout du compte ces deux appels seront résolus séparément, et le même calcul aura été éxécuté deux fois. C'est encore pire pour les cas terminaux, qui sont appelés une multitude de fois. Ce phénomène conduit à un remplissage très rapide de la pile d'appels.

5.4 Récursivité terminale

Une fonction récursive est dite *terminale* si la dernière opération réalisée pour calculer $f(x)$ est un appel récursif. L'intérêt d'avoir une fonction récursive terminale est qu'un appel récursif à f ne nécessite pas d'empilement sur la pile d'appels : l'appel récursif peut prendre la place de l'appel en cours dans la pile, puisqu'il n'y a pas d'opération à effectuer une fois l'appel récursif terminé. Certains compilateurs, comme celui-ci d'OCaml, sont capables de détecter les fonctions récursives terminales et limitent alors les empilements dans la pile d'appels. Par exemple, les fonctions `fact`, `fibo` et `somme` ne sont pas terminales.

Mais le calcul du pgcd de deux entiers naturels à l'aide de l'algorithme d'Euclide est récursif terminal :

[15]:
```
let rec pgcd p q = match p with
    | 0 -> q
    | _ -> pgcd (q mod p) p ;;
```

[15]:
```
val pgcd : int -> int -> int = <fun>
```

5.5 Accumulateur

Pour éviter un dépassement de capacité, on peut utiliser un outil : l'*accumulateur*.

Voici ce que donne une nouvelle version de la fonction `somme`, qui utilise un accumulateur :

[16]:
```
(* Cette fonction calcule 0+1+...+(n-1)+ n + acc*)
let rec somme_avec_acc n acc =
  match n with
  | 0 -> acc
  | _ -> somme_avec_acc (n-1) (n+acc)
;;
```

[16]:
```
val somme_avec_acc : int -> int -> int = <fun>
```

On utilise ici un deuxième argument pour effectuer les additions. Il suffit de prendre pour ce deuxième paramètre la valeur 0 pour calculer `somme n` :

[17]:
```
let somme n =
  somme_avec_acc n 0
;;
```

[17]:
```
val somme : int -> int = <fun>
```

[18]:
```
somme 100000000 ;;
```

[18]:
```
- : int = 5000000050000000
```

Lorsqu'on utilise un accumulateur, il n'est pas forcément agréable de devoir faire les appels à la fonction avec un paramètre supplémentaire. On définit souvent la fonction avec accumulateur comme une fonction locale à une autre fonction, qui se contentera de l'appeler avec la bonne valeur initiale pour l'accumulateur. On en profitera généralement pour lui donner un nom plus court :

[19]:
```
let somme n =
  let rec aux n acc =
    match n with
    | 0 -> 0
    | _ -> aux (n - 1) (n + acc)
  in aux n 0
;;
```

[19]:
```
val somme : int -> int = <fun>
```

Si l'utilisation d'un accumulateur présente l'intérêt de diminuer le nombre d'appels empilés, elle rend toutefois le code beaucoup moins lisible.

6 Exercices divers

6.1 Exercice 2

Justifier que la fonction récursive calculant le pgcd de deux entiers naturels non tous deux nuls par l'algorithme d'Euclide, écrite plus haut, termine.

6.2 Exercice 3

On dispose d'un stock illimité de billets et de pièces de valeurs c_1, ..., c_p. On souhaite dénombrer le nombre de manières possibles d'obtenir une somme donnée avec ces espèces. Écrire une fonction récursive prenant en paramètre la somme à atteindre et la liste des valeurs de pièces et calculant cette quantité.

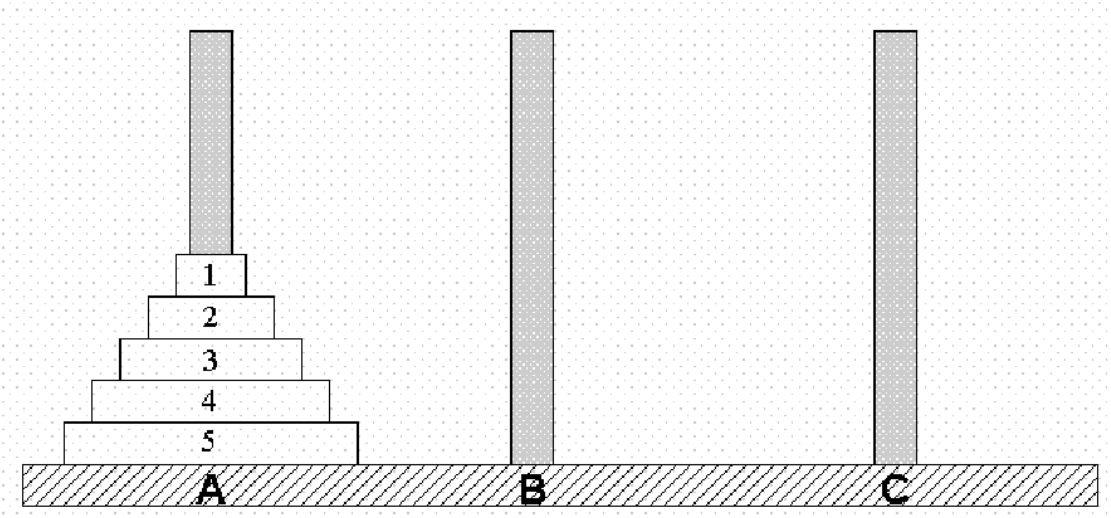
Il y a par exemple 11 manières possibles de payer 10€ avec des pièces et billets de 1, 2, 5 et 10€ (ce qui est facile à calculer à la main), et 451 manières possibles de payer 50€ avec des pièces et billets de 1, 2, 5, 10, 20 et 50€.

6.3 Exercice 4

Écrire une version avec accumulateur(s) des fonctions `fact` et `fibo`.

6.4 Exercice 5

Les *tours de Hanoï* est un puzzle inventé par le mathématicien français Édouard Lucas : il est constitué de trois tiges sur lesquelles peuvent être enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous enfilés sur la même tige, du plus grand au plus petit, et le but du jeu et de déplacer tous ces disques sur la 3ème tige en respectant les règles suivantes : - un seul disque peut être déplacé à la fois ; - on ne peut jamais poser un disque sur un disque de diamètre inférieur.



1) Écrire une fonction hanoi : `int -> unit` affichant une solution du jeu, en écrivant tous les mouvements nécessaires de la forme `a->b` où `a` est la tour d'origine et `b` la tour d'arrivée, en utilisant la fonction suivante :

[20]:
```
let prt_tp a b =
print_int a; print_string "->";
print_int b; print_string "\n"
;;
```

[20]:
```
val prt_tp : int -> int -> unit = <fun>
```

Par exemple, on a l'exécution suivante :

[21]:
```
hanoi 3;;
```

```
File "[21]", line 1, characters 0-5:
1 | hanoi 3;;
    ^^^^^
Error: Unbound value hanoi
```

2) Déterminer le nombre exact de déplacements lors de l'exécution de la fonction `hanoi`.