TD - Preuves d'algorithmes
Terminaison - Correction.

Exercice 1 (Somme).
On donne le code suivant :

```
let sum t =
  let s = ref 0 in
  let n = Array.length in
  for i = 0 to n-1 do
    s := !s + t.(i)
  done;
  !s;;
```

1. Que fait ce code ? Donner le prototype OCaml de cette fonction.
2. Prouver la terminaison de cet algorithme.
3. Prouver la correction de cet algorithme.

Corrigé de l'exercice 1. *[Retour à l'énoncé]*

1. Ce code calcule la somme des valeurs contenues dans un tableau d'entiers. Le prototype de cette fonction est :

   ```
   val sum: int array -> int <fun>
   ```

2. La preuve de terminaison de cet algorithme est immédiate car il s'agit d'un algorithme itératif avec une boucle inconditionnelle qui se termine toujours : l'indice de boucle n'est pas modifiable/modifiée à l'intérieur d'un tour de boucle.
3. Il s'agit d'un **algorithme itératif** utilisant des références OCaml, c'est-à-dire des **variables modifiables** : on s'oriente donc vers une démonstration à l'aide d'un **invariant de boucle**.
   On note n la taille du tableau donné en entrée de la fonction. Cette valeur est stockée dans la variable immuable `n`.
   On note t_k la valeur d'indice $0 \leq k \leq n-1$ du tableau `t` : $t = [t_0, \ldots, t_{n-1}]$
   On note s et i les contenus respectifs des variables `s` et `i`.
   On considère la propriété suivante :
   $$\mathcal{P} : s = \sum_{k=0}^{i-1} t_k$$
   Initialisation : Juste au début du premier tour de boucle, $i = 0$. La somme apparaissant dans $\mathcal{P}$ s'écrit $\sum_{k=0}^{-1} t_k$: elle ne contient aucun terme et vaut 0.
   Par ailleurs, la valeur 0 a été affectée à la variable modifiable (référence) `s` au tout début de l'algorithme, donc $s = 0$. Ainsi, l'égalité $s = \sum_{k=0}^{-1} t_k$ est vérifiée et $\mathcal{P}$ est donc vraie au début du premier tour de boucle
   Conservation : Supposons que la propriété est vraie au début d'un tour de boucle quelconque. Nous allons montrer qu'elle l'est alors également à la fin de ce même tour.
   On note s et i les contenus des variables `s` et `i` en début de tour.
   On note s' et i' les contenus des variables `s` et `i` en fin de tour.
   Lors du tour de boucle, on effectue d'abord une opération d'affectation qui met à jour le contenu de `s` :
   $$s' = s + t_i$$
   Puis le compteur de boucle est mis à jour
   $$i' = i + 1$$
   En fin de tour, on a donc, en utilisant l'hypothèse de validité de la propriété en début de boucle :
   $$s' = s + t_i = \sum_{k=0}^{i-1} t_k + t_i = \sum_{k=0}^{i} t_k = \sum_{k=0}^{i'-1} t_k$$
   On a donc montré que la propriété $\mathcal{P}$ est également vraie en fin de tour.

Ainsi, à la fin du dernier tour de la boucle inconditionnelle `for`, la propriété sera toujours vraie et $i = n$ (premier indice pour lequel la condition de boucle devient fausse). La variable `s` contiendra donc la valeur suivante en sortie de boucle, avant d'être renvoyée comme valeur de sortie par la fonction :

$$s = \sum_{k=0}^{n-1} t_k$$

On a donc montré que l'algorithme renvoie bien la somme des valeurs du tableau comme annoncé dans la question 1.

Exercice 2 (Tri par insertion).

1. Écrire une fonction récursive OCaml nommée `insere`, qui insère une nouvelle valeur dans une liste d'entiers déjà triée par ordre croissant, en respectant cet ordre.
2. Prouver la terminaison de cette fonction.
3. Prouver la correction de cette fonction.
4. Écrire une fonction récursive OCaml `tri_insertion` qui implémente le tri par insertion.
5. Prouver la terminaison de cette fonction.
6. Prouver la correction de cette fonction.

Corrigé de l'exercice 2. *[Retour à l'énoncé]*

Dans toute la suite, *trié* signifiera implicitement *trié par ordre croissant*.

1. La fonction `insere` peut être implémentée récursivement en OCaml de la manière suivante :

```
# (* precondition: l est triée *)
let rec insere v l =
    match l with
    | []              -> [v] (* cas de base *)
    | t::q when v < t -> v::l
    | t::q            -> t :: (insere v q);;
val insere : 'a -> 'a list -> 'a list = <fun>
```

2. **Terminaison.** On utilise la technique du variant. La suite choisie doit être à valeurs entières. Comme le code est récursif, le variant va sûrement vérifier une relation de récurrence. On note $|l|$ la taille d'une liste l et on introduit le variant :

$$u_k = |l_k|$$

où k est le numéro de l'appel récursif et l_k la liste fournie en entrée de cet appel. De manière évidente, en regardant le code, on a :

$$u_{k+1} = |l_{k+1}| = |q_k|$$

où q_k est la queue de la liste l_k utilisée dans l'appel précédent. Ainsi, $|q_k| = |l_k| - 1$ et on en déduit que :

$$u_{k+1} = |l_k| - 1 = u_k - 1$$

Il s'agit d'une suite à valeurs entières, arithmétique de raison -1 . Comme la taille initiale $u_0 = |l_0| \geq 0$, cette suite va donc nécessairement passer en 0 pour un appel $k_0 = |l_0|$.

Pour ce k_0-ième appel, la taille de la liste donnée en entrée de la fonction est donc nulle et on a une liste vide : le cas de base sera alors activé, et la récursion s'arrêtera.

3. **Correction.** Il s'agit d'un code **récursif** manipulant des **variables immuables** : on tente une démonstration de correction par **récurrence** sur la taille n de la liste donnée en entrée.

$\mathcal{P}(n)$: Pour une liste `l` triée de taille n, `insere v l` retourne une liste triée de taille $n+1$ contenant v

Cas de base : Si $n = 0$, alors la liste donnée en entrée est la liste vide. [1] Le code retourne une liste contenant l'unique valeur v. Ainsi $\mathcal{P}(0)$ est vraie.

1. La liste vide est bien une liste triée :il n'y en effet rien à trier !

Hérédité : Soit $n > 0$ et supposons que $\mathcal{P}(n)$ est vraie. Nous allons essayer de démontrer $\mathcal{P}(n+1)$. On analyse donc le travail réalisé par la fonction lorsque la liste $l = [l_0, l_1, \ldots, l_n]$ donnée en entrée est de taille $n+1$. On rappelle que la fonction ne prend en entrée que des listes triées (précondition) donc, par hypothèse, $l_0 < l_1 < \ldots < l_n$.

— Si la valeur v à insérer est strictement inférieure à la tête $t = l_0$ de l, alors la fonction retourne la liste $[v, l_0, l_1, \ldots, l_n]$ de taille $n+2$, qui est bien triée car $v < l_0 < l_1 < \ldots < l_n$. $\mathcal{P}(n+1)$ est donc vraie.

— Sinon, la fonction appelle récursivement `insere v q`. Dans cet appel, $q = [l_1, \ldots, l_n]$ est la queue de l : elle est de taille n et est triée car toute sous-liste d'une liste triée est triée. Par hypothèse de récurrence, cet appel retourne une liste triée $[l_1, l_k, v\, l_{k+1}, \ldots, l_n]$ triée de taille $n+1$ contenant v. L'algorithme rajoute l_0 en tête de cette liste et renvoie $[l_0, l_1, l_k, v, l_{k+1}, \ldots, l_{n-1}]$. Cette liste est de taille $n+2$. De plus, l_0 est bien à sa place car $l_0 < v$ et $l_0 < l_1 < \ldots < l_{n-1}$ donc l_0 est inférieur à tous les éléments de cette liste : l_0 doit donc bien être positionné en tête, ce qui est fait par l'algorithme. Ainsi, l'algorithme retourne une liste triée de taille $n+2$ contenant v.

Par principe de récurrence, nous avons donc démontré que $\mathcal{P}(n)$ est vraie $\forall\, n \geq 0$, ce qui achève la preuve de correction de l'algorithme.

4. La fonction `tri_insertion` peut être implémentée récursivement en OCaml de la manière suivante :

```
# let rec tri_insertion l =
  match l with
  |[] -> []
  |t::q -> insere t (tri_insertion q);;
val tri_insertion : 'a list -> 'a list = <fun>
```

5. La preuve de terminaison est identique à celle de la fonction `insere` en prenant le même variant.

6. Là encore, il s'agit d'un code **récursif** manipulant des **variables immuables** : on tente une démonstration de correction par **récurrence** sur la taille n de la liste donnée en entrée.

$\mathcal{P}(n)$: Pour une liste `l` de taille n, `tri_insertion l` retourne une liste triée contenant les mêmes valeurs que `l`

Cas de base : Si $n = 0$, la liste donnée en entrée est la liste vide. L'algorithme implémenté retourne alors la liste vide qui est triée et contient les mêmes éléments que la liste d'entrée, c'est-à-dire... aucun ! $\mathcal{P}(0)$ est donc vraie.

Hérédité : Supposons que la propriété $\mathcal{P}(n)$ est vraie. On considère une liste l d'entrée de taille $n+1$: $l = [l_0, l_1, \ldots l_n]$. L'algorithme implémenté appelle récursivement la fonction `tri_insertion` avec comme entrée la queue de l, $q = [l_1, \ldots l_n]$, qui est de taille n : par hypothèse de récurrence, cet appel retourne cette liste triée : $q_{\text{triee}} = [l_{\sigma(1)}, \ldots, l_{\sigma(n)}]$ et σ est une permutation des indices. Ensuite, l'algorithme appelle la fonction `insere` pour insérer la valeur l_0 dans q_{triee}. Cet appel respecte la précondition car q_{triee} est triée et, d'après la question 3, le résultat de cet appel est une liste triée de taille $n+1$ contenant les éléments de q_{triee} et la valeur l_0, c'est-à-dire contenant tous les éléments de l. On a donc montrée que $\mathcal{P}(n+1)$ est vraie

Par principe de récurrence, nous avons donc démontré que $\mathcal{P}(n)$ est vraie $\forall\, n \geq 0$, ce qui achève la preuve de correction de l'algorithme.

Nous avons donc montré la **correction totale** de l'algorithme de tri par insertion récursif.

Exercice 3 (Multiplication russe).

1. Cours : réécrire l'algorithme de multiplication russe sous forme récursive **terminale** en OCaml

2. Prouver la terminaison de cet algorithme.

3. Démontrer que :

$$\forall x \in \mathbb{R}, \forall q \in \mathbb{N}^*, \left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = \left\lfloor \frac{x}{q} \right\rfloor$$

4. En déduire une formule explicite pour le variant choisi.

5. Combien d'appels récursifs sont nécessaires pour que l'algorithme termine ?

6. Qu'en déduire sur la complexité temporelle de cet algorithme ?

Corrigé de l'exercice 3. *[Retour à l'énoncé]*

1. Voici une implémentation de l'algorithme de multiplication russe sous forme récursive terminale en OCaml

```
# let mult_russe p0 q0 =
  let rec aux acc p q =
    match p with
    | 0 -> acc
    | _ -> aux (acc + q*(p mod 2)) (p/2) (q*2)
  in aux 0 p0 q0;;
val mult russe : int -> int -> int = <fun>
```

Cette implémentation utilise un accumulateur `acc` pour transmettre à travers la chaîne d'appels le travail de multiplication effectué par les appels récursifs, sans devoir garder en mémoire ces informations dans des blocs d'activation empilés. Cela permet de réutiliser le même bloc d'activation pour tous les appels et d'éviter ainsi un empilement excessif de blocs d'activation pouvant mener à un effondrement de pile (*stack overflow*). L'algorithme est en fait codé dans une fonction auxiliaire `aux` gérant cet accumulateur, cette fonction auxiliaire étant ensuite appelée en initialisant l'accumulateur à 0 pour le premier appel.

2. Pour prouver la terminaison, on utilise, comme toujours, la technique du variant. Comme l'algorithme est implémenté de manière récursive, la suite qui fera office de variant a de forte chance d'être naturellement définie par récurrence.

On introduit donc la suite $(p_k)_{\mathbb{N}}$:

$$\begin{cases} p_0 \\ p_{k+1} = \lfloor \frac{p_k}{2} \rfloor \end{cases}$$

k représente le numéro d'appel imbriqué.

Il s'agit exactement du même variant que celui utilisé pour la preuve de terminaison de l'algorithme d'exponentiation rapide. La preuve de terminaison est donc identique, et on prouve ainsi que l'algorithme de multiplication russe se termine : il existe un appel k_0 pour lequel le cas de base est atteint : $p_{k_0} = 0$. Bien sûr, k_0 dépend de la valeur d'entrée p_0 !

3. Nous allons donc démontrer la propriété suivante :

Propriété 1

$$\forall x \in \mathbb{R}, \forall q \in \mathbb{N}^*, \left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = \left\lfloor \frac{x}{q} \right\rfloor$$

Démonstration

Soit $q \geq 1$. Écrivons la division euclidienne de $\lfloor x \rfloor \in \mathbb{N}$ par q :

$$\exists a \in \mathbb{N} \text{ et } r \in \mathbb{N} \text{ tels que } \lfloor x \rfloor = a\,q + r \text{ et } 0 \leq r < q$$

On a donc :

$$\frac{\lfloor x \rfloor}{q} = \underbrace{a}_{\in \mathbb{N}} + \underbrace{\frac{r}{q}}_{0 \leq \frac{r}{q} < 1}$$

donc, par définition de la partie entière :

$$\left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = a$$

Démontrer la propriété revient donc à démontrer que :

$$a = \left\lfloor \frac{x}{q} \right\rfloor$$

Par définition de la partie entière de x :

$$\begin{aligned}
& \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 \\
\Leftrightarrow \quad & a\,q + r \leq x < a\,q + r + 1 && \text{en remplaçant } \lfloor x \rfloor \text{ par } aq + r \\
\Leftrightarrow \quad & a + \underbrace{\frac{r}{q}}_{\geq 0} \leq \frac{x}{q} < a + \underbrace{\frac{r+1}{q}}_{\leq 1} && \text{en divisant l'encadrement par } q > 0 \\
\Leftrightarrow \quad & a \leq a + \frac{r}{q} \leq \frac{x}{q} < a + \frac{r+1}{q} \leq a + 1 \\
\Rightarrow \quad & a \leq \frac{x}{q} < a + 1
\end{aligned}$$

Comme $a \in \mathbb{N}$, cela signifie exactement que :

$$\left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = a$$

et la propriété est démontrée.

4. Nous allons démontrer la formule explicite suivante pour la suite $(p_k)_{\mathbb{N}}$ par récurrence :

$$\forall k \in \mathbb{N},\ p_k = \left\lfloor \frac{p_0}{2^k} \right\rfloor$$

Cas de base : Pour $k = 0$, la formule est vraie car

$$\left\lfloor \frac{p_0}{2^0} \right\rfloor = \left\lfloor \frac{p_0}{1} \right\rfloor = \lfloor p_0 \rfloor = p_0$$

et, pour la dernière égalité, on a utilisé le fait que la valeur d'entrée de l'algorithme p_0 est un entier

Hérédité : Supposons la formule vraie au rang $k \geq 1$. La formule de récurrence définissant la suite nous donne :

$$p_{k+1} = \left\lfloor \frac{p_k}{2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{p_0}{2^k} \right\rfloor}{2} \right\rfloor$$

En appliquant la propriété démontrée en question 3 pour $x = \dfrac{p_0}{2^k} \in \mathbb{R}$ et $q = 2 \in \mathbb{N}^*$, on obtient :

$$p_{k+1} = \left\lfloor \frac{p_k}{2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{p_0}{2^k} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{\frac{p_0}{2^k}}{2} \right\rfloor = \left\lfloor \frac{p_0}{2^{k+1}} \right\rfloor$$

et on donc bien montré la propriété au rang $k + 1$. ■

5. Détermine le nombre exact de tours effectués revient à déterminer k_0, indice pour lequel la suite $(p_k)_{\mathbb{N}}$ devient nulle (stationnaire). Nous allons montrer rigoureusement que :

Propriété 2

Le nombre de tours de boucles effectués dans l'algorithme de la multiplication russe dépend de la première entrée p_0 et vaut :

$$k_0 = 1 + \lfloor \log_2(p_0) \rfloor$$

On peut sans restriction supposer que p_0 est positif. En effet, s'il ne l'est pas, soit q_0 est positif et on reporte le signe moins sur q_0. Soit q_0 est négatif, et, comme les deux signes moins se compensent dans la multiplication, on peut les ignorer.

Comme n'importe quel nombre entier positif, la valeur initiale p_0 peut être encadrée par deux puissances de 2 consécutives :

$$\exists m \in \mathbb{N},\ 2^m \leq p_0 < 2^{m+1}$$

En divisant l'encadrement par 2^m (ce qui est légal et ne change pas le sens des inégalités car $2^m > 0$), on en déduit que : $1 \leq \dfrac{p_0}{2^m} < 2$

Par définition de la partie entière, on en déduit que : $\left\lfloor \dfrac{p_0}{2^m} \right\rfloor = 1 \Leftrightarrow p_m = 1$

De même, en divisant cette fois l'encadrement par 2^{m+1}, on en déduit que : $\dfrac{1}{2} \leq \dfrac{p_0}{2^{m+1}} < 1$.

Par définition de la partie entière, on en déduit que : $\left\lfloor \dfrac{p_0}{2^{m+1}} \right\rfloor = 0 \Leftrightarrow p_{m+1} = 0$

Ainsi, $p_m = 1$ et $p_{m+1} = 0$.

$m+1$ est donc le premier indice pour lequel la suite $(p_k)_{\mathbb{N}}$ est stationnaire et atteint sa valeur limite 0 : c'est donc k_0 !

$$k_0 = m + 1$$

Revenons maintenant la manière dont nous avons défini m.

$$\begin{array}{lllll} & 2^m & \leq p_0 & < 2^{m+1} & \\ \Leftrightarrow & m\ln(2) & \leq \ln(p_0) & < (m+1)\ln(2) & \text{en prenant le logarithme, qui est une fonction strictement croissante} \\ \Leftrightarrow & m & \leq \log_2(p_0) & < m+1 & \text{en divisant l'encadrement par } \ln(2) > 0 \end{array}$$

et par définition de la partie entière, en sachant de plus que m est un entier par définition, on en déduit que :

$$m = \lfloor \log_2(p_0) \rfloor$$

En rassemblant l'ensemble des éléments démontrés, on en déduit que :

$$k_0 = 1 + m = 1 + \lfloor \log_2(p_0) \rfloor$$

Le nombre de tours de boucles effectués est donc égal à $1 + \lfloor \log_2(p_0) \rfloor$

6. Maintenant que nous savons que le nombre de tours de boucles est $1 + \lfloor \log_2(x_0) \rfloor$, on peut compter le nombre d'opérations élémentaires dans le pire des cas, qui correspond au cas où le test dans la boucle est toujours vrai :

$$3 \text{ affectations} + (1 + \lfloor \log_2(x_0) \rfloor) \times (2 \text{ tests} + 4 \text{ opérations})$$

Plus p_0 est un grand nombre, plus l'algorithme sera coûteux en nombre d'opérations. . On a même un modèle de croissance de cette complexité : la complexité temporelle évolue asymptotiquement de la même manière que $\log_2(p_0)$. On dit que l'algorithme a une **complexité temporelle logarithmique**.

Entraînement à la maison

Exercice 4 (Occurrences d'une valeur dans un tableau).
On considère un algorithme **séquentiel** nommé `nombre_occurrences` qui renvoie le nombre d'occurrences d'une valeur v dans un tableau d'entiers.

1. Écrire une fonction implémentant cet algorithme en C
2. Prouver la terminaison de l'algorithme
3. Prouver la correction de l'algorithme

Corrigé de l'exercice 4. *[Retour à l'énoncé]*

1. Cette fonction prend 3 entrées
   - un tableau d'entiers t
   - sa taille (nécessaire en C, mais pas en OCaml) n
   - une valeur v

   Elle renvoie en sortie un entier correspondant au nombre d'occurrences de v dans le tableau t.

```c
int nombre_occurrences(int *t, int n, int v)
{
  assert(n >= 0);
  assert(t != NULL);

  int i;
  int c = 0;
  for (i = 0; i < n; i = i + 1)
  {
    if (t[i] == v)
      c = c + 1;
  }

  return c;
}
```

2. **Terminaison.** La terminaison est évidente car l'algorithme utilise une **boucle inconditionnelle** `for`, dont le compteur `i` n'est jamais modifié à l'intérieur de la boucle.
3. **Correction.** Il s'agit d'un algorithme itératif : nous allons donc exhiber un **invariant de boucle** pour en prouver la correction. On note i et c les valeurs contenues dans les variables `i` et `c`. On note t_k l'élément d'indice k du tableau `t`.
   On considère l'invariant de boucle suivant :

$$\mathcal{P} : c = \text{Card } U_i$$

   où

$$U_i = \{k \in [\![0, i-1]\!], t_k = v\}$$

   et l'on a noté Card le cardinal d'un ensemble fini, c'est-à-dire le nombre d'éléments de cet ensemble.

   Initialisation : Au début de la première boucle, $i = 0$ et l'ensemble $\{k \in [\![0, -1]\!], t_k = v\}$ est l'ensemble vide. Son cardinal est donc égal à 0. Comme c est également nul, $\mathcal{P}$ est vraie.

   Conservation : Supposons maintenant que la propriété $\mathcal{P}$ est vrai au début du i-ième tour de boucle.

   On note i et c les valeurs contenues dans les variables `i` et `c` en début de tour.

   On note i' et c' les valeurs contenues dans les variables `i` et `c` en fin de tou.

   On a toujours $i' = i + 1$.

   Si $t_i = v$: dans ce cas, la variable `c` est incrémentée et, en fin de tour, $c' = c + 1$. Par ailleurs $U_{i+1} = U_i \cup \{i\}$ et donc Card U_{i+1} = Card $U_i + 1$. Or, en début de tour, on a supposé que la propriété était vérifiée donc Card $U_i = c$. En remplaçant Card U_i par c ci-dessus, on a donc :

$$\text{Card } U_{i+1} = c + 1 \Leftrightarrow \text{Card } U_{i'} = c'$$

Si $t_i \neq v$: dans ce cas, la variable `c` n'est pas incrémentée et, en fin de tour, $c' = c$. Par ailleurs, $U_{i+1} = U_i$ donc Card U_{i+1} = Card U_i. Or, en début de tour, on a supposé que la propriété était vérifiée donc Card $U_i = c$. On a donc :

$$\text{Card } U_{i+1} = c \Leftrightarrow \text{Card } U_{i'} = c'$$

Dans tous les cas, on a montré que si la propriété est vraie en début de tour, elle l'est aussi en fin de tour.

Ainsi, à la fin du dernier tour, $i = n$ et $\mathcal{P}$ est toujours vraie, ce qui signifie que la variable `c` contient la valeur

$$\text{Card } \{k \in [\![0, n-1]\!], t_k = v\}$$

ce qui correspond exactement au nombre d'occurrences de la valeur v dans le tableau.

Exercice 5 (Preuves de terminaison).
Montrez que les fonctions suivantes terminent. **Les arguments donnés en entrée de ces fonctions sont tous des entiers.**

```
let rec g a =
  if a < 0 then 1
  else if a mod 2 = 0 then g (a+1)
  else g (a-3)
```

```
let rec f a b =
  if a <= 0 || b <=0 then 1
  else if a mod 2 = 0 then f (a/3) (2*b)
  else f (3*a) (b/5)
```

En cas de panne d'inspiration, vous pouvez coder et introduire des affichages dans ces fonctions pour bien comprendre leur fonctionnement sur des exemples et vous persuader de leur terminaison.

Corrigé de l'exercice 5. *[Retour à l'énoncé]*

1. **Preuve de terminaison de la fonction g.** On introduit la suite $(a_k)_{\mathbb{N}}$ suivante, définie par récurrence :

$$\begin{array}{ll} a_0 = a & \text{valeur d'entrée donnée par l'utilisateur} \\ a_{k+1} = a_k + 1 & \text{cas où l'entrée est paire} \\ a_{k+1} = a_k - 3 & \text{cas où l'entrée est impaire} \end{array}$$

k représente le numéro de l'appel récursif dans la pile d'appels imbriqués.
Cette suite est bien à valeurs entières. Il nous reste à analyser ses variations pour prouver une éventuelle décroissance.
Ici, la suite $(a_k)_{\mathbb{N}}$ n'est pas monotone. Toutefois, on peut réécrire les relations de récurrence en écrivant $k = 2k' + 1$ si k est impair et $k = 2k'$ s'il est pair.

$$\begin{array}{ll} a_0 = a & \text{valeur d'entrée donnée par l'utilisateur} \\ a_{2k'+1} = a_{2k'} + 1 & \text{cas où l'entrée est paire} \\ a_{2k'+2} = a_{2k'+1} - 3 & \text{cas où l'entrée est impaire} \end{array}$$

En substituant $a_{2k'+1}$ dans la seconde relation :

$$\begin{array}{l} a_0 = a \\ a_{2k'+2} = a_{2k'} - 2 \end{array}$$

Par ailleurs, on peut réécrire la première relation en remplaçant k' par $k' + 1$:

$$a_{2k'+3} = a_{2k'+2} + 1$$

et ainsi, en substituant $a_{2k'+2}$ par $a_{2k'+1} - 3$ dans cette relation, on obtient :

$$\begin{array}{l} a_1 = a_0 - 3 \\ a_{2k'+3} = a_{2k'+1} - 2 \end{array}$$

On peut alors obtenir des relations de récurrences pour les deux suites extraites correspond respectivement aux indices pairs et impairs.
Pour $(v_{k'})_{k' \in \mathbb{N}} = (a_{2k'})_{k' \in \mathbb{N}}$:

$$\begin{array}{l} v_0 = a \\ v_{k'+1} = v_{k'} - 2 \end{array}$$

Pour $(w_{k'})_{k'\in\mathbb{N}} = (a_{2k'+1})_{k'\in\mathbb{N}}$:

$$w_1 = a - 3$$
$$w_{k'+1} = w_{k'} - 2$$

Ces deux suites sont bien entendu à valeurs entières. Par ailleurs, elle sont toutes les deux arithmétiques de raison -2 donc strictement décroissantes, et deviennent donc strictement négative au bout d'un nombre fini d'itérations. Elles correspondent aux valeurs successives prises par le paramètre d'entrée dans les appels imbriqués.

La suite $(v'_k)_{k'\in\mathbb{N}}$ sera le variant utilisé si l'entrée a fournie par l'utilisateur est paire.

La suite $(w'_k)_{k'\in\mathbb{N}}$ sera le variant utilisé si l'entrée a fournie par l'utilisateur est impaire.

Ainsi, quelque soit l'entrée fournie par l'utilisateur (paire ou impaire), on a exhibé un variant qui nous assure que le cas de base sera atteint. Ainsi, la fonction `g` termine dans tous les cas.

2. **Preuve de terminaison de la fonction `f`.** On introduit les valeurs a_k et b_k correspondant aux valeurs des deux arguments d'entrée pour l'appel récursif numéro k.

   On définit $u_k = a_k \times b_k$.

   Nous allons cette fois-ci raisonner par l'absurde. Supposons que l'algorithme ne termine pas. Alors le cas de base n'est jamais atteint et, d'après le cas de base tel qu'il est écrit dans la fonction `f`, cela signifie que les suites $(a_k)_\mathbb{N}$ et $(b_k)_\mathbb{N}$ sont infinies, i.e elles ont une infinité de termes.

   De plus, comme le cas de base n'est jamais atteint, on en déduit que a_k et b_k sont toujours positifs, pour n'importe quel appel récursif k.

   Par définition de u_k, on en déduit que la suite $(u_k)_\mathbb{N}$ est également infinie.

   Nous allons montrer que $(u_k)_\mathbb{N}$ est une suite à valeurs dans $\mathbb{N}$ strictement décroissante. Comme $(u_k)_\mathbb{N}$ est infinie, cela sera contradictoire avec le fait que $\mathbb{N}$ est un ensemble bien ordonné et viendra conclure notre raisonnement par l'absurde.

   $(u_k)_\mathbb{N}$ **est une suite à valeurs dans** $\mathbb{N}$ car son terme général est un produit d'entiers naturels.

   $(u_k)_\mathbb{N}$ **est strictement décroissante.**

$$u_k = a_k \times b_k$$
$$u_{k+1} = a_{k+1} \times b_{k+1}$$

   En lien avec l'algorithme implémenté par la fonction `f`, il faut distinguer deux cas :

   Si a_k **est pair :** $a_{k+1} = \left\lfloor \frac{a_k}{3} \right\rfloor$ et $b_{k+1} = 2b_k$ d'après le code OCaml de `f`. Donc :

$$u_{k+1} = \left\lfloor \frac{a_k}{3} \right\rfloor \times 2b_k$$

   Or $\left\lfloor \frac{a_k}{3} \right\rfloor \leq \frac{a_k}{3}$ et donc :

$$u_{k+1} \leq \frac{a_k}{3} \times 2b_k = \frac{2}{3} a_k\, b_k < a_k\, b_k = u_k$$

   Donc $(u_k)_\mathbb{N}$ est strictement décroissante.

   Si a_k **est impair :** $a_{k+1} = 3a_k$ et $b_{k+1} = \left\lfloor \frac{b_k}{5} \right\rfloor$ d'après le code OCaml de `f`. Donc :

$$u_{k+1} = 3a_k \times \left\lfloor \frac{b_k}{5} \right\rfloor$$

   Or $\left\lfloor \frac{b_k}{5} \right\rfloor \leq \frac{b_k}{5}$ et donc :

$$u_{k+1} \leq 3a_k \times \frac{b_k}{5} = \frac{3}{5} a_k\, b_k < a_k\, b_k = u_k$$

   Donc $(u_k)_\mathbb{N}$ est strictement décroissante.

   $(u_k)_\mathbb{N}$ est donc strictement décroissante dans tous les cas.

   Comme nous l'avons vu, $(u_k)_\mathbb{N}$ est une suite infini strictement décroissante à valeurs dans $\mathbb{N}$: cela ne peut exister, on a donc une contradiction.

   On en conclut donc que l'algorithme termine toujours.