Séquence 9 - TD1 - Nouveaux outils théoriques pour les preuves d'algorithmes

Exercice 1 (Propriétés de base.).

1. Soit $(X, <)$ un ensemble ordonné et soit $Y \subseteq X$. Montrer que Y possède au plus un plus petit élément (et donc au plus un plus grand élément), ce qui justifie le fait de parler DU plus petit élément (resp. plus grand élément) de Y .
2. Montrer que dans un ensemble totalement ordonné, tout élément minimal est le plus petit élément (et donc tout élément maximal est le plus grand élément).
3. Donner un exemple fini d'ensemble bien ordonné. Donner un exemple infini d'ensemble bien ordonné.
4. Donner un exemple d'ensemble bien ordonné qui n'est pas totalement ordonné.

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

1. Soient x_1 et x_2 deux plus petits éléments de Y , supposons $x_1 \neq x_2$. Alors par définition du plus petit élément on a à la fois $x_1 < x_2$ et $x_2 < x_1$ donc par transitivité $x_1 < x_1$ ce qui contredit l'anti-réflexivité de la relation d'ordre stricte. Donc $x_1 = x_2$ et donc Y admet au plus un plus petit élément.
2. Supposons x minimal, soit $y \in X$ distinct de x. Comme l'ordre est total, on a soit $x < y$ soit $y < x$. Mais comme x est minimal, on ne peut avoir $y < x$, et on a donc $x < y$. Ainsi x est bien le plus petit élément de X.
3. L'ensemble vide (qui est fini !) muni de l'ordre vide est bien ordonné. L'ensemble des lettres de l'alphabet muni de la relation d'ordre usuelle est bien ordonné. $\mathbb{N}$ (infini) muni de l'ordre usuel est bien ordonné.
4. $\mathbb{N} \times \mathbb{N}$ muni de l'ordre produit est bien ordonné mais non totalement ordonné

Exercice 2.
Dans chacun des cas suivants donner un exemple d'un ensemble ordonné ayant la propriété demandée :

1. un ensemble totalement ordonné infini ayant à la fois un plus petit élément et un plus grand élément,
2. un ensemble totalement ordonné infini ayant un plus grand élément mais pas de plus petit élément,
3. un ensemble infini muni d'un ordre qui n'est pas total et qui a un plus grand élément et un plus petit élément,
4. un ensemble infini muni d'un ordre qui admet des éléments minimaux mais qui n'a pas de plus petit élément,
5. un ensemble infini bien ordonné qui admet un plus grand élément,
6. un exemple d'ensemble totalement ordonné mais non bien ordonné. Pouvez-vous trouver un exemple fini ?

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

1. $[0,1]$ muni de l'ordre usuel est un ensemble totalement ordonné infini ayant à la fois un plus petit élément (0) et un plus grand élément (1)
2. $]0,1]$ muni de l'ordre usuel est un ensemble totalement ordonné infini ayant un plus grand élément (1) mais pas de plus petit élément (0 n'est pas dedans, et on peut s'approcher infiniment près de 0),
3. $\mathcal{P}(\mathbb{N})$ muni de l'inclusion est un ensemble infini muni d'un ordre qui n'est pas total et qui a un plus grand élément $\mathbb{N}$ et un plus petit élément $\emptyset$.
4. L'ensemble des parties **non vides** de $\mathbb{N}$ muni de l'inclusion (qui n'est pas un ordre total) admet des éléments minimaux (les singletons) mais n'a pas de plus petit élément.
5. $]0,1]$ muni de l'ordre usuel est un ensemble totalement ordonné infini ayant un plus grand élément (1)
6. $\mathbb{R}$ muni de l'ordre usuel un exemple d'ensemble totalement ordonné mais non bien ordonné. On ne peut pas trouver d'exemple fini : en effet on montre par récurrence sur la cardinalité que tout ensemble fini non vide totalement ordonné a un plus petit élément ; en particulier tout-sous ensemble non vide d'un ensemble fini totalement ordonné a un plus petit élément et donc tout ensemble fini est bien ordonné.

Exercice 3.
L'ordre de Sarkovski $\succ$ sur $\mathbb{N}^*$ est défini par :

$$3 \succ 5 \succ 7 \succ 9 \ldots \succ 2\times 3 \succ 2\times 5 \ldots\ldots \succ 2^3\times 3 \succ 2^3\times 5 \succ \ldots \succ 2^3 \succ 2^2 \succ 2 \succ 1$$

1. Comparer 1988, 1989 et 1990 pour cet ordre
2. L'ordre de Sarkovski est-il un ordre bien fondé sur N^* ?

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

On remarque que l'ordre de Sarkovski considère les nombres impairs comme plus grands que tous les nombres pairs, à l'exception notable de 1. Plus un nombre à de puissances de 2 dans sa décomposition en produit de facteurs premiers, plus il est petit.

1. 1989 est impair, donc plus grand que les deux autres, qui sont pairs. Par ailleurs, $1988 = 2\times 994$, tandis que $1990 = 2\times 995$. Comme 994 est encore pair, 1988 est plus petit que 1990. Au final :
$$1989 \succ 1990 \succ 1988$$
2. Cet ordre n'est pas bien fondé, car il existe au moins une suite infinie strictement décroissante : la suite des nombres impairs $(2n+1)_{n\in\mathbb{N}^*}$.

Exercice 4 (Terminaison de la fusion des listes).

1. Réécrire un code OCaml `merge` réalisant la fusion de deux listes triées par ordre croissant en une nouvelle liste triée.
2. Prouver sa terminaison

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

1. Voici une proposition de code pour la fonction `merge` (nous verrons une version plus intéressante que celle-ci sur le plan de complexité spatiale !)

```
# (* precondition: l1 et l2 sont triées *)
let rec merge l1 l2 =
  match (l1, l2) with
  | ([], []) -> []
  | ([], _) -> l2
  | (_, []) -> l1
  | (t1::q1, t2::q2) when (t1 < t2) -> t1::(merge q1 l2)
  | (t1::q1, t2::q2)                -> t2::(merge l1 q2);;
val merge : 'a list -> 'a list -> 'a list = <fun>
```

2. On note $\mathcal{A}$ l'ensemble des paramètres d'entrée de la fonction `merge` : il s'agit de l'ensemble des couples de listes.
On plonge $\mathcal{A}$ dans l'ensemble bien ordonné $(\mathbb{N} \times \mathbb{N}, \prec_{\text{lex}})$ en introduisant la fonction Φ :

$$\begin{array}{rccc} \phi : & \mathcal{A} & \to & (\mathbb{N} \times \mathbb{N}, \prec_{\text{lex}}) \\ & (\ell_1,\, \ell_2) & \mapsto & (|\ell_1|,\, |\ell_2|) \end{array}$$

et on a noté $|\ell|$ la taille de la liste ℓ, c'est-à-dire le nombre de ses éléments.
Toujours avec les notations du cours, le seul élément minimal de $(\mathbb{N}\times\mathbb{N}, \preceq_{\text{lex}})$ est $(0, 0)$, et c'est l'image associée à un unique élément $\mathcal{B} = \{(\texttt{[]},\, \texttt{[]})\}$.
On applique le théorème de terminaison :

Cas de base : pour $(\ell_1, \ell_2) = (\texttt{[]},\, \texttt{[]})$, l'algorithme termine (il renvoie la liste vide)

Induction : soit $(\ell_1, \ell_2) \in \mathcal{A}$ une entrée. On suppose que, pour toutes les entrées (ℓ'_1, ℓ'_2) telles que $\phi(\ell'_1, \ell'_2) \prec_{\text{lex}} \phi(\ell_1, \ell_2)$, l'appel `merge` ℓ'_1 ℓ'_2 termine. Montrons que l'appel `merge` ℓ_1 ℓ_2 termine.
Cet appel n'engendre au plus qu'un seul appel récursif :

- soit il s'agit de l'appel récursif `merge` t_1 ℓ_2, qui s'effectue sur le couple (t_1, ℓ_2) dont l'image par ϕ est telle que $\phi(t_1, \ell_2) \prec_{\text{lex}} \phi(\ell_1, \ell_2)$ car $|t_1| = |\ell_1| - 1 < |\ell_1|$. Par hypothèse d'induction, on en déduit que l'appel `merge` t_1 ℓ_2 termine et donc que `merge` ℓ_1 ℓ_2
- soit il s'agit de l'appel récursif `merge` ℓ_1 t_2, qui s'effectue sur le couple (ℓ_1, t_2) dont l'image par ϕ est telle que $\phi(\ell_1, t_2) \prec_{\text{lex}} \phi(\ell_1, \ell_2)$ car $|t_2| = |\ell_2| - 1 < |\ell_2|$. Par hypothèse d'induction, on en déduit que l'appel `merge` ℓ_1 t_2 termine et donc que `merge` ℓ_1 ℓ_2

Dans tous les cas, on a donc montré que `merge` ℓ_1 ℓ_2 termine.

Par principe d'induction, on en déduit donc que l'appel `merge` ℓ_1 ℓ_2 termine pour toutes les entrées $(\ell_1, \ell_2) \in \mathcal{A}$

On aurait tout aussi bien pu utiliser l'ordre produit ici.
Ici, le théorème de terminaison est un peu surpuissant. On peut encore s'en sortir en prenant un variant égal à la somme de la taille des deux listes.

Exercice 5 (Fonction d'Ackermann).
Prouver la terminaison de la fonction d'Ackermann $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$: définie par :

$$\begin{cases} A(0, m) & = m + 1 \\ A(n, 0) & = A(n - 1, 1) \\ A(n, m) & = A(n - 1, A(n, m - 1)) \end{cases}$$

```
let ackermann n p = match (n,p) with
 | (0,_)-> p+1
 | (_,0)-> ackermann (n-1) 1
 |_ -> ackermann (n-1) (ackermann n (p-1));;
```

On pose $E = \mathbb{N}\times\mathbb{N}$, muni de l'ordre lexicographique que nous noterons $\preceq$, qui forme un ensemble bien ordonné. Bien sûr, E admet un seul élément minimal $(0,0)$ donc $\mathcal{B} = \{(0,0)\}$
$\mathcal{A} = E = \mathbb{N} \times \mathbb{N}$ car les arguments de la fonction `ackermann` peuvent varier dans E tout entier.
On pose ϕ égale à la fonction identité sur l'ensemble E.

Nous allons montrer par induction la propriété suivante :

$\mathcal{P}(n,p) : \forall (n,p) \in \mathbb{N}\times\mathbb{N}$, `ackermann n p` se termine et renvoie une valeur entière positive

Cas de base : Le seul élément minimal de $\mathbb{N}\times\mathbb{N}$ est $(0,0)$, et `ackermann 0 0` se termine et renvoie une valeur entière égale à 1, d'après l'algorithme.

Induction : Soit $x = (n,p) \in \mathbb{N} \times \mathbb{N}$ un couple d'entrée de l'algorithme. On fait l'hypothèse d'induction : on suppose que, $\forall y = (n',p') \prec x = (n,p)$ (au sens lexicographique toujours), l'algorithme `ackermann n' p'` se termine et renvoie un résultat dans $\mathbb{N}$.

On cherche à montrer que l'appel `ackermann n p` termine et renvoie un résultat entier. D'après l'algorithme proposé, l'appel `ackermann n p` fait :

- 0 appel récursif à `ackermann` si $n = 0$, donc l'appel termine et la valeur renvoyée est $p+1$ dans ce cas, qui est bien une valeur entière positive
- 1 appel récursif à `ackermann` si $n \geq 1$, et $p = 0$, cet appel est `ackermann (n-1) 1`, et on a bien $(n-1,1) \in \mathbb{N} \times \mathbb{N}$ et $(n-1,1) \prec (n,p)$, donc, par hypothèse d'induction , cet appel termine et renvoie une valeur entière positive ;
- 2 appels récursifs `ackermann` dans tous les autres cas (c'est-à-dire $n \geq 1$ et $p \geq 1$) :
  - le premier appel est `ackermann n (p-1)` : comme $(n,p-1) \in \mathbb{N} \times \mathbb{N}$ et $(n,p-1) \prec (n,p)$, cet appel se termine et renvoie bien une valeur entière positive, qui peut donc bien servir comme deuxième paramètre d'entrée de l'appel récursif qui suit ;
  - pour le deuxième appel imbriqué, le deuxième paramètre est bien un entier naturel positif car on a montré par hypothèse d'induction que l'appel `ackermann n (p-1)` se termine et renvoie une valeur entière positive que l'on notera x.
    Le deuxième appel peut donc s'écrire `ackermann (n-1) x`, et on a bien $(n-1,x) \in \mathbb{N}\times\mathbb{N}$, et $(n-1,x) \prec (n,p)$ avec l'ordre lexicographique, quelque soit la valeur $x \in \mathbb{N}$. Par hypothèse d'induction à nouveau, `ackermann (n-1) x` termine et renvoie une valeur entière positive.

Dans tous les cas, on a montré que, si `ackermann x y` termine pour tout couple d'entrées $(x,y) \prec (n,p)$, alors `ackermann n p` termine et renvoie une valeur entière positive.

D'après le théorème de terminaison, `ackermann` termine et renvoie une valeur entière positive pour toutes les entrées possibles.

Exercice 6 (Coefficients binomiaux).

1. Écrire en OCaml une fonction `comb` qui calcule un coefficient binomial $\binom{n}{p}$, n et p étant donnés.
2. Prouver la terminaison de cette fonction
3. Pour les plus rapides : redémontrer la formule utilisée de deux manières différentes : par le calcul et par une stratégie de dénombrement.
4. A la maison, pour retravailler OCaml : écrire une fonction itérative `triangle_pascal` qui retourne les lignes 0 à n du triangle de Pascal sous forme d'un tableau de tableaux d'entiers :

```
# triangle_pascal 3;;
- : int array array = [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]|]
```

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

1. Voici une proposition de code pour la fonction `comb` (non terminal ici, elle peut être rendue terminale) Considérons la fonction :

```
let rec comb (n,p) = match (n,p) with
 | (n,0)-> 1
 | (n,p) -> if p>n then 0 else comb((n-1),(p-1)) + comb((n-1),p);;
```

2. Montrons que la fonction `comb` se termine pour tous les paramètres d'entrée possibles.
On pose $E = \mathbb{N} \times \mathbb{N}$, muni de l'ordre lexicographique que nous noterons $\preceq$, $(E, \preceq)$ formant un ensemble bien ordonné.
Bien sûr, E admet un seul élément minimal $(0,0)$ donc $\mathcal{B} = \{(0,0)\}$
$\mathcal{A} = E$ car les arguments de la fonction `comb` peuvent varier dans E tout entier.
On pose ϕ égale à la fonction identité sur l'ensemble E.

   Cas de base : Le seul élément minimal de $\mathbb{N}\times\mathbb{N}$ est $(0,0)$, et `comb`00 termine d'après l'algorithme implémenté.

   Induction : Soit $x = (n,p) \in \mathbb{N}\times\mathbb{N}$. On veut prouver que l'appel `comb x`, c'est-à-dire `comb n p`, termine. L'algorithme proposé fait :

   — 0 appel récursif à `comb` si $p = 0$;

   — 0 appel récursif à `comb` si $p > n$, et en particulier si $n = 0$ et $p \geq 1$;

   — 2 appels récursifs à `comb` dans tous les autres cas (c'est-à-dire $1 \leq p \leq n$), et avec les arguments $(n-1, p-1)$ et $(n-1, p)$, qui sont bien dans $\mathbb{N} \times \mathbb{N}$, et strictement inférieurs à (n,p) au sens de l'ordre lexicographique.

   Selon le théorème de terminaison, `comb n p` termine pour tout couple d'entrée $(n,p) \in \mathbb{N} \times \mathbb{N}$, c'est-à-dire que `comb` termine toujours.

Remarque.

— On a utilisé le théorème de terminaison en prenant $(E, \preceq) = \mathbb{N} \times \mathbb{N}$ muni de l'ordre lexicographique et ϕ égale à la fonction identité, mais on aurait pu aussi prendre $(E, \preceq) = (\mathbb{N}, \leq)$, et $\phi(n, p) = n + p$, ce qui conduisait à une récurrence classique sur $n+p$ (ou une formulation sous la forme de la technique du variant)

— Le lecteur aura reconnu une simple écriture, fort maladroite, du calcul des coefficients du binôme. La complexité en est scandaleuse, et il proposera diverses versions bien meilleures de ce calcul.

3. Démonstration par le calcul :

$$\begin{pmatrix} n-1 \\ p \end{pmatrix} + \begin{pmatrix} n-1 \\ p-1 \end{pmatrix} = \frac{(n-1)!}{p!(n-1-p)!} + \frac{(n-1)!}{(p-1)!(\cancel{n-1} - (\cancel{p-1}))!}$$

$$= \frac{(n-1)!}{p!(n-1-p)!} + \frac{(n-1)!}{(p-1)!(n-p)!} = \frac{(n-1)!}{p(p-1)!(n-1-p)!} + \frac{(n-1)!}{(p-1)!(n-1-p)!(n-p)}$$

$$= \frac{(n-1)!}{(p-1)!(n-1-p)!} \times \left(\frac{1}{p} + \frac{1}{n-p}\right) = \frac{(n-1)!}{(p-1)!(n-1-p)!} \times \frac{n\cancel{-p+p}}{p(n-p)}$$

$$= \frac{(n-1)!}{(p-1)!(n-1-p)!} \times \frac{n}{p(n-p)}$$

$$= \frac{n \times (n-1)!}{p \times (p-1)!(n-1-p)! \times (n-p)} = \frac{n!}{p!(n-p)!} = \begin{pmatrix} n \\ p \end{pmatrix}$$

Démonstration par une méthode de dénombrement : $\begin{pmatrix} n \\ p \end{pmatrix}$ représente le nombre de manières de piocher p éléments dans un ensemble à n éléments. Notons $E = \{x_1; x_2; \ldots; x_n\}$ cet ensemble à n éléments.
On peut découper l'ensemble des parties de E à p éléments en deux catégories disjointes

Ceux qui ne contiennent pas x_n **:** il y en a $\begin{pmatrix} n-1 \\ p \end{pmatrix}$ car cela revient à choisir les p éléments de la partie dans $E - \{x_n\}$, qui est un ensemble à $n-1$ éléments

Ceux qui contiennent x_n **:** dans ce cas, l'un des éléments de la partie est fixé et il reste à choisir les $p-1$ éléments restants dans l'ensemble $E - \{x_n\}$ à $n-1$ éléments. On a donc $\begin{pmatrix} n-1 \\ p-1 \end{pmatrix}$ parties dans cette catégorie.

Au final, les deux catégories formant une partition de l'ensemble des parties de E à p éléments, le nombre total de parties de E à p éléments est bien la somme des parties appartenant à chacune des catégories :

$$\begin{pmatrix} n \\ p \end{pmatrix} = \begin{pmatrix} n-1 \\ p \end{pmatrix} + \begin{pmatrix} n-1 \\ p-1 \end{pmatrix}$$

4. Voici une proposition de code pour la fonction `triangle_pascal`

```
# let triangle_pascal n =
  let t = Array.make (n+1) [||] in (*tableau de tableaux*)
  for i = 0 to n do
    t.(i) <- Array.make (i+1) 0;
    (t.(i)).(0) <- 1;
    for p = 1 to (i-1) do
      (t.(i)).(p) <- (t.(i-1)).(p-1) +  (t.(i-1)).(p)
    done;
    (t.(i)).(i) <- 1;
  done;
  t;;
val triangle_pascal : int -> int array array = <fun>
# triangle_pascal 3;;
- : int array array = [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]|]
# triangle_pascal 5;;
- : int array array =
[|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; 1|];
  [|1; 5; 10; 10; 5; 1|]|]
```

Exercice 7 (Tri par épuisement des inversions).
On propose l'algorithme de tri suivant pour trier un tableau t de n entiers : tant que le tableau n'est pas trié, on sélectionne au hasard deux indices tels que $i < j$ et $t[i] > t[j]$ et on échange les valeurs de $t[i]$ et $t[j]$.
Prouver que cet algorithme étonnant termine !

Corrigé de l'exercice 6. *[Retour à l'énoncé]*

On utilise le variant suivant : pour chaque appel récursif k, on note u_k le nombre d'inversions restantes dans le tableau. Montrons que l'algorithme termine. On procède par l'absurde : supposons l'algorithme ne termine pas. Alors la suite de valeurs u_k est infinie car le nombre d'appels récursifs est infini. De plus, cette suite $(u_k)_{k\in\mathbb{N}}$ est bien à valeurs dans $\mathbb{N}$, et elle est strictement décroissante à chaque appel récursif par définition de l'algorithme. Cela est impossible car $(\mathbb{N}, \leq)$ est bien ordonné. On en déduit que l'algorithme termine.

Exercice 8 (Fonction de Sudan).
La fonction de Sudan est la fonction $F : \mathbb{N}^3 \to \mathbb{N}$ définie par :

$$\begin{cases} F(0,x,y) & = x+y \\ F(n,x,0) & = x \\ F(n+1,x,y+1) & = F(n, F(n+1,x,y), F(n+1,x,y)+y+1) \end{cases}$$

1. Écrire un code OCaml implémentant cette fonction
2. Prouver sa terminaison

Corrigé de l'exercice 7. *[Retour à l'énoncé]*

1. Voici une proposition de code pour la fonction `sudan`, qui fait attention à ne pas faire deux fois le même appel récursif en liant le résultat de cet appel à l'identificateur `a` :

```
# let rec sudan n x y =
  Printf.printf "%d %d %d\n" n x y;
  match (n, x, y) with
  | (0, _, _) -> x+y
  | (_, _, 0) -> x
  | _ -> let a =  sudan n x (y-1)  in sudan (n-1) a (a + y);;
val sudan : int -> int -> int -> int = <fun>
# sudan 1 2 1;;
1 2 1
1 2 0
0 2 3
- : int = 5
# sudan 2 2 1;;
2 2 1
2 2 0
1 2 3
1 2 2
1 2 1
1 2 0
0 2 3
0 5 7
0 12 15
- : int = 27
```

2. Pour cet exemple, la présence d'appels récursifs imbriqués nécessite l'écriture d'une preuve d'induction rigoureuse.

On se place dans l'ensemble ordonné $E = (\mathbb{N}^3, \preceq)$ où $\preceq$ désigne l'ordre lexicographique sur $\mathbb{N}^3$ induit par l'ordre usuel $\leq$ sur $\mathbb{N}$. D'après un résultat du cours, $(\mathbb{N}^3, \preceq)$ est bien ordonné.

- L'ensemble des paramètres d'entrée possibles de la fonction de Sudan évoluent dans $\mathbb{N}^3$ tout entier donc, avec les notations du cours, $\mathcal{A} = \mathbb{N}^3$.
- $\mathbb{N}^3$ a un seul élément minimal qui est $(0, 0, 0)$ donc $\mathcal{B} = \{(0, 0, 0)\}$
- Ici, la fonction ϕ est la fonction identité.

Nous allons montrer la propriété suivante, paramétrée par $e = (n, x, y) \in E = \mathbb{N}^3$

$\mathcal{P}(e) : \forall e = (n, x, y) \in \mathbb{N}^3$, l'appel `sudan n x y` se termine et renvoie une valeur entière positive.

Cas de base : Le seul élément minimal de E est $(0, 0, 0)$ et, d'après l'algorithme, `sudan 0 0 0` se termine et renvoie la valeur 0 qui est entière positive. La propriété $\mathcal{P}$ est donc vraie pour tous les éléments minimaux de E (il n'y en a qu'un :-))

Induction : On fixe $e = (n, x, y)$ et on suppose que, pour tout $e' = (n', x', y')$ tels que $e' = (n', x', y') \prec e = (n, x, y)$, la propriété $\mathcal{P}(e')$ est vraie, c'est-à-dire que `sudan n' x' y'` termine et renvoie une valeur entière positive.
L'appel `sudan n x y` effectue :

- 0 appel récursif si $y = 0$, et donc se termine dans ce cas, et renvoie la valeur x d'après l'algorithme, qui est bien une valeur entière positive
- 2 appels récursifs sinon :
  - le premier appel récursif `sudan n x (y-1)` s'effectue sur le triplet d'entrée $e' = (n, y, y - 1)$ qui est strictement inférieur à $e = (n, x, y)$ au sens lexicographique. Par hypothèse d'induction, cet appel se termine et renvoie une valeur entière positive, nommée `a` dans le code.
  - le second appel récursif `sudan (n-1) a (a+y)` a du sens car on vient de voir que `a` est bien associé à une valeur entière positive. De plus, cet appel se fait avec un triplet d'entrée $e' = (n-1, a, a+y)$ qui strictement inférieur à $e = (n, x, y)$ au sens lexicographique. Par hypothèse d'induction, il termine et renvoie une valeur entière positive.

Ainsi, on a bien montré que, dans tous les cas, sous hypothèse d'induction, $\mathcal{P}(e)$ est vraie.

Par le théorème du principe d'induction, on a donc montré que $\mathcal{P}(e)$ est vraie $\forall a \in \mathbb{N}^3$, ce qui signifie que l'algorithme de Sudan termine pour tout triplet de valeurs données en entrée, et qu'il renvoie toujours une valeur entière positive.

Remarque : la fonction de Sudan permet d'atteindre très vite des valeurs extrêmement grandes très longues à calculer, ce qui peut faire douter de sa terminaison si l'on se limite uniquement à une approche expérimentale.

Exercice 9 (Fonction f_{91} de McCarthy).
La fonction $f_{91} : \mathbb{Z} \to \mathbb{Z}$ de McCarthy est définie de la manière suivante :

$$\begin{cases} f_{91}(n) = f(f(n+11)) & \text{si } n \leq 100 \\ f_{91}(n) = n - 10 & \text{sinon} \end{cases}$$

Jouer avec la fonction pour comprendre son fonctionnement, donner une expression explicite de cette fonction et en déduire sa terminaison.

Corrigé de l'exercice 8. *[Retour à l'énoncé]*

```
# let rec mccarthy n =
  if (n > 100) then n - 10
  else
    mccarthy ( mccarthy (n+11) );;
val mccarthy : int -> int = <fun>
# mccarthy 3;;
- : int = 91
# mccarthy 11;;
- : int = 91
# mccarthy 50;;
- : int = 91
# mccarthy 90;;
- : int = 91
# mccarthy 91;;
- : int = 91
# mccarthy 95;;
- : int = 91
# mccarthy 100;;
- : int = 91
# mccarthy 101;;
- : int = 91
# mccarthy 102;;
- : int = 92
# mccarthy 103;;
- : int = 93
# mccarthy (-5);;
- : int = 91
```

Nous allons démontrer la propriété suivante $\forall x \in \mathbb{Z}$:

$$\mathcal{P}(x) : \ f(x) \text{ termine et } f(x) = \begin{cases} x - 10 & \text{si } x > 100 \\ 91 & \text{sinon} \end{cases} \text{ est à valeurs entières positives}$$

Pour les entrées $n > 100$, l'algorithme de McCarthy termine évidemment et renvoie une valeur entière positive égale à $n - 10$, la propriété est évidemment vraie.

Pour les entrées $n \in]-\infty; 100]\!]$, c'est beaucoup plus subtil. On se place dans l'ensemble suivant : $E = (]-\infty; 100]\!], \geq)$ Attention ici, on utilise l'ordre $\geq$. L'ensemble $(]-\infty; 100]\!] \subset \mathbb{Z}, \geq)$ est bien ordonné pour cet ordre inversé et il y a un seul élément minimal dans $]-\infty; 100]\!]$ pour cet ordre qui est le nombre 100.
On va faire une preuve par induction, en prenant en compte le fait que la relation d'ordre utilisée est la relation d'ordre usuelle sur $\mathbb{Z}$ mais inversée.

Cas de base : si $x = 100$, $\mathcal{P}(x)$ est vraie car :

$$f_{91}(100) = f_{91}(f_{91}(100+11)) = f_{91}(f_{91}(111)) = f_{91}(111-10) = f_{91}(101) = 101-10 = 91$$

Induction : Soit $x < 100$. On suppose $\mathcal{P}(y)$ vraie pour tout $y > x$ (attention à la relation d'ordre utilisée, qui est inversée !)

Si $90 \leq x < 100$ **:** alors $x + 11 > 100$ et $f_{91}(x + 11) = x + 11 - 10 = x + 1$. Ainsi :

$$f_{91}(x) = f_{91}(f_{91}(x + 11)) = f_{91}(x + 11 - 10) = f_{91}(x + 1) = 91\,,$$

et la dernière égalité a été obtenue par hypothèse d'induction car $x + 1 > x$.

Si $x < 90$ **:** alors $x + 11 < 100$ donc :

$$f_{91}(x) = f_{91}(f_{91}(x + 11)) = f_{91}(91) = 91$$

en utilisant l'hypothèse d'induction successivement sur $x + 11 > x$ et sur $91 > 90 > x$.

On a donc montré que $\mathcal{P}(x)$ est vraie dans tous les cas sous hypothèse d'induction.

On a donc montré la propriété pour toutes les entrées $n \in]-\infty; 100]\!] \subset \mathbb{Z}$ par induction, et pour les $n > 100$, on a dit au début que la propriété était évidente. La propriété est donc vraie $\forall x \in \mathbb{Z}$.