UCLouvain

École polytechnique de Louvain

LINFO 1123

Calculabilité, logique et complexité

Solutions des quiz

Y. Deville
C.H. Bertrand Van Ouytsel
V. Coppé
A. Gerniers
N. Golenvaux
M. Parmentier

1 Ensembles énumérables et non-énumérables

1. *L'ensemble des nombres impairs positifs est-il énumérable ?*

   Réponse : Vrai

   La fonction $f : \mathbb{N} \to E$ qui envoie n sur $2n + 1$ est une bijection, car :

   - c'est une surjection : si y est un nombre impair, alors $\frac{y-1}{2} \in \mathbb{N}$ et $f(\frac{y-1}{2}) = y$;
   - c'est une injection : si $f(n) = f(m)$, alors $2n + 1 = 2m + 1$ et donc $n = m$.

2. *L'ensemble des nombres premiers positifs est-il énumérable ?*

   Réponse : Vrai

   La fonction $f : \mathbb{N} \to E$ qui envoie n sur le n-ième plus petit nombre premier est une bijection (car l'ensemble des nombres premiers est infini, inclus dans $\mathbb{N}$ et totalement ordonné).

3. *L'ensemble des nombres entiers (positifs et négatifs) est-il énumérable ?*

   Réponse : Vrai

   La fonction $f : \mathbb{N} \to \mathbb{Z}$ qui envoie n sur $-\frac{n}{2}$ si n est pair et $\frac{n-1}{2}$ si n est impair est une bijection.

4. *L'ensemble des nombres rationnels est-il énumérable ?*

   Réponse : Vrai

   La fonction $f : \mathbb{N} \to \mathbb{Q}$ qui envoie n sur le n-ième nombre obtenu en parcourant le tableau ci-dessous (en suivant les diagonales descendantes de droite à gauche et en négligeant les répétitions) est une bijection.

	0	1	-1	2	-2	...
1	$\frac{0}{1}$	$\frac{1}{1}$	$\frac{-1}{1}$	$\frac{2}{1}$	$\frac{-2}{4}$	
2	$\frac{0}{2}$	$\frac{1}{2}$	$\frac{-1}{2}$	$\frac{2}{2}$	$\frac{-2}{4}$	
3	$\frac{0}{3}$	$\frac{1}{3}$	$\frac{-1}{3}$	$\frac{2}{3}$	$\frac{-2}{4}$	
4	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{-1}{4}$	$\frac{2}{4}$	$\frac{-2}{4}$	
⋮						

5. *L'ensemble des nombres irrationnels compris entre* 0 *et* 1 *est-il énumérable ?*

   Réponse : Faux

   L'ensemble des nombres réels entre 0 et 1 n'est pas énumérable (voir cours). Or, $[0; 1] \cap \mathbb{Q}$ est énumérable car c'est un sous-ensemble infini de $\mathbb{Q}$ (qui est énumérable). Comme :

   $$[0; 1] = \big([0; 1] \cap \mathbb{Q}\big) \cup \big([0; 1] \setminus \mathbb{Q}\big)$$

   l'ensemble $[0; 1] \setminus \mathbb{Q}$ ne peut pas être énumérable.

En effet, supposons par l'absurde que $[0;1] \setminus \mathbb{Q}$ est énumérable, on a alors deux bijections :

$$f_1 : \mathbb{N} \to [0;1] \cap \mathbb{Q} \qquad \text{et} \qquad f_2 : \mathbb{N} \to [0;1] \setminus \mathbb{Q}$$

Alors la fonction $g : \mathbb{N} \to [0;1]$ qui envoie n sur $f_1(\frac{n}{2})$ si n est pair et n sur $f_2(\frac{n-1}{2})$ si n est impair est une bijection, et donc $[0;1]$ est énumérable, ce qui est absurde.

6. *L'ensemble des fonctions de $\mathbb{N}$ dans $\{0,1\}$ est-il énumérable ?*

Réponse : Faux

La démonstration se fait avec l'aide de la méthode de la diagonalisation de Cantor vue en classe.

On suppose $F = \{f : \mathbb{N} \to \{0,1\}\}$ énumérable. Il existe donc une énumération des éléments de F : $f_0, f_1, \ldots, f_k, \ldots$. On peut représenter une fonction $f_k \in F$ comme la suite $f_k(0), f_k(1), \ldots, f_k(k), \ldots$. On peut donc construire une table infinie :

	0	1	2	...	k	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...	$f_0(k)$	
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...	$f_1(k)$	
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...	$f_2(k)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
f_k	$f_k(0)$	$f_k(1)$	$f_k(2)$	...	$f_k(k)$	
$\vdots$						$\ddots$

Soit la fonction f constituée des éléments de la diagonale :

$$f = f_0(0), f_1(1), f_2(2), \ldots, f_k(k), \ldots$$

On construit la fonction :

$$f' = f'(0), f'(1), f'(2), \ldots, f'(k), \ldots \quad \text{où} \quad f'(i) = 1 - f_i(i)$$

La fonction f' est également une fonction de $\mathbb{N}$ dans $\{0,1\}$. Elle est donc dans l'énumération puisque par hypothèse, F est énumérable. Dès lors, il existe p tel que $f_p = f'$:

$$\begin{aligned} & f_p = f_p(0), f_p(1), f_p(2), \ldots, f_p(p), \ldots \\ = \quad & f' = f'(0), f'(1), f'(2), \ldots, f'(p), \ldots \end{aligned}$$

Contradiction car $f'(p) \neq f_p(p)$ par définition de f'. Donc, $f' \neq f_p$ ce qui implique que f' n'est pas dans l'énumération. Conclusion, F n'est pas énumérable.

7. *Est-il vrai que l'ensemble des mots de longueur finie d'un alphabet énumérable est lui-même énumérable ?*

Réponse : Vrai

Intuitivement :

L'ensemble des mots de longueur finie est l'union de l'ensemble des mots de longueur 0, de longueur 1, de longueur 2, etc. L'ensemble des mots d'une certaine longueur n est énumérable, car on peut le représenter sous forme d'un tableau à n dimensions, que l'on parcoure en diagonale pour obtenir une bijection avec $\mathbb{N}$:

- Pour $n = 0$, “” (mot vide).
- Pour $n = 1$:

$$\begin{array}{ccccc} \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \dots \end{array}$$

- Pour $n = 2$:

$$\begin{array}{ccccc} \mathrm{aa} & \mathrm{ab} & \mathrm{ac} & \mathrm{ad} & \dots \\ \mathrm{ba} & \mathrm{bb} & \mathrm{bc} & \mathrm{bd} & \dots \\ \mathrm{ca} & \mathrm{cb} & \mathrm{cc} & \mathrm{cd} & \dots \\ \vdots & & & & \end{array}$$

- Etc.

Grâce à ça, on peut représenter l'ensemble des mots de longueur finie par un tableau, où chaque ligne représente l'ensemble des mots d'une certaine longueur :

$$\begin{array}{cccccccc} \text{“”} & & & & & & \\ \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \dots \\ \mathrm{aa} & \mathrm{ab} & \mathrm{ba} & \mathrm{ac} & \mathrm{bb} & \mathrm{ca} & \dots \\ \mathrm{aaa} & \mathrm{aab} & \mathrm{aba} & \mathrm{baa} & \mathrm{aac} & \mathrm{abb} & \dots \\ \vdots & & & & & & \end{array}$$

En parcourant ce tableau en diagonale, on obtient bien une bijection avec $\mathbb{N}$.

Preuve formelle :

Justifions d'abord par récurrence que l'ensemble des mots M_n d'une longueur fixée n d'un alphabet énumérable est lui-même énumérable.

- Pour $n = 0$, M_n ne contient qu'un élément (le mot vide) et est donc évidemment énumérable.
- Supposons que M_n soit énumérable pour un certain n, montrons qu'il en est de même pour M_{n+1}. Puisque l'alphabet $\mathcal{A}$ est énumérable, il existe une bijection $g : \mathbb{N} \to \mathcal{A}$. Puisque l'alphabet M_n est énumérable, il existe une bijection $f_n : \mathbb{N} \to M_n$.
  Alors, la fonction $f_{n+1} : \mathbb{N} \to M_{n+1}$ qui envoie n sur le n-ième élément obtenu en parcourant le tableau ci-dessous (en suivant les diagonales descendantes de droite à gauche) est une bijection :

	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$\dots$
$g(0)$	$f_n(0) + g(0)$	$f_n(1) + g(0)$	$f_n(2) + g(0)$	$f_n(3) + g(0)$	
$g(1)$	$f_n(0) + g(1)$	$f_n(1) + g(1)$	$f_n(2) + g(1)$	$f_n(3) + g(1)$	
$g(2)$	$f_n(0) + g(2)$	$f_n(1) + g(2)$	$f_n(2) + g(2)$	$f_n(3) + g(2)$	
$g(3)$	$f_n(0) + g(3)$	$f_n(1) + g(3)$	$f_n(2) + g(3)$	$f_n(3) + g(3)$	
$\vdots$					

  Remarque. Dans ce contexte, le symbole “+” désigne la concaténation.

Justifions à présent que l'ensemble des mots $M = \bigcup_{n=0}^{\infty} M_n$ de longueur finie d'un alphabet énumérable est lui-même énumérable.

- Pour tout n, on a une bijection $f_n : \mathbb{N} \to M_n$.
- Alors, la fonction $h : \mathbb{N} \to M$ qui envoie n sur le n-ième élément obtenu en parcourant le tableau ci-dessous (en suivant les diagonales descendantes de droite à gauche) est une bijection

	0	1	2	3	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	
$\vdots$					

8. *Est-il vrai que l'ensemble des mots de longueur infinie d'un alphabet énumérable est lui-même énumérable ?*

   Réponse : Faux

   Contre-exemple : l'ensemble des nombres réels entre 0 et 1 peuvent être vus comme l'ensemble des mots de longueur potentiellement infinie réalisés à partir de l'alphabet $\mathcal{A} = \{0, 1\}$. Ce dernier ne peut donc pas être énumérable.

9. *Supposons avoir deux ensembles A et B avec la même cardinalité. Si A n'est pas énumérable, B peut-il être énumérable ?*

   Réponse : Faux

   Supposons par l'absurde que B est énumérable. Alors il existe une bijection $f : \mathbb{N} \to B$. Si A et B ont la même cardinalité, alors il existe une bijection $g : A \to B$. Soit h la fonction inverse de la fonction g (qui existe et est unique car g est une bijection). Alors la fonction $(f \circ h) : \mathbb{N} \to A$ est une bijection et donc A est énumérable. C'est absurde, donc B ne peut pas être énumérable.

10. *Les ensembles $]0, 1[$, $]-1, 1[$, $]-\frac{\pi}{2}, \frac{\pi}{2}[$ et $\mathbb{R}$ ont-ils tous la même cardinalité ?*

    Réponse : Vrai

    Les fonctions suivantes sont toutes les trois des bijections :

$$\begin{aligned} f :]0, 1[&\longrightarrow]-1, 1[\\ x &\longmapsto 2x - 1 \end{aligned}$$

$$\begin{aligned} g :]-1, 1[&\longrightarrow]-\tfrac{\pi}{2}, \tfrac{\pi}{2}[\\ x &\longmapsto \frac{x\pi}{2} \end{aligned}$$

$$\begin{aligned} h :]-\tfrac{\pi}{2}, \tfrac{\pi}{2}[&\longrightarrow \mathbb{R} \\ x &\longmapsto \tan(x) \end{aligned}$$

2 Fonctions calculables et ensembles récursifs

1. *La fonction prenant en arguments 6 nombres entiers compris entre 1 et 45 et produisant 1 s'ils correspondent à la combinaison gagnante pour le prochain Lotto belge et 0 sinon est-elle calculable ?*

   Réponse : Vrai

   Pour le justifier, on peut générer autant de programmes que de combinaisons possibles pour le Lotto. Pour chaque combinaison possible C, le programme sera :

```
def lotto(args):
  if args == C:
    return 1
  else:
    return 0
```

   Au moins un de ces programmes calcule la fonction décrite.

2. *L'ensemble des nombres premiers est-il récursif ?*

   Réponse : Vrai

   Voici un programme qui permet de le justifier. Celui-ci prend en entrée un nombre naturel n et renvoie 1 si n est premier, 0 sinon.

```
def is_prime(n):
  if n == 0 or n == 1:
    return 0
  else:
    for i in range(2, n):
      if n mod i == 0:
        return 0
  return 1
```

   Grâce à cette fonction, on peut savoir pour tout nombre n s'il est premier ou non, donc l'ensemble des nombres premiers est récursif.

3. *Est-il vrai qu'un ensemble X est récursif si et seulement si son complémentaire (dans $\mathbb{N}$), $\overline{X}$, est lui-même récursif ?*

   Réponse : Vrai

   Soit $P_X(n)$ le programme qui permet de justifier que X est récursif (il renvoie donc 1 si $n \in X$, et 0 dans le cas contraire). En utilisant ce programme, on peut construire un nouveau programme $P_{\overline{X}}(n)$ qui renvoie si oui ou non $n \in \overline{X}$:

$$P_{\overline{X}}(n) \equiv \texttt{return } 1 - P_X(n)$$

   Ce programme permet donc de justifier que $\overline{X}$ est récursif si X est récursif.

   Puisque le complémentaire de $\overline{X}$ est X, l'implication dans l'autre sens (X récursif si $\overline{X}$ récursif) découle de celle-ci.

4. *Est-il vrai qu'un ensemble X est récursif si et seulement si X est récursivement énumérable et $\overline{X}$ est récursivement énumérable ?*

   Réponse : Vrai

- Si X est récursif, alors il est certainement récusivement énumérable (le programme qui permet de justifier que X est récursif permet également de justifier que X est récursivement énumérable). Par la question précédente, on sait alors que le complémentaire de X est alors lui aussi récursif et donc récursivement énumérable.
- Pour ce qui est de l'implication dans l'autre sens : supposons que X et $\overline{X}$ sont récursivement énumérables. Soient les deux programmes $P_{X_{re}}(n)$ et $P_{\overline{X}_{re}}(n)$ qui correspondent. Deux réponses possibles : la réponse "moderne" et la réponse "classique".
  - La réponse "moderne" consiste à dire qu'il suffit de créer un programme P_X qui exécute en parallèle $P_{X_{re}}(n)$ et $P_{\overline{X}_{re}}(n)$ (multithreading). Aussi tôt que $P_{X_{re}}(n)$ ou $P_{\overline{X}_{re}}(n)$ se termine pour un certain $n \in \mathbb{N}$ (et nécessairement au moins un des deux doit se terminer), on peut décider si n appartient ou non à X.

  Si cette réponse moderne vous convient, fantastique (elle sera acceptée à l'examen) ! Mais dans le cas contraire, il vous faut la réponse "classique", qui explicite ce qu'on cache dérrière les mots tels que "multithreading" ou "exécuter en parallèle".
  - Pour la réponse "classique", introduisons la notion d'opérations élémentaires d'un programme. Par opérations élémentaires, on entend ici les étapes individuelles des instructions des programmes[1]. On construit alors le programme suivant. Pour tout $n \in \mathbb{N}$, ce programme produit 1 si $n \in X$ et 0 si $n \notin X$. X est donc bien récursif.

$P_X(n) \equiv$

```
k = 0
while True :
  do the k first elementary operations of P_{X_re}(n)
  if P_{X_re}(n) terminates :
    if P_{X_re}(n) == 1 :
      return 1
    else :
      return 0

  do the k first elementary operations of P_{X̄_re}(n)
  if P_{X̄_re}(n) terminates :
    if P_{X̄_re}(n) == 1 :
      return 0
    else :
      return 1

  k = k + 1
```

5. *Si deux ensembles X et Y sont récursifs, alors il est toujours vrai que $X \cup Y$, $X \cap Y$ et $\overline{X}$ sont aussi des ensembles récursifs.*

   Réponse : Vrai

   Si X et Y sont récursifs, on a deux programmes $P_X(n)$ et $P_Y(n)$ qui permettent repectivement de décider si un élément $n \in \mathbb{N}$ est dans X ou Y. Pour justifier le fait que $X \cup Y$, $X \cap Y$ et $\overline{X}$, il suffit

[1] Ces opérations élémentaires dépendent du formalisme que vous utilisez. Dans le cas du formalisme des machines de Turing, le modèle traditionnel de la calculabilité (dont nous parlerons en détail plus tard dans le cours), ces opérations élémentaires sont explicites. En attendant, pour nourrir votre intuition, dites-vous par exemple qu'une instruction telle que "`for i in range(10): k = k + 1`" correspond à 10 opérations élémentaires.

Remarque. Il est nécessaire de parler des opérations élémentaires des deux programmes $P_{X_{re}}$ et $P_{\overline{X}_{re}}$ et non des instructions de ces deux programmes parce qu'une unique instruction peut mener un programme à boucler indéfiniment. Alterner les instructions (et non les opérations élémentaires) de ces deux programmes pourrait donc par exemple mener à une situation où le programme construit entrerait dans une boucle infinie du programme $P_{\overline{X}_{re}}$ avant d'avoir atteint la dernière instruction du programme $P_{X_{re}}$.

de considérer les programmes suivants :

$$P_{X\cup Y}(n) \equiv \left[\begin{array}{l} \texttt{if } P_X(n) == 1 \texttt{ or } P_Y(n) == 1 : \\ \quad \texttt{return } 1 \\ \texttt{else:} \\ \quad \texttt{return } 0 \end{array}\right.$$

$$P_{X\cap Y}(n) \equiv \left[\begin{array}{l} \texttt{if } P_X(n) == 1 \texttt{ and } P_Y(n) == 1 : \\ \quad \texttt{return } 1 \\ \texttt{else:} \\ \quad \texttt{return } 0 \end{array}\right.$$

$$P_{\overline{X}}(n) \equiv \texttt{return } 1 - P_X(n)$$

6. *Si deux ensembles X et Y sont récursivement énumérables, alors il est toujours vrai que $X \cup Y$, $X \cap Y$ et $\overline{X}$ sont aussi des ensembles récursivement énumérables.*

   Réponse : Faux

   $X\cup Y$ et $X\cap Y$ seront bien toujours également récursivement énumérables, mais $\overline{X}$ pas nécessairement.

   Contre-exemple : $X = K = \{n \in \mathbb{N} \mid (n, n) \in HALT\}$. Comme K est récursivement énumérable mais n'est pas récursif, $\overline{K}$ ne peut pas être récursivement énumérable.

7. *Est-il vrai que la somme de deux fonctions calculables à une variable est toujours une fonction calculable ?*

   Remarque. *La somme de deux fonctions partielles n'est définie qu'aux points pour lesquels les deux fonctions de départ sont bien définies.*

   Réponse : Vrai

   Soient deux fonctions calculables à une variable f_1, f_2 et soient P_{f_1}, P_{f_2} deux programmes qui calculent ces fonctions. Alors la fonction $f = f_1 + f_2$ définie sur l'intersection des domaines de f_1 et f_2 est calculée par le programme suivant.

$$P_f(n) \equiv \texttt{return } P_{f_1}(n) + P_{f_2}(n)$$

8. *Est-il vrai qu'un ensemble X est récursivement énumérable si et seulement si il existe une fonction calculable partielle f telle que $X = \mathrm{dom}(f)$?*

   Réponse : Vrai

   - Démontrons d'abord que l'existence d'une fonction calculable partielle f telle que $X = \mathrm{dom}(f)$ est une condition suffisante. Soit P_f un programme qui permet de calculer la fonction f. Alors le programme suivant permet de justifier que X est récursivement énumérable.

$$P_{X_{re}}(n) \equiv \left[\begin{array}{l} P_f(n) \\ \texttt{return } 1 \end{array}\right.$$

   - À présent, démontrons que l'existence d'une fonction calculable partielle f telle que $X = \mathrm{dom}(f)$ est une condition nécessaire. Soit $P_{X_{re}}$ un programme qui permet de justifier que X est récursivement énumérable. Alors le programme suivant calcule une fonction f telle que

$X = \mathrm{dom}(f)$:

$$P_f(n) \equiv \left[\begin{array}{l} \texttt{if } P_{X_{re}}(n) == 1 : \\ \quad \texttt{return } n \\ \texttt{else:} \\ \quad \texttt{while True} : \\ \quad\quad \texttt{pass} \end{array}\right.$$

9. *Est-il vrai qu'un ensemble X est récursivement énumérable si et seulement si il existe un programme Python qui énumère les éléments de X ?*

   Réponse : Vrai

   - Démontrons d'abord que l'existence d'un tel programme est une condition suffisante. Soit P_{enum_X} un programme qui énumère les éléments de X. Alors le programme suivant permet de justifier que X est récursivement énumérable.

$$P_{X_{re}}(n) \equiv \left[\begin{array}{l} k = 0 \\ \texttt{while True} : \\ \quad \texttt{if } P_{enum_X}(k) == n : \\ \quad\quad \texttt{return } 1 \\ \quad k = k + 1 \end{array}\right.$$

   - À présent, démontrons que l'existence d'un tel programme est une condition nécessaire. Soit $P_{X_{re}}$ un programme qui permet de justifier que X est récursivement énumérable. Alors on peut créer un programme P_{enum_X} qui énumère les éléments de X : on lance en parallèle le programme $P_{X_{re}}$ pour chaque donnée k. $P_{enum_X}(n)$ renvoie ensuite la donnée k du n-ième thread qui termine.

     Plus formellement, on peut construire une table avec à la position i, j les j premières opérations élémentaires du programme $P_{X_{re}}(i)$. Cela permet d'exécuter les programmes en "parallèle" en suivant les diagonales descendantes de cette table :

	Opérations élémentaires du programme						
$P_{X_{re}}(0)$	•	•	•	•	•	•	$\cdots$
$P_{X_{re}}(1)$	•	•	•	•	•	•	$\cdots$
$P_{X_{re}}(2)$	•	•	•	•	•	•	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

     Ce qui donne :

$$P_{enum_X}(n) \equiv \left[\begin{array}{l} c = 0 \\ i = 0 \\ \texttt{while True} : \\ \quad j = 0 \\ \quad \texttt{while } j < i : \\ \quad\quad k = i - j \\ \quad\quad \texttt{do the k first elementary operations of } P_{X_{re}}(j) \\ \quad\quad \texttt{if } P_{X_{re}}(j) \texttt{ terminates and did not terminate with} \\ \quad\quad k - 1 \texttt{ elementary operations} : \\ \quad\quad\quad \texttt{if } P_{X_{re}}(j) == 1 : \\ \quad\quad\quad\quad \texttt{if } c == n : \\ \quad\quad\quad\quad\quad \texttt{return } j \\ \quad\quad\quad\quad c = c + 1 \\ \quad\quad j = j + 1 \\ \quad i = i + 1 \end{array}\right.$$

10. *Est-il vrai qu'un ensemble X est récursivement énumérable si et seulement si X est l'ensemble vide ou il existe une fonction calculable totale f telle que $X = image(f)$?*

Réponse : Vrai

- Démontrons d'abord qu'il s'agit d'une condition suffisante. Si X est l'ensemble vide, X est évidemment récursivement énumérable. Supposons que X est non vide. Soit une fonction calculable totale f telle que $X = \text{image}(f)$. Soit P_f un programme qui calcule cette fonction. Alors le programme suivant permet de justifier que X est récursivement énumérable :

$$P_{X_{re}}(n) \equiv \left[\begin{array}{l} k = 0 \\ \texttt{while True}: \\ \quad \texttt{if } P_f(k) == n: \\ \quad\quad \texttt{return } 1 \\ \quad k = k + 1 \end{array}\right.$$

- À présent, démontrons qu'il s'agit d'une condition nécessaire. Soit $P_{X_{re}}$ un programme qui permet de justifier que X est récursivement énumérable.
  - Si X est l'ensemble vide, alors X est l'ensemble vide.
  - Si X est non-vide et fini, alors $X = \{x_0, x_1, \ldots, x_{n-1}\}$ et on peut alors facilement trouver une fonction totale calculable telle que $X = \text{image}(f) : f(k) = x_{k\%n}$ (% est l'opérateur modulo).
  - Si X est non-vide et infini, alors le programme P_{enum_X} (voir exercice 9) calcule une fonction totale f telle que $X = \text{image}(f)$.

3 Réduction et théorème d'Hoare-Allison

Soit L un sous-ensemble récursif non trivial du langage Python qui ne permet de calculer que des fonctions totales.

1. *Est-il vrai que la fonction halt de L est calculable avec L ?*

   Réponse : Vrai

   Étant donné que L ne calcule que des fonctions totales, tous les programmes de L se terminent pour toutes les entrées. Le programme de L suivant calcule donc la fonction *halt* (qui est bien une fonction totale) :

   $$P_{HALT_L}(n, x) \equiv \texttt{return 1}$$

2. *Est-il vrai que la fonction halt de L est calculable avec Python ?*

   Réponse : Vrai

   Voici un programme Python qui calcule cette fonction :

   $$P_{HALT_L}(n, x) \equiv \texttt{return 1}$$

3. *Est-il vrai que la fonction interpréteur de L est calculable avec L ?*

   Réponse : Faux

   Par le théorème d'Hoare-Allison, puisque L est un formalisme de calculabilité qui ne permet de calculer que des fonctions totales, la fonction interpréteur de L n'est pas calculable avec L.

4. *Est-il vrai que la fonction interpréteur de L est calculable avec Python ?*

   Réponse : Vrai

   L'interpréteur de Python `PyPy` (qui est un interpréteur de Python écrit en Python) convient. Sinon, plus explicitement :

   $$P_{interpret_L}(n, x) \equiv \texttt{return compile}(n)[x]$$

   Remarque. `compile` est une fonction native de Python potentiellement très dangereuse ! Faites très attention si vous décidez de jouer avec. Le code du programme Python ci-dessus est volontairement syntaxiquement incorrect.

5. *Est-il vrai qu'il existe une fonction totale calculable qui n'est pas calculable avec L ?*

   Réponse : Vrai

   La fonction interpréteur de L (voir question 3 ci-dessus) est un exemple. En général, le théorème d'Hoare-Allison dit que si un langage ne calcule que des fonctions totales, alors il ne peut pas calculer toutes les fonctions totales.

6. *Existe-t-il un langage (éventuellement un sous-ensemble du langage Python) qui permet de calculer à la fois sa fonction halt et sa fonction interpréteur ?*

   Réponse : Oui, mais il est nécessairement trivial/dégénéré (par le théorème d'Haore-Allison).

   Exemple : le langage constitué d'une unique instruction : $L = \{\texttt{return 1}\}$. Ce langage ne permet de créer qu'un seul programme et ce programme calcule à la fois la fonction *halt* et la fonction interpréteur de ce langage.

7. Est-il vrai qu'un ensemble non récursif ne peut pas être récursivement énumérable ?

   Réponse : Faux

   K est un ensemble non récursif mais il est récursivement énumérable.

8. *Est-il vrai qu'un ensemble non récursif ne peut pas être co-récursivement énumérable ?*

   Réponse : Faux

   $\overline{K}$ est un ensemble non récursif mais il est co-récursivement énumérable.

9. *L'union de deux ensembles non récursifs est nécessairement non récursif.*

   Réponse : Faux

   K et $\overline{K}$ sont deux ensembles non récursif mais $K \cup \overline{K} = \mathbb{N}$ est récursif.

10. *L'intersection de deux ensembles non récursifs est nécessairement non récursif.*

    Réponse : Faux

    K et $\overline{K}$ sont deux ensembles non récursif mais $K \cap \overline{K} = \emptyset$ est récursif.

4 Théorème de Rice

En vous aidant du théorème de Rice, déterminez si les ensembles définis ci-dessous sont récursifs.

1. *L'ensemble des programmes qui calculent la fonction* $f : \mathbb{N} \to \mathbb{N}$ *telle que* $f(n) = 2n$.

   Réponse : Non-récursif

   - Soit A l'ensemble des programmes i qui calculent $\varphi_i = f(n) = 2n$. De tels programmes existent, donc $A \neq \emptyset$.
   - Alors tous les programmes j dans $\overline{A}$ calculent $\varphi_j \neq \varphi_i$. Il existe bien des programmes qui ne calculent pas $f(n)$, donc $A \neq \mathbb{N}$.
   - Par la contraposée de Rice, A n'est pas récursif.

2. *L'ensemble des programmes qui renvoient* 0 *pour au moins une entrée.*

   Réponse : Non-récursif

   - $A \neq \emptyset$ (e.g. $P_i(x) \equiv$ `return 0`) et $A \neq \mathbb{N}$ (e.g. $P_j(x) \equiv$ `return 42`)
   - $\forall i \in A,\ \exists x : \varphi_i(x) = 0$ et $\forall j \in \overline{A},\ \forall x : \varphi_j(x) \neq 0$

     $\implies \forall i \in A,\ \forall j \in \overline{A},\ \exists x : \varphi_i(x) \neq \varphi_j(x)$

     $\implies \forall i \in A,\ \forall j \in \overline{A} : \varphi_i \neq \varphi_j$
   - Par la contraposée de Rice, A n'est pas récursif

3. *L'ensemble des programmes qui terminent après moins de 1000 instructions pour l'entrée* 0.

   Réponse : Récursif

   On peut décider A en exécutant les 999 premières instructions de $P(0)$. Vérifions que Rice s'applique : comme $A \neq \emptyset$ et $A \neq \mathbb{N}$, par le théorème de Rice, $\exists i \in A$ et $\exists j \in \overline{A}$ tel que $\varphi_i = \varphi_j$.

   Par exemple :

   $P_i \equiv$ `return 42`

   $P_j \equiv$
   ```
   for k in range(1000):
     sleep(1)
   return 42
   ```

4. *L'ensemble des programmes qui terminent pour au moins deux entrées différentes.*

   Réponse : Non-récursif

   Soit $A = \{i \mid \exists x, y : x \neq y \wedge \varphi_i(x) \neq \perp \wedge \varphi_i(y) \neq \perp\}$.

   Remarque. $\wedge$ désigne le "et" logique, $\vee$ le "ou" logique et $\Rightarrow$ l'implication.

   - $A \neq \emptyset$ (e.g. $P_i(x) \equiv$ `return 0`) et $A \neq \mathbb{N}$ (e.g. $P_j(x) \equiv$ `while True: pass`)
   - $\forall i \in A, \exists x, y : x \neq y \wedge \varphi_i(x) \neq \perp \wedge \varphi_i(y) \neq \perp$

     $\forall j \in \overline{A}, \forall x, y : x \neq y \Rightarrow (\varphi_j(x) = \perp \vee \varphi_j(y) = \perp)$

     $\implies \forall i \in A, \forall j \in \overline{A}, \exists x, y : x \neq y \wedge (\varphi_i(x) \neq \varphi_j(x) \vee \varphi_i(y) \neq \varphi_j(y))$

     $\implies \forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$
   - Par la contraposée du théorème de Rice, A n'est pas récursif.

5. *L'ensemble des programmes qui calculent une fonction donnée $f : \mathbb{N} \to \mathbb{N}$. (Votre réponse dépend-elle de la fonction f ?)*

   Réponse :

   - **Non récursif si f est calculable.** Soit $A = \{i \mid \forall n : \varphi_i(n) = f(n)\}$. $A \neq \emptyset$ car la fonction étant calculable, un programme i qui la calcule existe. $A \neq \mathbb{N}$ car il existe des programmes qui ne calculent pas f.

     $\implies \forall i \in A, \forall n : \varphi_i(n) = f(n)$ et $\forall j \in \overline{A}, \exists n : \varphi_j(n) \neq f(n)$

     $\implies \forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$

     Par la contraposée du théorème de Rice, A n'est pas récursif.

   - **Récursif si f est non calculable** car dans ce cas $A = \emptyset$ (car il n'est pas possible d'écrire un programme qui calcule une fonction non calculable).

Les affirmations suivantes sont-elle vraies ?

6. *Il existe un programme Python qui décide si la fonction calculée par un programme donnée a un domaine fini.*

   Réponse : Faux

   - Soit $A = \{i \mid \varphi_i$ possède un domaine fini $\}$. $A \neq \emptyset$ car il existe de telles fonctions (e.g. une fonction retournant 42 pour l'entrée 1 et $\perp$ pour toutes les autres entrées). $A \neq \mathbb{N}$ car il existe aussi des fonctions calculables n'ayant pas un domaine fini (par exemple : $f(n) = n$).

   - $\forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$ car ces fonctions ont des domaines différents : la première a un domaine fini tandis que la seconde a un domaine infini.

   - Par la contraposée du théorème de Rice, A n'est pas récursif. La fonction caractéristique de A n'est donc pas calculable et un tel programme ne peut donc exister.

7. *Il n'existera jamais un programme Python ramasse-miettes (garbage collector) qui soit à la fois sûr et optimal.*

   Pour cette question, on considère qu'un programme ramasse-miette (à la fois sûr et optimal) est un programme capable de déterminer sans jamais faire d'erreur durant l'exécution de n'importe quel programme s'il est possible de libérer de l'espace mémoire qui a été utilisée précédemment (autrement dit, si l'espace mémoire en question ne sera plus utilisé d'une quelconque manière que ce soit durant toute la suite de l'exécution du programme).

   Réponse : Vrai

   Soit le programme suivant :

   ```
   x = 0
   P_n(k)
   return x
   ```

   Décider *de manière certaine* si on peut libérer `x` après la 1ère instruction revient à toujours pouvoir calculer $halt(n, k)$ pour savoir si `x` risque d'être ré-utilisé par après dans le programme. En effet, si un ramasse-miettes sûr et optimal ne libère pas `x` après la première instruction, c'est parce que l'exécution de $P_n(k)$ se termine et qu'on a besoin de `x` pour l'instruction `return`. Si $P_n(k)$ ne se termine pas, alors un ramasse-miettes optimal doit libérer `x` immédiatement après la 1ère instruction, car le `return x` ne s'exécutera jamais. Or *halt* est une fonction non-calculable. Il n'est donc pas possible de savoir si `x` sera ré-utilisé.

8. *L'ensemble* $A = \{i \in \mathbb{N} \mid P_i(x) \text{ se termine pour un certain } x\}$ *est récursif.*

   Réponse : Faux

   Soit $A = \{i \mid \exists x : \varphi_i(x) \neq \bot\}$

   - $A \neq \emptyset$ (e.g. $P_i(x) \equiv$ `return 0`) et $A \neq \mathbb{N}$ (e.g. $P_j(x) \equiv$ `while True: pass`)
   - $\forall i \in A, \exists x : \varphi_i(x) \neq \bot$ et $\forall j \in \overline{A}, \forall x : \varphi_j(x) = \bot$

     $\implies \forall i \in A, \forall j \in \overline{A}, \exists x : \varphi_i(x) \neq \bot \wedge \varphi_j(x) = \bot$

     $\implies \forall i \in A, \forall j \in \overline{A}, \exists x : \varphi_i(x) \neq \varphi_j(x)$

     $\implies \forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$
   - Par la contraposée du théorème de Rice, A n'est donc pas récursif.

9. *L'ensemble* $A = \{i \in \mathbb{N} \mid P_i(x) \text{ se termine pour un certain } x\}$ *est récursivement énumérable.*

   Réponse : Vrai

   Pour décider si un certain $i \in A$, on peut "exécuter en parallèle" $P_i(0)$, $P_i(1)$, $P_i(2)$, ..., i.e. exécuter P_i pour toutes les entrées possibles (soit une infinité d'exécutions en parallèle). Si l'une de ces exécutions se termine, alors on sait que $i \in A$.

   On peut simuler l'exécution parallèle d'une infinité de P_i en exécutant 1 instruction à la fois en suivant la table suivante en diagonale, jusqu'au moment où un P_i se termine.

	Instructions du programme
$P_i(0)$	• • • • • • ⋯
$P_i(1)$	• • • • `return 1`
$P_i(2)$	• • • • • • ⋯
⋮	

   Dans ce cas, on sait que $i \in A$. Sinon, on a une boucle infinie. A est donc bien récursivement énumérable.

10. *Nous avons vu que la fonction qui calcule les résultats du prochain lotto est bien calculable (il suffit d'écrire autant de programmes qu'il y a de combinaisons possibles, chacun produisant une des combinaisons possibles quelle que soit l'entrée).*

    Mais est-il possible, étant donné un programme Python, de déterminer s'il produira justement la bonne combinaison ? Votre réponse est-elle différente en fonction de si vous connaissez à l'avance la bonne combinaison ?

    Réponse : Impossible par le théorème de Rice, même si on connait la combinaison à l'avance !

    - Soit $A = \{i \mid \varphi_i$ produit la bonne combinaison$\}$. $A \neq \emptyset$ car la fonction est calculable. $A \neq \mathbb{N}$ car il existe aussi des fonctions calculables renvoyant la mauvaise combinaison.
    - $\forall i \in A, \forall j \in \overline{A} : \varphi_i \neq \varphi_j$ car, par définition, les programmes i renvoient la bonne combinaison et les programmes j renvoient une mauvaise combinaison.
    - Par la contraposée du théorème de Rice, A n'est pas récursif. La fonction caractéristique de A n'est donc pas calculable et il n'est donc pas possible de déterminer si un programme produira la bonne combinaison.
    - Connaître la combinaison à l'avance ne change rien à la démonstration.

5 Théorème S-m-n et théorème du point fixe

1. *Un transformateur de programmes est toujours une fonction calculable.*

   Réponse : Faux

   Un transformateur de programmes peut être vu comme une fonction $f : \mathbb{N} \to \mathbb{N}$. Or, il existe certaines fonctions totales qui ne sont pas calculables.

   Remarque. Même si on impose la condition qu'un transformateur de programme doit être injectif, la réponse ne change pas. Voici un exemple de fonction injective non calculable :

   $$f(x) = \begin{cases} 2x & \text{si } halt(x,x) = 1 \\ 2x+1 & \text{si } halt(x,x) = 0 \end{cases}$$

2. *En vous aidant du théorème S-m-n, déterminez s'il existe une fonction totale calculable $f : \mathbb{N} \to \mathbb{N}$ telle que pour tout $x \in \mathbb{N}$, on a :* $\mathrm{dom}(\varphi_{f(x)}) = \{y \in \mathbb{N} \mid \exists z \in \mathbb{N} : y = z^x\}$.

   Indice. *Considérez la fonction ci-dessous :*

   $$f(x,y) = \begin{cases} \text{le plus petit } z \in \mathbb{N} \text{ tel que } y = z^x & \text{si un tel } z \text{ existe} \\ \text{non défini} & \text{sinon} \end{cases}$$

   Réponse : Vrai

   Selon S-m-n, $\exists g$ total calculable tel que $\forall x, y$:

   $$\varphi_k(x,y) = \varphi_{g(k,x)}(y) = \begin{cases} z \,|\, y = z^x & \text{si } z \text{ existe} \\ \perp & \text{sinon} \end{cases}$$

   $y \in \mathrm{dom}(\varphi_{g(k,x)}(y)) \iff \exists z \in \mathbb{N} : y = z^x$

3. *Quel est le sens intuitif de l'affirmation suivante ?*

   Il existe une fonction calculable $f : \mathbb{N} \to \mathbb{N}$ telle que, pour tout $x \in \mathbb{N}$, si φ_x est la fonction caractéristique d'un ensemble $A_x \subseteq \mathbb{N}$, alors $\varphi_{f(x)}$ est la fonction caractéristique de $\overline{A_x}$.

   (a) Il existe une fonction calculable qui permet de décider si le complémentaire d'un ensemble récursif est aussi un ensemble récursif.

   (b) Si on désigne les ensembles récursifs par un indice de leur fonction caractéristique, alors l'opération de passage au complémentaire est calculable.

   (c) L'indice de n'importe quelle fonction calculable est soit l'indice d'une fonction caractéristique d'un ensemble A, soit l'indice d'une fonction caractéristique du complémentaire de A.

   Réponse : (b)

4. *L'affirmation suivante est-elle vraie ?*

   Il existe une fonction fonction calculable $f : \mathbb{N} \to \mathbb{N}$ telle que, pour tout $x \in \mathbb{N}$, si φ_x peut être vue comme la fonction caractéristique d'un ensemble $A_x \subseteq \mathbb{N}$ (autrement dit : si l'image de φ_x est $\{0,1\}$), alors $\varphi_{f(x)}$ est la fonction caractéristique de $\overline{A_x}$.

   Indice. *Considérez la fonction ci-dessous.*

   $$f(i,x) = \max\{0, 1 - \varphi_i(x)\}$$

   Réponse : Vrai

   $\varphi_k(i,x) = \max\{0, 1 - \varphi_i(x)\}$ permet de décider $x \in \overline{A_i}$. Par S-m-n, $\exists g$ total calculabel tel que :

   $$\varphi_k(i,x) = \varphi_{g(k,i)}(x)$$

5. *La réciproque du théorème du point fixe est-elle vraie ?*

   Autrement dit : si on a $n \in \mathbb{N}$ et $f : \mathbb{N} \to \mathbb{N}$ une fonction totale, le fait qu'il existe un $k \in \mathbb{N}$ tel que $\varphi_k = \varphi_{f(k)}$ implique-t-il que f est calculable ?

   Réponse : Faux

   Soit la fonction totale :

$$f(k) = \begin{cases} k & \text{si } k \in K \\ 0 & \text{sinon} \end{cases}$$

   Soit :

$$P_k(x) \equiv \texttt{return } x$$

   Comme $\varphi_k(k)$ se termine, $k \in K$ et donc $\varphi_{f(k)} = \varphi_k$. Pourtant f n'est pas calculable sinon K serait récursif.

6 Automates

1. *L'affirmation suivante est-elle vraie ? L'ensemble des fonctions calculées par le langage BLOOP est exactement l'ensemble des fonctions totales calculables.*

   Réponse : Faux

   Le théorème d'Hoare-Allison nous dit qu'un langage de programmation qui ne permet de calculer que des fonctions totales ne permet pas de calculer sa fonction interpréteur (qui est une fonction totale).

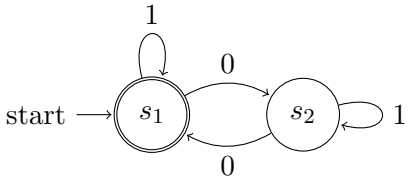
2. *L'affirmation suivante est-elle vraie ? $\overline{K}$ est ND-récursivement énumérable.*

   Réponse : Faux

   Si un ensemble est ND-récursivement énumérable, il est récursivement énumérable. Or, $\overline{K}$ n'est pas récursivement énumérable (voir question 6 de la section 2).

   Sous-question. Comment justifier qu'un ensemble ND-récursivement énumérable est automatiquement récursivement énumérable ?

3. *Que permet de décider l'automate fini déterministe ci-dessous ?*



   Réponse : Si l'entrée contient un nombre pair de 0.

4. *Quel ensemble de mots l'automate suivant décide-t-il ?*

   - Alphabet : $\Sigma = \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$
   - Ensemble des états : $S = \{s_0, s_1\}$
   - État initial : s_0
   - Ensemble des états acceptants : $A = \{s_1\}$
   - Table de la fonction de transition :

	x	y	z
s_0	s_1	s_0	s_0
s_1	s_1	s_1	s_1

   Réponse : L'ensemble des mots formés à partir des lettres x, y, z qui contiennent au moins un x.

Les affirmations suivantes sont-elle vraies ?

5. *L'interpréteur des automates finis peut être représenté à l'aide d'un automate fini.*

   Réponse : Faux

   Le formalisme des automates finis ne permet de calculer que des fonctions totales. Comme avec le langage BLOOP de la question 1, on en déduit qu'il ne permet pas de calculer sa fonction interpréteur (qui est une fonction totale).

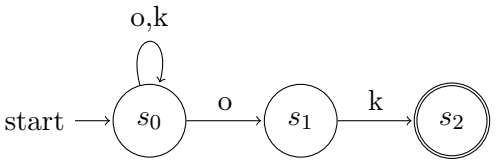
6. *Tout sous-ensemble d'un langage accepté par un automate fini est récursif.*

   Réponse : Faux

   Il existe un automate qui accepte l'ensemble de tous les mots formé à partir de l'alphabet $\Sigma = \{\mathrm{a}\}$ (qui est certainement récursif). Mais le sous-ensemble de cet ensemble des mots formés à partir de cet alphabet dont la longueur est dans K (pour un langage de programmation choisi) n'est pas récursif.

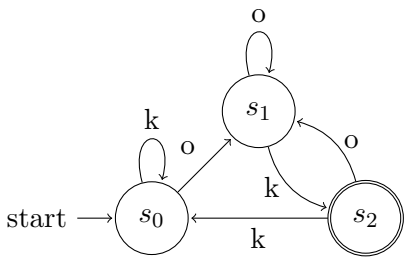
7. *Il existe un automate fini* non déterministe *qui permet de déterminer le langage des mots construits à partir de l'alphabet* $\Sigma = \{\mathrm{k}, \mathrm{o}\}$ *qui terminent avec* "ok".

   Réponse : Vrai



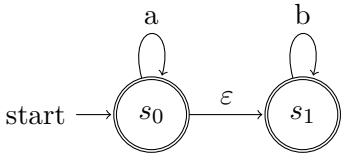
8. *Il existe un automate fini* déterministe *qui permet de déterminer le langage des mots construits à partir de l'alphabet* $\Sigma = \{\mathrm{k}, \mathrm{o}\}$ *qui terminent avec* "ok".

   Réponse : Vrai



9. *Il existe un automate fini avec transition vide qui permet de déterminer le langage* $\{\mathrm{a}^m\mathrm{b}^n \mid n, m \in \mathbb{N}\}$.

   Réponse : Vrai



   Sous-question. La réponse change-t-elle si on fixe m et n ?

10. *Il existe un automate fini déterministe qui permet de déterminer le langage* $\{\mathrm{a}^m\mathrm{b}^n \mid n, m \in \mathbb{N}\}$.

    Réponse : Vrai

    Les automates avec ε-transition ont la même puissance que les automates finis déterministes (voir slides du cours).

    Sous-question. La réponse change-t-elle si on fixe m et n ?

7 Machines de Turing

Voici la description d'une machine de Turing avec comme alphabets $\Sigma = \{a, b\}$ et $\Gamma = \{a, b, x, B, 0, 1\}$:

state	**symbol**	**state**	**movement**	**symbol**
start	a	seekB	$\rightarrow$	x
start	b	seekA	$\rightarrow$	x
start	x	start	$\rightarrow$	B
start	B	stop	$\downarrow$	1
seekA	a	restart	$\leftarrow$	x
seekA	b	seekA	$\rightarrow$	b
seekA	x	seekA	$\rightarrow$	x
seekA	B	false	$\leftarrow$	B
seekB	b	restart	$\leftarrow$	x
seekB	a	seekB	$\rightarrow$	a
seekB	x	seekB	$\rightarrow$	x
seekB	B	false	$\leftarrow$	B
restart	a	restart	$\leftarrow$	a
restart	b	restart	$\leftarrow$	b
restart	x	restart	$\leftarrow$	x
restart	B	start	$\rightarrow$	B
false	a	false	$\leftarrow$	B
false	b	false	$\leftarrow$	B
false	x	false	$\leftarrow$	B
false	B	stop	$\downarrow$	0

1. *Quel est le résultat pour l'entrée* "bbba" *?*

   Réponse : 0

2. *Quel est le résultat pour l'entrée* "baba" *?*

   Réponse : 1

3. *Quelle seront les deux premières cases dont le contenu sera modifié en* x *si l'entrée commence par un* a *?*

   Réponse : La première case (= le premier a) et le premier b.

4. *Que signifie le symbole* x *(du point de vue de l'exécution) ?*

   Réponse : Il représente l'emplacement d'un symbole d'entrée qui a déjà été vérifié par la machine de Turing.

5. *Que calcule cette machine de Turing ?*

   Réponse : Cette machine de Turing vérifie si l'entrée possède le même nombre de a que de b.

Les affirmations suivantes sont-elle vraies ?

6. *Si une machine de Turing permet de décider un langage L, alors L est récursif.*

   Réponse : Vrai

   Si une machine de Turing peut décider L, alors L est T-récursif. Par la thèse de Church-Turing, un ensemble T-récursif est récursif.

7. *L'exécution d'une machine de Turing se termine toujours.*

   Réponse : Faux

   On peut par exemple imaginer une machine qui alternerait indéfiniment entre des mouvements droite/gauche.

8. *Les machines de Turing non déterministes sont plus puissantes (i.e. permettent de calculer davantage de fonctions) que les machines de Turing déterministes.*

   Réponse : Faux

   Il existe une machine de Turing qui interprète les machines de Turing non déterministes. Les fonctions calculables par les machines non déterministes sont donc calculables par les déterministes. Elles ont ainsi la même puissance.

9. *Les machines de Turing équipées d'un oracle pour l'ensemble :*

$$A = \{x \in \mathbb{N} \mid x \,\%\, 2 = 0\}$$

   sont plus puissantes que les machines de Turing de base.

   Réponse : Faux

   Décider A est possible sans oracle (e.g. $P(x) \equiv$ `return 1 - (x % 2)`). Une machine de Turing équipée d'un tel oracle n'est donc pas plus puissante qu'une machine de Turing de base.

10. *Les machines de Turing équipées d'un oracle pour l'ensemble :*

$$A = \{x \in \mathbb{N} \mid \varphi_x(x) = 42\}$$

    sont plus puissantes que les machines de Turing de base.

    Réponse : Vrai

    On peut montrer que A est non-récursif. Supposons par l'absurde que A est récursif et appliquons la méthode de réduction. Soit P_A le programme donné par le fait que A est récursif, montrons que K est récursif :

$$P_K(n) \equiv \left[\begin{array}{l} \texttt{write } P_q(y) \equiv \left[\begin{array}{l} P_n(n) \\ \texttt{return } 42 \end{array}\right. \\ \texttt{return } P_A(q) \end{array}\right.$$

    C'est absurde. Donc A ne peut pas être récursif.

    Une machine de Turing équipée d'un tel oracle serait donc capable d'évaluer la fonction caractéristique de A (un ensemble non récursif), de calculer *halt*, etc., à l'inverse d'une machine de Turing de base qui ne le pourrait pas.

8 Logique des propositions

1. *En logique des propositions, si on se limite à deux variables propositionnelles A et B, il n'est possible de construire que 20 formules :*

   A, B, $\neg A$, $\neg B$, $(A \vee A)$, $(B \vee B)$, $(A \vee B)$, $(B \vee A)$, $(A \wedge A)$, $(B \wedge B)$, $(A \wedge B)$, $(B \wedge A)$, $(A \Rightarrow A)$, $(B \Rightarrow B)$, $(A \Rightarrow B)$, $(B \Rightarrow A)$, $(A \Leftrightarrow A)$, $(B \Leftrightarrow B)$, $(A \Leftrightarrow B)$, $(B \Leftrightarrow A)$

   Réponse : Faux

   On peut construire les règles de création de formules sur des formules non atomiques pour créer des formules plus complexes. Exemple : $(A \wedge \neg B) \vee (\neg A \wedge B)$.

2. *Pour un choix fini et fixé de variables propositionnelles A, B, C, D, E, F, l'ensemble des chaînes de caractères construites à partir des symboles $A, B, C, D, E, F, (,), \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ qui sont des formules de la logique des propositions est un ensemble récursif.*

   Réponse : Vrai

   Voici un algorithme qui prend en entrée une chaîne de caractères et qui décide s'il s'agit d'une formule valable :

   $$P(c) \equiv \left[\begin{array}{l} n = \text{longueur de la chaine de caractères } c \\ L = \text{liste qui contient toutes les formules d'une} \\ \quad\quad \text{longueur} < n \text{ qu'il est possible de construire} \\ \quad\quad \text{à partir des variables que l'on retrouve dans } c \\ \texttt{return } c \texttt{ in } L \end{array}\right.$$

3. *Avec l'interprétation A : vrai et B : faux, la formule $((A \Rightarrow B) \Rightarrow A)$ est fausse.*

   Réponse : Faux

   Si A est vrai et B est faux, alors $(A \Rightarrow B)$ est faux. Si une proposition est fausse, elle implique n'importe quelle autre proposition (qu'elle soit vraie ou fausse). Dès lors, $((A \Rightarrow B) \Rightarrow A)$ est vrai.

4. *Une formule de la logique des propositions peut avoir une infinité de modèles différents.*

   Réponse : Faux

   Un modèle d'une formule est une interprétation dans laquelle la formule est vraie. Or, pour une formule avec n variables, il y a un nombre fini d'interprétations : 2^n. Le nombre de modèles est donc nécessairement fini.

5. *Si une interprétation I est un modèle pour une formule p de logique des propositions, alors il existe nécessairement au moins une autre formule q pour laquelle I est également un modèle.*

   Réponse : Vrai

   Si l'on construit une formule q comme une disjonction dont l'une des propositions est la formule p, la formule q sera également vraie pour l'interprétation I.

6. *Toute tautologie est satisfaisable.*

   Réponse : Vrai

   Par définition : une formule est une tautologie lorsqu'elle est vraie dans toutes les interprétations. Elle est donc forcément satisfaisable.

7. *Si une formule p de la logique des propositions est satisfaisable, alors la formule $\neg p$ ne l'est pas.*

   Réponse : Faux

   Par exemple, la formule A est satisfaisable avec l'interprétation A : vrai. La formule $\neg A$ est également satisfaisable, avec l'interprétation A : faux.

8. *Si on a deux formules p et q de la logique des propositions, alors au moins une des affirmations suivantes doit être vraie :*

   - $p \models q$
   - $q \models p$
   - $\neg p \models q$
   - $q \models \neg p$
   - $p \models \neg q$
   - $\neg q \models p$
   - $\neg p \models \neg q$
   - $\neg q \models \neg p$

   Réponse : Faux

   Soient les variables A et B, si l'on pose $p = A$ et $q = B$, on peut trouver pour chaque affirmation une interprétation qui la rend fausse.

9. *La formule de la logique des propositions suivante est sous forme normale conjonctive :*

$$(\neg A \vee B) \wedge (B \vee \neg C \vee (D \wedge A)) \wedge \neg(A \vee E)$$

   Réponse : Faux

   Une formule est en forme normale conjonctive si elle est une conjonction de disjonctions de variables ou de leur négation. Il y a donc deux problèmes avec la formule donnée :

   - La deuxième clause comprend un élément qui n'est pas un littéral.
   - Il y a une négation devant la troisième clause.

10. *La formule suivante est-elle satisfaisable ?*

$$(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$$

    Réponse : Vrai

    Une interprétation valide est la suivante. On choisit A : faux pour que les deux premières disjonctions soient vraies. On choisit B : faux pour que les deux disjonctions suivantes soient également vraies. Pour que la dernière disjonction soit vraie, il suffit de prendre C : vrai ou D : faux.

9 Complexité algorithmique

1. *Quelle est la complexité de cet algorithme ?*

```
sum(n)
  sum = n * n * n
    for i = 0 to n
      for j = 0 to i
        sum++
```

Réponse : $T(n) = 1 + \frac{1}{2}n(n+1) \Longrightarrow \mathcal{O}(n^2)$

2. *Quelle est la complexité de cet algorithme ?*

```
fun(n)
  if (n > 1)
    return 2 * fun(n/2)
  else
    return n
```

Réponse : $\mathcal{O}(\ln(n))$, car :

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T\left(\frac{n}{2}\right) + 1 \\ &= T\left(\frac{n}{2^k}\right) + k \qquad\qquad \frac{n}{2^k} = 1 \Rightarrow k = \log_2(n) \\ &= 1 + \log_2(n) \end{aligned}$$

3. *Quelle est la complexité de cet algorithme ?*

```
more_fun(n)
  if (n > 1)
    return more_fun(n/2) + more_fun(n/2)
  else
    return n
```

Réponse : $\mathcal{O}(n)$, car :

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + 1 \\ &= 2^k \cdot T\left(\frac{n}{2^k}\right) + 2^k - 1 \qquad\qquad \frac{n}{2^k} = 1 \Rightarrow k = \log_2(n) \\ &= 2n - 1 \end{aligned}$$

4. *Quelle est la complexité de cet algorithme ?*

```
explosion(n)
  if (n > 1)
    return explosion(n-1) + explosion(n-2)
  else
    return n+1
```

Réponse : $\mathcal{O}(3^n)$

On sait que $T(0) = 1$ and $T(1) = 1$. Pour tout $n \in \mathbb{N}$, on a la relation $T(n) = T(n-1)+T(n-2)+1$.

On peut calculer d'abord à la main les termes suivants : 1, 1, 3, 5, 9, 15, 25, ... Les éléments $T(n)$ de cette suite sont tous tels que $\mathcal{F}(n) \leq T(n) \leq 3^n$ pour tout $n \in \mathbb{N}$ (où $\mathcal{F}(n)$ est la suite de Fibonacci) (cela se prouve directement par récurrence).

(Pour info : $T(n) = \mathcal{F}(n) + \mathcal{L}(n) - 1$ où $\mathcal{L}(n)$ est la suite de Lucas (une variante de Fibonacci qui commence par $1, 3, ...$).)

5. *L'égalité suivante est-elle vraie ?*

$$\mathcal{O}(\log_2(x^2)) = \mathcal{O}(\log_{10}(x^{10}))$$

Réponse : Vrai

On a :

$$\log_2(x^2) = 2 \cdot \log_2(x) = 2 \cdot \frac{\ln(x)}{\ln(2)} \qquad \text{et} \qquad \log_{10}(x^{10}) = 10 \cdot \log_{10}(x) = 10 \cdot \frac{\ln(x)}{\ln(10)}$$

Les facteurs constants pouvant être ignorés, on est bien en $\mathcal{O}(\ln(x))$ dans les 2 cas.

6. *Si on a deux fonctions f, g de $\mathbb{N}$ dans $\mathbb{N}$ telles que pour tout $n \in \mathbb{N}$, $g(n) \neq 0$ et $g(n) \in \mathcal{O}(f(n))$, la limite suivante existe-t-elle nécessairement ?*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Réponse : Faux

Prenons $f(n) = 2$ et $g(n) = \begin{cases} 1 & \text{si } n \text{ impair} \\ 2 & \text{si } n \text{ pair} \end{cases}$

Pour tout $n \in \mathbb{N}$, $g(n) \neq 0$ et $g(n) \leq f(n)$ donc $g(n) \in \mathcal{O}(f(n))$. Or $\frac{f(n)}{g(n)}$ oscille entre 1 et 2 donc la limite n'existe pas.

7. **Réponse :** *Puisque les matrices sont des objets à deux dimensions (lignes et colonnes), l'algorithme élémentaire de multiplication matricielle (de deux matrices carrées de taille n) a une complexité dans $\mathcal{O}(n^2)$.*

Réponse : Faux

Les matrices sont des objets à deux dimensions mais le calcul de chaque élément du résultat d'une multiplication matricielle implique $2n$ termes et nécessite $2n - 1$ opérations ! Comme il y a en tout n^2 éléments à calculer, cet algorithme a donc une complexité dans $\mathcal{O}(n^3)$.

8. *Un problème résolu par un algorithme dont la complexité est dans $\mathcal{O}(2^n)$ est intrinsèquement complexe.*

Réponse : Faux

Un tel problème *n'est pas forcément* intrinsèquement complexe. La complexité d'un problème est la complexité de l'algorithme *le plus efficace* résolvant ce problème. Il est tout à fait possible d'écrire un programme dans $\mathcal{O}(2^n)$ qui résout un problème dans $\mathcal{O}(n^2)$, mais il ne serait pas le plus efficace.

9. *Il est toujours impossible de trouver une solution à un problème intrinsèquement complexe en un temps réaliste quelque soit la donnée.*

   Réponse : Faux

   Pour des petits exemples, il est possible de trouver une solution à un problème intrinsèquement complexe en un temps réaliste. De plus, il est même possible que pour *certains* grands exemples, une solution puisse également être trouvée en un temps réaliste.

10. *Comme la loi de Moore prédit que la progression de la puissance de calcul des ordinateur suit une progression exponentielle, les problèmes intrinsèquement complexes pourront être facilement résolus dans quelques dizaines d'années.*

    Réponse : Faux

    Alors que la progression de la puissance de calcul permet d'améliorer la résolution de problèmes polynomiaux d'un facteur multiplicatif, elle n'améliore la résolution des problèmes intrinsèquement complexes que d'un facteur additif. Peu importe les améliorations technologiques, un problème intrinsèquement complexe ne pourra être résolu que pour des exemples de petites tailles.

10 Complexité et réductions

1. *Si un problème A peut être réduit à un problème B et que A est calculable, alors B est calculable.*

   Réponse : Faux

   $A \leq B$ signifie si B est calculable, alors on peut calculer A. Par contre, la calculabilité de A ne dit rien sur celle de B.

2. *Soient deux ensembles $A, B \subseteq \mathbb{N}$. Si $A \leq_a B$ et si B est récursif, alors A est récursif.*

   Réponse : Vrai

   Il s'agit de la définition de $\leq_a$.

3. *Soient deux ensembles $A, B \subseteq \mathbb{N}$. Si $A \leq_a B$ et si B n'est pas récursif, alors A n'est pas récursif.*

   Réponse : Faux

   Un contre-exemple est donné par K_{BLOOP} et K. Comme les programmes de BLOOP s'arrêtent toujours, K_{BLOOP} est récursif, au contraire de K. Or, comme BLOOP est un sous-ensemble de Python ; on a donc bien $K_{\text{BLOOP}} \leq_a K$.

4. *Soient deux ensembles $A, B \subseteq \mathbb{N}$. Si A est récursif, alors $A \leq_f B$.*

   Réponse : Vrai

   $A \leq_f B$ est vrai dans le cas où si on peut décider B, on peut aussi décider A. Or A est déjà décidable, donc A est réductible à n'importe quel B.

5. *Soient deux ensembles $A, B \subseteq \mathbb{N}$. Alors $A \leq_f B \Leftrightarrow \bar{A} \leq_f \bar{B}$.*

   Réponse : Vrai

   S'il existe une fonction totale calculable telle que $a \in A \Leftrightarrow f(a) \in B$, cette même fonction implique que $a \notin A \Leftrightarrow f(a) \notin B$, et donc $a \in \bar{A} \Leftrightarrow f(a) \in \bar{B}$.

6. *Soit C une classe de problèmes. Si on trouve un algorithme permettant de décider un des problèmes $A \in C$, alors il est possible de décider n'importe quel autre problème dans C.*

   Réponse : Faux

   Pour que cette affirmation soit vraie, il faut que A soit C-complet, i.e. que tout autre problème de C puisse être réduit à A :

   $$\forall B \in C : B \leq A$$

   A doit donc être le problème "le plus compliqué" de la classe C afin de rendre cette affirmation vraie.

7. *Soient $A, B \subseteq \mathbb{N}$. Si $A \leq_a B$ et qu'on connait la complexité de B, alors on peut en déduire la complexité de A.*

   Réponse : Faux

   La réduction algorithmique permet de déduire la récursivité de A en connaissant celle de B, mais ne donne aucune information sur la complexité. En effet, on peut utiliser autant de fois que l'on veut la récursivité de B pour calculer A. $\leq_a$ ne donne donc aucune limite sur la complexité de A.

8. *Soient $A, B \subseteq \mathbb{N}$. Si $A \leq_f B$ et qu'on connait la complexité de B, alors on peut en déduire la complexité de A.*

   Réponse : Vrai

   Par définition de $\leq_f$, on utilise un unique test $f(a) \in B$ pour décider $a \in A$. On peut donc déduire la complexité de A à partir de celle de la fonction de décision de B et de celle de f.

9. *Étant donné que les machines de Turing sont un modèle complet de calculabilité et un modèle de complexité (i.e. on peut calculer précisément la complexité d'une machine de Turing), tout problème peut se calculer sur une machine de Turing avec la même complexité qu'en Python.*

   Réponse : Faux

   Tous les modèles de complexité ont des complexités reliées de manière *polynomiale*. Une MT peut donc s'exécuter avec une complexité supérieure à l'équivalent en Python (mais cette exécution restera toujours pratiquement faisable si c'est le cas avec Python).

10. *La classe NP (pour "Non Polynomial") est définie comme l'ensemble des problèmes ne pouvant pas être résolus par des programmes de complexité polynomiale.*

    Réponse : Faux

    La classe *NP* (pour "*Non-déterministe* Polynomial") est définie comme l'ensemble des problèmes pouvant être résolus par des programmes *non-déterministes* de complexité *polynomiale* :

    $$NP = \bigcup_{i \geq 0} NTIME(n^i)$$

    La classe *NP* inclut donc aussi les problèmes pouvant être résolus par des programmes *déterministes* de complexité polynomiale, i.e. $P \subseteq NP$, au contraire de ce que semble dire l'affirmation ci-dessus (qui affirme erronément $NP = \overline{P}$).

11 Classes de complexité et NP-complétude

1. *Soient deux problèmes de décision A et B. Si $A \leq_p B$, alors on a automatiquement $A \leq_a B$ et $A \leq_f B$.*

   Réponse : Vrai

   $A \leq_p B$ signifie que A est *polynomialement réductible* à B. Il s'agit d'un cas particulier plus restrictif de $\leq_f$ où l'on impose que la fonction f soit de complexité temporelle polynomiale. Précédemment, nous avions également vu que $A \leq_f B$ impliquait $A \leq_a B$.

2. *Soient X et Y deux problèmes de décisions tels que $X \leq_f Y$. Si Y est dans P, alors X aussi.*

   Réponse : Faux

   $X \leq_f Y$ nous indique qu'il existe une fonction totale calculable f telle que $x \in X \Leftrightarrow f(x) \in Y$ mais on ne connait pas la complexité de la fonction f. Si celle-ci prend un temps exponentiel par exemple (donc non-polynomial) alors X n'appartient pas forcément à P.

3. *Soient X et Y deux problèmes de décisions tels que $X \leq_p Y$. Si X est dans P, alors Y aussi.*

   Réponse : Faux

   Y peut être dans *NP* car $X \in P \subset NP$. Par exemple, on pourrait avoir $X =$ déterminer si un nombre est pair, et $Y =$ SAT.

4. *Soient X et Y deux problèmes de décisions tels que $X \leq_p Y$. Si Y est dans P, alors X aussi.*

   Réponse : Vrai

   La réduction $\leq_p$ a été définie dans cette optique. Si Y est dans P et qu'il existe une fonction totale calculable en un temps polynomial f telle que $x \in X \Leftrightarrow f(x) \in Y$, X est également dans P. Il suffit d'appliquer cette fonction f à l'élément de X qui nous intéresse et ensuite d'utiliser l'algorithme de décision polynomial de Y. Cela se fera bien en un temps polynomial.

5. *Soient X et Y deux problèmes de décisions tels que $X \leq_p Y$. Si X est NP-complet, alors Y aussi.*

   Réponse : Faux

   Y peut être *NP*-difficile ou non calculable (exemple : $X =$ SAT et $Y =$ HALT).

6. Soient X et Y deux problèmes de décisions tels que $X \leq_p Y$. Si X est *NP*-complet et Y est *NP*, alors Y est *NP*-complet.

   Réponse : Vrai

   Si X est un problème *NP*-complet (par rapport à $\leq_p$), cela signifie qu'il appartient à *NP* mais aussi que tout problème appartenant à *NP* peut-être réduit à X. Or nous avons ici que $X \leq_p Y$. Y est donc nécessairement *NP*-complet sinon X ne pourrait pas être réduit à Y.

7. *Le théorème de Cook affirme que SAT est NP-complet. Désignons par SATpair le problème quasi-identique de satisfaction d'une formule propositionnelle mais qui ne peut posséder qu'un nombre pair de variables propositionnelles. Le problème SATpair est-il aussi compliqué que le problème SAT ? Formulé autrement : le problème SATpair est-il NP-complet ?*

Réponse : Vrai

SAT $\leq_p$ SATpair (il suffit de rajouter une variable bidon s'il y a un nombre impair de variables) et SATpair $\in NP$, donc SATpair est NP-complet.

8. *Soient X, Y et Z, trois problèmes de décision. Si $X \leq_p Y$ et $Y \leq_p Z$, peut-on affirmer avec certitude que $X \leq_p Z$?*

   Remarque. *Nous supposons que, dans ces réductions polynomiales, les tailles des outputs des fonctions de complexité temporelle polynomiale f sont bornées par les complexités de f.*

   Réponse : Vrai

   La réduction polynomiale $\leq_p$ est une relation transitive. En effet, s'il existe une fonction f calculable de manière polynomiale telle que $x \in X \Leftrightarrow f(x) \in Y$ et une fonction g calculable de manière polynomiale telle que $y \in Y \Leftrightarrow g(y) \in Z$, il existe une fonction h (comme $f \circ g(x) = f(g(x))$ grâce à la suppposition sur la taille de l'output) calculable en un temps polynomial telle que $x \in X \Leftrightarrow h(x) \in Z$.

9. *Il existe une infinité de problèmes NP-complets.*

   Réponse : Vrai

   Notons SATmin$N$, le problème SAT pour les formules qui contiennent au moins N variables propositionnelles différentes (SATmin2 correspond donc aux problèmes SAT pour les formules qui contiennent au moins 2 variables propositionnelles différentes, SATmin3 pour les formules qui contiennent au moins 3 variables propositionnelles différentes et ainsi de suite). Il existe une infinité de ces problèmes SATmin.

   Or, nous avons que $\forall N$, SATmin$N \leq_p$ SAT (car SATmin$N$ est un cas particulier de SAT) et SAT $\leq_p$ SATmin$N$ (car si une expression ne contient pas suffisamment de variables, il suffit de lui ajouter des variables bidons, ex : $\wedge$ Bidon1 $\wedge$ Bidon2...). Il existe donc une infinité de problèmes NP-complets.

10. $P = NP$

    Réponse : La personne qui peut répondre et le prouver gagne 1 000 000 USD !

12 Propriétés des modèles de calculabilité

1. *Il est possible de créer un langage de programmation qui (en tant que formalisme de calculabilité) possède la propriété SA (soundness algorithmique) mais pas la propriété SD (soundness des définitions).*

   Réponse : Faux

   Si la fonction interpréteur $I(n, k)$ de ce langage de programmation est calculable (par une machine de Turing M_I), alors toute fonction φ_{n^*} calculable à l'aide de ce langage de programmation est calculable (par une machine de Turing). En effet, il suffit d'utiliser la machine de Turing M_{n^*} telle que pour tout $k \in \mathbb{N}$ on ait $M_{n^*}(k) = M_I(n^*, k)$ (que l'on peut obtenir de façon purement théorique grâce au théorème S-m-n).

2. *Le langage BLOOP est un formalisme de calculabilité qui possède les propriétés suivantes :*

   (a) *SD, SA, S, non CD, non CA, non U*

   (b) *SD, SA, S, non CD, non CA, U*

   (c) *SD, non SA, S, non CD, non CA, non U*

   (d) *SD, SA, S, CD, non CA, non U*

   (e) *SD, non SA, S, non CD, non CA, non U*

   (f) *SD, SA, S, CD, non CA, U*

   Réponse : SD, SA, S, non CD, non CA, non U

   On a SD et SA car BLOOP est un sous-ensemble du langage Java (qui vérifie SD et SA). Par contre il est impossible d'avoir U, CD et CA par le théorème d'Hoare-Allison.

3. *Si un formalisme de calculabilité possède la propriété CA (complétude algorithmique), alors il possède nécessairement la propriété CD (complétude des définitions).*

   Réponse : Vrai

   Supposons CA vrai. Si une fonction f est calculable par une machine de Turing M, la propriété CA nous donne un programme D de notre nouveau formalisme qui calcule cette même fonction. Donc f est calculable à l'aide de notre nouveau formalisme, qui vérifie donc CD.

4. *Si un formalisme de calculabilité possède les propriétés SD (soundness des définitions) et U (descriptions universelle), alors il possède nécessairement la propriété SA (soundness algorithmique).*

   Réponse : Vrai

   Par U, la fonction interpréteur du nouveau formalisme est calculable à l'aide de ce formalisme. Par SD, toute fonction calculable par ce nouveau formalisme est calculable (par une machine de Turing), donc en particulier l'interpréteur du nouveau formalisme l'est (ce qui correspond précisément à SA).

5. *Si un formalisme de calculabilité possède les propriétés SA (soundness algorithmique) et CD (complétude des définitions), alors il possède nécessairement la propriété U (descriptions universelle).*

   Réponse : Vrai

   Par SA, la fonction interpréteur du nouveau formalisme est calculable (à l'aide d'une machine de Turing). Par CD, toute fonction calculable est calculable par ce nouveau formalisme, donc en particulier l'interpréteur du nouveau formalisme l'est (ce qui correspond précisément à U).

6. *Les automates finis déterministes fournissent un formalisme de calculabilité qui possède la propriété SD (soundness des définitions).*

   Réponse : Vrai

   Toute fonction calculable par un automate fini déterministe peut être calculée par une machine de Turing.

7. *Les automates finis déterministes fournissent un formalisme de calculabilité qui possède la propriété CD (complétude des définitions).*

   Réponse : Faux

   Les automates finis ne permettent de calculer que des fonctions totales. Par le théorème d'Hoare-Allison, ils ne permettent donc pas de calculer toutes les fonctions calculables (par une machine de Turing), en particulier leur fonction interpréteur.

8. *Les automates finis déterministes fournissent un formalisme de calculabilité qui possède la propriété U (description universelle).*

   Réponse : Faux

   Les automates finis ne permettent de calculer que des focntions totales. Par le théorème d'Hoare-Allison, ils ne permettent donc pas de calculer toutes les fonctions calculables (par une machine de Turing), en particulier leur fonction interpréteur.

9. *Les automates finis déterministes fournissent un formalisme de calculabilité qui possède la propriété SA (soundness algorithmique).*

   Réponse : Vrai

   Il est tout à fait possible de simuler un automate fini déterministe quelconque en Java/Python ou à l'aide d'une machine de Turing. Autrement dit, la fonction interpréteur des automates finis est calculable.

10. *Les automates finis déterministes fournissent un formalisme de calculabilité qui possède la propriété CA (complétude algorithmique).*

    Réponse : Faux

    Puisque ce formalisme ne vérifie pas CD, il est impossible qu'il vérifie CA (puisque CA $\Rightarrow$ CD).