UCLouvain

École polytechnique de Louvain

LINFO 1123

Calculabilité, logique et complexité

Solutions des exercices

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

1 Ensembles énumérables et non-énumérables

1. *Answer the following questions in the given order.*

   (a) *True or false : a set X is countable iff $\mathcal{P}(X)$ (the set of all the subsets of X, the empty set and X itself included) is countable.*

   Réponse : Faux

   Un contre-exemple est donné par $\mathbb{N}$. En effet, il y a une bijection $g : \mathcal{F}(\mathbb{N}, \{0,1\}) \to \mathcal{P}(\mathbb{N})$ entre l'ensemble $\mathcal{F}(\mathbb{N}, \{0,1\})$ des fonctions de $\mathbb{N}$ dans $\{0,1\}$ avec l'ensemble $\mathcal{P}(\mathbb{N})$ des sous-ensembles de $\mathbb{N}$:

   $$(f : \mathbb{N} \to \{0,1\}) \overset{g}{\longmapsto} \{n \in \mathbb{N} \mid f(n) = 1\}$$

   Or, nous savons déjà que l'ensemble $\mathcal{F}(\mathbb{N}, \{0,1\})$ n'est pas énumérable, donc l'ensemble $\mathcal{P}(\mathbb{N})$ ne peut pas l'être non plus.

   (b) *Show Cantor's Theorem : if X is a set, then $\mathcal{P}(X)$ has never the same cardinality as X.*

   Réponse : Supposons par l'absurde qu'il existe une bijection $f : X \to \mathcal{P}(X)$. Posons l'ensemble :

   $$E = \{x \in X \mid x \notin f(x)\}$$

   Pour chaque $x \in X$, $f(x)$ est un sous-ensemble de X. La définition de E fait donc bien sens et E est un sous-ensemble de X.

   Puisque f est une bijection, notons $g : \mathcal{P}(X) \to X$ son unique fonction inverse. Puisque $E \in \mathcal{P}(X)$, l'élément $e = g(E)$ est bien un élément de X. L'élément e doit donc soit se trouver dans E, soit ne pas se trouver dans E :

   - Supposons qu'il se trouve dans E, alors par définition de E on a que $e \notin f(e) = E$, ce qui est absurde.
   - De même, supposons qu'il ne se trouve pas dans E, alors par définition de E on a que $e \in f(e) = E$, ce qui est absurde.

   En conclusion, e est un élément de X qui n'est ni dans E ni dans $X \backslash E$. C'est absurde. Donc notre supposition de départ, celle selon laquelle la bijection f peut exister, ne peut être vraie.

   (c) *If X is finite, what is the cardinality of $\mathcal{P}(X)$ with respect to the cardinality of X ?*

   Réponse : Notons $|X|$ la cardinalité de l'ensemble fini X (c'est-à-dire son nombre d'éléments). Alors le nombre de sous-ensembles possibles de X est :

   - le nombre de sous-ensembles de X à 0 éléments (il n'y en a qu'un : l'ensemble vide) ;
   - plus le nombre de sous-ensembles de X à un élément (il y en a exactement $|X|$) ;
   - plus le nombre de sous-ensembles de X à 2 éléments (pour construire un tel sous-ensemble, il faut choisir 2 éléments distincts parmi $|X|$, il y en a donc $C^2_{|X|}$) ;
   - plus le nombre de sous-ensembles de X à 3 éléments (pour construire un tel sous-ensemble, il faut choisir 3 éléments distincts parmi $|X|$, il y en a donc $C^3_{|X|}$) ;
   - et ainsi de suite jusqu'à l'unique sous-ensemble de X qui contient $|X|$ éléments : l'ensemble X lui-même.

   En conclusion, il y a donc :

   $$C^0_{|X|} + C^1_{|X|} + C^2_{|X|} + C^3_{|X|} + \cdots + C^{|X|-1}_{|X|} + C^{|X|}_{|X|}$$

   sous-ensembles possibles de X, ce qui est égal à (formule de combinatoire bien connue) $2^{|X|}$. La cardinalité de $\mathcal{P}(X)$ est donc $2^{|X|}$ (raison pour laquelle l'ensemble des sous-ensembles d'un ensemble est parfois noté 2^X, même quand X n'est pas nécesssairement fini).

2. *Using a cardinality argument, show that there are functions that are not computable by a Python program.*

   Réponse : L'ensemble des programmes peut être vu comme l'ensemble des mots finis réalisés à partir d'un alphabet fini, ce que nous savons déjà être énumérable. Or, nous savons également que l'ensemble $\mathcal{F}(\mathbb{N}, \{0,1\})$ n'est pas énumérable. Il est donc impossible qu'il existe au moins un programme pour chaque fonction de $\mathbb{N}$ dans $\{0,1\}$, et il existe donc certainement au moins une infinité de fonctions non calculables (à l'aide d'un programme python).

3. *If A_i are countable sets :*

   (a) *Prove that $A_1 \times A_2$ is a countable set.*

   Réponse : Soient $f : \mathbb{N} \to A_1$ et $g : \mathbb{N} \to A_2$ deux bijections. Pour construire une bijection $h : \mathbb{N} \to A_1 \times A_2$, il suffit de considérer la fonction qui envoie n sur le n-ième élément obtenu en parcourant le tableau ci-dessous (en suivant les diagonales descendantes de droite à gauche) :

	$f(0)$	$f(1)$	$f(2)$	$f(3)$	...
$g(0)$	$(f(0), g(0))$	$(f(1), g(0))$	$(f(2), g(0))$	$(f(3), g(0))$	
$g(1)$	$(f(0), g(1))$	$(f(1), g(1))$	$(f(2), g(1))$	$(f(3), g(1))$	
$g(2)$	$(f(0), g(2))$	$(f(1), g(2))$	$(f(2), g(2))$	$(f(3), g(2))$	
$g(3)$	$(f(0), g(3))$	$(f(1), g(3))$	$(f(2), g(3))$	$(f(3), g(3))$	
$\vdots$					

   (b) *Prove that $\bigcup_{i=0}^{\infty} A_i$ is a countable set.*

   Réponse : Soient $f_i : \mathbb{N} \to A_i$ des bijections. Pour constuire une bjection $h : \mathbb{N} \to \bigcup_{i=0}^{\infty} A_i$, il suffit de considérer la fonction qui envoie n sur le n-ième élément obtenu en parcourant le tableau ci-dessous (en suivant les diagonales descendantes de droite à gauche et en négligeant les répétitions) :

	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	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	
$\vdots$					

4. *Write a program that lists these sets :*

   (a) $\mathbb{Z}$.

   Réponse : Supposons qu'on ait une représentation des nombres entiers avec une infinité de bits (i.e. `sys.maxint` $= \infty$).

```
def enum_Z():
  print(0)
  n = 0
  while True:
    n += 1
    print(n)
    print(-n)
```

   (b) $\{a, b, c\}^*$ *(i.e. all words formed with the alphabet* $\{a, b, c\}$*).*

   Réponse : Supposons qu'on ait une mémoire infinie.

```
def enum_abc():
  alphabet = {"a", "b", "c"}    # the alphabet we are using
  print("")                     # the empty word
  list = [""]                   # a list that contains all the words
                                # of length 0
  while True:
    new_list = []
    for word in list:
      for symbol in alphabet:
        print(word + symbol)    # every word of length n can
                                # be obtained as a word of length
                                # n-1 + one symbol of the alphabet
        new_list.append(word + symbol)  # we update our new list of
                                        # words of length n
    list = new_list             # we replace our list of words of
                                # length n-1 with our new list of
                                # words of length n
```

   (c) *The set of all Python programs.*

   Réponse : Supposons qu'on ait un compilateur de Python, dénoté par `C`, qui renvoie 1 si un mot donné est un programme Python valide, 0 sinon. Supposons également qu'on ait une fonction `get_symbols` qui renvoie l'ensemble des symboles de l'encodage utilisé (e.g. UTF-8) autorisés dans un programme python.

```
def enum_py():
  alphabet = get_symbols()      # renvoie {"a", "b", " ", "\n", ...}
  list = [""]
  while True:
    new_list = []
    for string in list:
      for symbol in alphabet:
        if C(string + symbol) == 1: # this time, we have to check
          print(string + symbol)    # if the string is a valid Python
                                    # program before printing it
        new_list.append(string + symbol)
    list = new_list
```

2 Fonctions calculables et ensembles récursifs

1. *Let $X \subseteq \mathbb{N}$ and $f(n)$ a total function defined by :*

$$f(n) = \#\{x \in X \mid x < n\}$$

where $\#A$ is the cardinal of A. Show that X is recursive iff f is computable.

Réponse : Commençons par supposer que X est récursif. Alors il existe un programme P_X qui décide X. Alors le programme suivant calcule la fonction f :

$$P_f(n) \equiv \left[\begin{array}{l} \texttt{result} = 0 \\ \texttt{for } k \texttt{ in range}(n): \\ \quad \texttt{result += } P_X(k) \\ \texttt{return result} \end{array}\right.$$

Supposons maintenant que la fonction f est calculable. Alors il existe un programme P_f qui calcule f. En l'utilisant, on peut construire un programme qui décide l'ensemble X :

$$P_X(n) \equiv \texttt{return } P_f(n+1) - P_f(n)$$

2. *True or false ?*

   (a) *If the domain of a function is finite, then the function is computable.*

   Réponse : Vrai
   Soit f une fonction dont le domaine est fini. Soient $n_1, n_2, \ldots, n_k$ tous les éléments de son domaine. Soient respectivement $m_1, m_2, \ldots, m_k$ les images de ces nombres par la fonction f. Alors le programme ci-dessous calcule la fonction f.

$$P_f(n) \equiv \left[\begin{array}{l} \texttt{if } n == n_1: \\ \quad \texttt{return } m_1 \\ \texttt{elif } n == n_2: \\ \quad \texttt{return } m_2 \\ \qquad \vdots \\ \texttt{elif } n == n_k: \\ \quad \texttt{return } m_k \\ \texttt{else}: \\ \quad \texttt{while True}: \texttt{pass} \end{array}\right.$$

   (b) *If a function is computable, then its domain is recursive.*

   Réponse : Faux
   Contre-exemple : le programme ci-dessous calcule une fonction dont le domaine est l'ensemble K (qui n'est pas récursif) :

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

   (c) *A function whose table can be defined in a finite way is necessarily computable.*

   Réponse : Faux
   Contre-exemple : la fonction dont la table est décrite à l'aide de la phrase finie suivante n'est pas calculable :

$\ll$ La fonction f définie sur $\mathbb{N}$ telle que pour tout $n \in \mathbb{N}$, $f(n) = 1$ si $n \in K$ et 0 si $n \notin K$. $\gg$

3. *The Matiyasevich theorem is a famous theorem in number theory which gives an answer to Hilbert's tenth problem. The formulation of the theorem is quite simple :*

   A subset of $\mathbb{N}$ *is diophantine iff it is recursively enumerable.*

   (Any subset of $\mathbb{N}$ *is called diophantine iff it is of the form* $E_D = \{a \in \mathbb{N} \mid \exists x_1, \ldots, x_k \in \mathbb{N} : D(a, x_1, \ldots, x_k) = 0\}$ *where* $D(a, x_1, \ldots, x_k)$ *is a fixed polynomial with one parameter* a *and with* k *variables* $x_1, \ldots, x_k$*.)*

   Prove half of the theorem : show that if a set X *is diophantine, then it is recursively enumerable.*

   Hint. *Write a program which proves that the set of perfect squares, which is a diophantine set (*$\{a \in \mathbb{N} \mid \exists x \in \mathbb{N} : x^2 - a = 0\}$*), is recursively enumerable but does not prove that it is recursive (even though it is). Then do the same for the set of sums of two perfect squares.*

   Réponse : Soit E_D un ensemble diophantien et soit $D(a, x_1, \ldots, x_k)$ le polynôme associé. Le programme ci-dessous permet de justifier que E_D est récursivement énumérable :

$$
P_{E_D}(n) \equiv \left\lceil
\begin{array}{l}
i_1 = 0 \\
i_2 = 0 \\
\quad \vdots \\
i_k = 0 \\
\texttt{while True}: \\
\quad \texttt{while } i_2 < i_1 : \\
\quad\quad \texttt{while } i_3 < i_2 : \\
\quad\quad\quad \ddots \\
\quad\quad\quad\quad \texttt{while } i_k < i_{k-1} : \\
\quad\quad\quad\quad\quad \texttt{if } D(a, i_1, \ldots, i_k) == 0 : \\
\quad\quad\quad\quad\quad\quad \texttt{return } 1 \\
\quad\quad\quad\quad\quad i_k = i_k + 1 \\
\quad\quad\quad\quad i_k = 0 \\
\quad\quad\quad\quad i_{k-1} = i_{k-1} + 1 \\
\quad\quad\quad \ddots \\
\quad\quad i_3 = 0 \\
\quad\quad i_2 = i_2 + 1 \\
\quad i_2 = 0 \\
\quad i_1 = i_1 + 1
\end{array}
\right.
$$

4. *Show that all monotonically decreasing total functions* $f : \mathbb{N} \to \mathbb{N}$ *are computable.*

   Réponse : Soit une fonction $f : \mathbb{N} \to \mathbb{N}$ qui est décroissante. Puisque que $\mathbb{N}$ est discret et est borné inférieurement par 0, f a nécessairement une image finie. Soient $m_1, m_2, \ldots, m_k$ les éléments de cet image.

   Puisque que f est décroissante :

   - Pour tout $i \in \mathbb{N}$ avec $1 \leq i \leq k-1$, il existe $a_i, b_i \in \mathbb{N}$ avec $a_i \leq b_i$ tels que tout élément de $f^{-1}(m_i)$ est compris entre a_i et b_i.
   - Pour $i = k$ il existe a_k tel que tout élément de $f^{-1}(m_k)$ est plus grand ou égal à a_k.

Dès lors, le programme suivant calcule la fonction f :

$$P_f(n) \equiv \left[\begin{array}{l} \texttt{if } a_1 \leq n \leq b_1 : \\ \quad \texttt{return } m_1 \\ \texttt{elif } a_2 \leq n \leq b_2 : \\ \quad \texttt{return } m_2 \\ \qquad \vdots \\ \texttt{elif } a_k \leq n : \\ \quad \texttt{return } m_k \end{array}\right.$$

3 Réduction et théorème d'Hoare-Allison

1. *Look at the Hoare-Allision diagonalization proof in the lecture slides. Why does the Hoare-Allison theorem not apply to a formalism that allows one to compute non-total functions ?*

   Réponse : Lorsqu'on modifie la fonction $diag(n) = interpret(n, n)$:

   $$diag\_mod(n) = interpret(n, n) + 1$$

   L'expression « $interpret(n, n) + 1$ » ne fait pas nécessairement toujours sens (si pour un certain $n \in \mathbb{N}$, $P_n(n)$ ne termine pas).

   Si on décide de forcer cette étape en posant $\perp + 1 = \perp$, c'est l'étape suivante de la démonstration qui ne fonctionne plus. En effet, l'égalité $diag\_mod(n) = interpret(n, n)+1$ n'est alors plus nécessairement une contradiction (si $interpret(d, d) = \perp$).

2. *Let L be a (non-trivial) programming language in which the function :*

   $$halt_L(n, x) = \begin{cases} 1 & \text{if } P_n \text{ stops on } x \\ 0 & \text{otherwise} \end{cases}$$

   is computable. Using the diagonalization, prove that the function $interpret_L(n, x)$ is not computable in L.

   Réponse : Supposons par l'absurde que la fonction $interpret_L(n, x)$ est calculable dans L. Soit $P_{interpret_L}(n, x)$ le programme qui calcule cette fonction. Comme pour la preuve du théorème d'Hoare-Allison, on réalise un tableau listant tous les programmes de L et toutes les entrées possibles et on sélectionne la diagonale :

   $$diag(n) = interpret_L(n, n)$$

   Cette fonction est calculable dans L puisque nous avons supposé (par l'absurde) que $interpret(n, x)$ est calculable dans L. Modifions cette fonction de la façon suivante :

   $$diag\_mod(n) = \begin{cases} interpret_L(n, n) + 1 & \text{si } halt_L(n, n) = 1 \\ 0 & \text{si } halt_L(n, n) = 0 \end{cases}$$

   Nous pouvons affirmer que cette fonction $diag\_mod(n)$ est calculable dans L car la fonction $halt_L(n, x)$ est calculable dans L (par hypothèse).

   Comme $diag\_mod(n)$ est calculable dans L, soit d un programme de L qui calcule cette fonction. Étudions la valeur de $diag\_mod(d)$:

   - Si $halt_L(d, d) = 1$, alors $diag\_mod(d) = interpret_L(d, d) + 1 = diag\_mod(d) + 1$, ce qui est absurde.
   - Si $halt_L(d, d) = 0$, alors $diag\_mod(d) = 0$. Mais $halt_L(d, d) = 0$ nous dit également que $P_d(d)$ ne termine pas, autrement dit que la fonction calculée par P_d (qui est $diag\_mod$) n'est pas définie en d, ce qui est absurde.

   En conclusion, on arrive à une contradiction dans tous les cas. Il ne peut donc être vrai que la fonction $interpret_L(n, x)$ est calculable dans L.

3. *Any complete formalism of computability must allow to compute its own interpreter. Given that the Python language is a complete formalism of computability, it implies that it is theoretically possible to compute the universal function of Python with Python. In practice, how would you proceed to write a program in Python which would compute this function ?*

Réponse : Un programme qui calcule cette fonction universelle n'est rien d'autre qu'un interpréteur de Python. Il est tout à fait possible de coder un interpréteur de Python en Python (`PyPy` est un exemple), même si c'est tout sauf simple en pratique. Plus explicitement, voici une version élémentaire d'un tel programme :

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

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.

4. *In order to prove the undecidability of a problem, we have so far used the diagonalization method. We now show a more practical method, called the* reduction method. *It is used to prove the undecidability (i.e. the non recursivity) of a set B, knowing the undecidability of the set A. Its principle is simple :*

   1. *We build an algorithm P_A deciding A assuming the existence of an algorithm P_B deciding B. Algorithm P_A can thus use P_B as a subroutine. We say that the decidability of A is* reduced *to the decidability of B.*
   2. *We conclude that B is not decidable, since if B were decidable, then A would also be decidable, which is impossible by hypothesis.*

   Let $H = \{(n, k) \mid P_n(k)$ terminates$\}$. Using the reduction method, prove that the following sets are undecidable because H is undecidable.

   (a) $S_1 = \{n \mid P_n(0)$ terminates$\}$

   Réponse : Supposons par l'absurde que S_1 est récursif. Soit $P_{S_1}(n)$ un programme qui décide S_1. Alors le programme suivant décide $HALT$:

   $$P_H(n, k) \equiv \texttt{return } P_{S_1}\Big(P(x) \equiv \texttt{return } P_n(k)\Big)$$

   C'est absurde. Il ne peut donc être vrai que S_1 est récursif.

   (b) $S_2 = \{n \mid \varphi_n(k) = k \ \forall k\}$

   Réponse : Supposons par l'absurde que S_2 est récursif. Soit $P_{S_2}(n)$ un programme qui décide S_2. Alors le programme suivant décide $HALT$:

   $$P_H(n, k) \equiv \texttt{return } P_{S_2}\left(P(x) \equiv \left[\begin{array}{l} P_n(k) \\ \texttt{return } x \end{array}\right)\right.$$

   C'est absurde. Il ne peut donc être vrai que S_2 est récursif.

   (c) $S_3 = \{(n, m) \mid \varphi_n = \varphi_m\}$

   Réponse : Supposons par l'absurde que S_3 est récursif. Soit $P_{S_3}(n, m)$ un programme qui décide S_3. Alors le programme suivant décide $HALT$:

   $$P_H(n, k) \equiv \texttt{return } P_{S_3}\left(P_a(x) \equiv \texttt{return } 1\,,\ P_b(y) \equiv \left[\begin{array}{l} P_n(k) \\ \texttt{return } 1 \end{array}\right)\right.$$

   C'est absurde. Il ne peut donc être vrai que S_3 est récursif.

(d) $S_4 = \{n \mid \varphi_n \text{ is a non-total function}\}$

Réponse : Supposons par l'absurde que S_4 est récursif. Soit $P_{S_4}(n)$ un programme qui décide S_4. Alors le programme suivant décide $HALT$:

$$P_H(n,k) \equiv \texttt{return } 1 - P_{S_4}\Big(P(x) \equiv \texttt{return } P_n(k)\Big)$$

C'est absurde. Il ne peut donc être vrai que S_4 est récursif.

(e) $S_5 = \{(n,m) \mid \forall k : \varphi_n(k) \neq \varphi_m(k)\}$

Réponse : Supposons par l'absurde que S_5 est récursif. Soit $P_{S_5}(n)$ un programme qui décide S_5. Alors le programme suivant décide $HALT$:

$$P_H(n,k) \equiv \texttt{return } P_{S_5}\left(P_a(x) \equiv \left[\begin{array}{l}\texttt{while True:}\\ \quad\texttt{pass}\end{array}\right., \; P_b(y) \equiv \texttt{return } P_n(k)\right)$$

C'est absurde. Il ne peut donc être vrai que S_5 est récursif.

4 Théorème de Rice

1. *Let* $A = \{i \mid \exists x, y : \varphi_i(x) = \varphi_i(y) \neq \bot, x \neq y\}$ *be the set of programs that halts with returning the same value for at least two different inputs. Using Rice's theorem, show that* A *is not recursive.*

   Réponse :

   1. On vérifie que $A \neq \emptyset$ et $A \neq \mathbb{N}$:

   $$\varphi_1(x) \triangleq 42 \quad \Longrightarrow \quad 1 \in A \quad \Longrightarrow \quad A \neq \emptyset$$

   $$\varphi_2(x) \triangleq x \quad \Longrightarrow \quad 2 \in \overline{A} \quad \Longrightarrow \quad A \neq \mathbb{N}$$

   2. On montre qu'aucun programme dans A ne calcule la même fonction qu'un programme dans $\overline{A}$:

   - $\forall i \in A$, on a :
   $$\exists x, y : x \neq y \wedge \varphi_i(x) = \varphi_i(y) \wedge \varphi_i(x) \neq \bot \wedge \varphi_i(y) \neq \bot$$
   - $\forall j \in \overline{A}$, on a :
   $$\forall x, y : x \neq y \Rightarrow \varphi_j(x) \neq \varphi_j(y) \vee \varphi_j(x) = \bot \vee \varphi_j(y) = \bot$$
   - Par conséquent, $\forall i \in A, \forall j \in \overline{A}$, on a :
   $$\exists x, y : \ x \neq y \wedge \varphi_i(x) = \varphi_i(y) \wedge (\varphi_j(x) \neq \varphi_j(y) \vee \varphi_i(x) \neq \varphi_j(x) \vee \varphi_i(y) \neq \varphi_j(y))$$

     Ce qui implique que $\varphi_i \neq \varphi_j$.

   Par le théorème de Rice, A n'est donc pas récursif.

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

2. *Let* $A \subseteq \mathbb{N}$, $A \neq \mathbb{N}$ *and* $A \neq \emptyset$. *If there exists* $i \in A$ *and* $j \in \mathbb{N} \backslash A$ *such that* $\varphi_i = \varphi_j$, *can we say that* A *is recursive ? Prove your answer.*

   Réponse : Faux

   Contre-exemple :

   $$A = \{i \mid \forall x : P_i(x) \text{ s'arrête et } P_i \text{ a un nombre impair d'instructions}\}$$

   Soient les programmes :

   $$P_1(x) \equiv \texttt{return x} \qquad P_2(x) \equiv \left[\begin{array}{l} \texttt{sleep(1)} \\ \texttt{return x} \end{array} \right.$$

   On a bien $A \neq \emptyset$, $A \neq \mathbb{N}$ et $\exists i \in A, j \in \overline{A} : \varphi_i = \varphi_j$. Pourtant, A n'est pas récursif. En effet, supposons par l'absurde qu'il le soit. Alors on a un programme $P_A(n)$ qui décide A. Alors, on peut l'utiliser pour décider $HALT$:

   $$P_{HALT}(n,k) \equiv \left[\begin{array}{l} \texttt{if } P_n \texttt{ has an odd number of instructions :} \\ \quad \texttt{write } P_q(y) \equiv \left[\begin{array}{l} \texttt{return } P_n(k) \end{array} \right. \\ \texttt{else :} \\ \quad \texttt{write } P_q(y) \equiv \left[\begin{array}{l} \texttt{sleep(1)} \\ \texttt{return } P_n(k) \end{array} \right. \\ \texttt{return } P_A(q) \end{array} \right.$$

   C'est absurde. Donc il ne peut être vrai que A est récursif.

3. *Let $A \subseteq \mathbb{N}$ not recursive ($A \neq \mathbb{N}$ and $A \neq \emptyset$). Can we say that $\forall i \in A$ and $\forall j \in \mathbb{N} \backslash A$, $\varphi_i \neq \varphi_j$? Prove your answer.*

   Réponse : Faux

   Contre-exemple :

   $$A = \{i \,|\, \forall x : P_i(x) \text{ s'arrête et } P_i \text{ a un nombre impair d'instructions}\}$$

   n'est pas récursif (ce qui implique $A \neq \emptyset$ et $A \neq \mathbb{N}$). Pourtant $\exists i \in A, j \in \overline{A} : \varphi_i = \varphi_j$.

4. *Let $A = \{i \,|\, P_i(x)$ halts for every input $x\}$.*

   (a) *Is A recursive ? Why ?*

   Réponse : A n'est pas récursif. En effet :

   $$\forall i \in A : \forall x, \varphi_i(x) \neq \perp$$

   $$\forall j \in \overline{A} : \exists x, \varphi_j(x) = \perp$$

   $$\implies \varphi_i \neq \varphi_j$$

   Par le théorème de Rice, A n'est pas récursif (car $A \neq \emptyset$ et $A \neq \mathbb{N}$).

   (b) *Is A recursively enumerable ? Why ?*
   Hint. *Use the reduction method.*

   Réponse : A n'est pas récursivement énumérable car on peut réduire $\overline{HALT}$ (qui n'est pas récursivement énumérable) à A. Un exemple de réduction est donné par :

$$P_{\overline{H}_{re}}(n,k) \equiv \left[\begin{array}{l} \texttt{write } P_q(x) \equiv \left[\begin{array}{l} \texttt{do } x \texttt{ instructions of } P_n(k) \\ \texttt{if } P_n(k) \texttt{ has terminated:} \\ \quad \texttt{while true: pass} \\ \texttt{else:} \\ \quad \texttt{return } 1 \end{array}\right. \\ \texttt{return } P_{A_{re}}(q) \end{array}\right.$$

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

1. *The S property is defined as follows :*

$$\forall k\ \exists S \text{ total \& computable} : \varphi_k(x,y) = \varphi_{S(x)}(y)$$

Prove that the S property is a particular case of S-m-n (i.e. prove that S-m-n implies S for $m = n = 1$).

Réponse : Par la propriété S-m-n, on a :

$$\exists S \text{ total calculable } \forall k : \varphi_k(x,y) = \varphi_{S(k,x)}(y)$$

ce qui implique que :

$$\forall k\ \exists S \text{ total calculable} : \varphi_k(x,y) = \varphi_{S(k,x)}(y)$$

En utilisant S-m-n pour S et en renommant $S'(S,k)$ en k', on obtient :

$$S(k,x) = \varphi_S(k,x) = \varphi_{S'(S,k)}(x) = \varphi_{k'}(x) = S''(x)$$

En conclusion :

$$\forall k\ \exists S'' \text{ total calculable} : \varphi_k(x,y) = \varphi_{S''(x)}(y)$$

S est donc bien un cas particulier de S-m-n.

2. *Using the fixed point theorem, show that there exists a program P_n such that P_n terminates only for input n.*

Hint. *Use the function $g(n,x) = 1$ if $x = n$, $\perp$ otherwise together with the S property*

Réponse : Soit la fonction :

$$g(n,x) = \begin{cases} 1 & \text{si } n = x \\ \perp & \text{sinon} \end{cases}$$

qui est calculable. Par la propriété S :

$$\exists S \text{ total calculable} : \varphi_g(n,x) = \varphi_{S(n)}(x)$$

Par le point fixe, $\exists n : \varphi_n = \varphi_{S(n)}$, ce qui implique que :

$$\exists n : \varphi_n(x) = \varphi_{S(n)}(x) = \varphi_g(n,x)$$

Vu que g est calculable, P_n existe.

3. *Using the fixed point theorem, show that there exists a program P_n that always outputs n (i.e. that prints its source code).*

Réponse : Soit la fonction $g(n,x) = n$, qui est calculable. Par la propriété S :

$$\exists S \text{ total calculable} : \varphi_g(n,x) = \varphi_{S(n)}(x)$$

Par le point fixe, $\exists n : \varphi_n = \varphi_{S(n)}$, ce qui implique que :

$$\exists n : \varphi_n(x) = \varphi_{S(n)}(x) = \varphi_g(n,x)$$

Vu que g est calculable, P_n existe.

4. *Prove Rice's theorem using the fixed point theorem.*

   Hint. *Define the function $f(x) = i$ if $x \in A$, j if $x \in \overline{A}$, with $i \in \overline{A}$ and $j \in A$.*

   Réponse : Soient $i \in A$ et $j \in \overline{A}$. On définit :

$$f(x) = \begin{cases} j & \text{si } x \in A \\ i & \text{si } x \in \overline{A} \end{cases}$$

   En supposant que A est récursif, on a f qui est totale et calculable. De plus, on sait que $A \neq \emptyset \neq \overline{A}$. Par le point fixe, $\exists k : \varphi_k = \varphi_{f(k)}$. On a 2 cas :

   - Si $k \in A$, alors $\varphi_k = \varphi_j$
   - Si $k \in \overline{A}$, alors $\varphi_k = \varphi_i$

   On a donc bien $\exists i \in A, \exists j \in \overline{A}$ tel que $\varphi_i = \varphi_j$.

5. *Prove that $K = \{n \in \mathbb{N} \mid \varphi_n(n) \neq \perp\}$ is not recursive using the fixed point theorem.*

   Réponse : Supposons K récursif. Soient les fonctions calculables $\varphi_i(x) = x$ et $\varphi_j(x) = \perp$. On définit

$$f(x) = \begin{cases} j & \text{si } x \in K \\ i & \text{si } x \in \overline{K} \end{cases}$$

   qui est totale et calculable. Par le point fixe, $\exists k : \varphi_k = \varphi_{f(k)}$. On a 2 cas :

   - Si $k \in K$, on a $\varphi_k(k) \neq \perp$ et $f(k) = j$, ce qui implique que $\varphi_k(k) = \varphi_{f(k)}(k) = \varphi_j(k) = \perp$.
   - Si $k \notin K$, on a $\varphi_k(k) = \perp$ et $f(k) = i$, ce qui implique que $\varphi_k(k) = \varphi_{f(k)}(k) = \varphi_i(k) = k$.

   On obtient une contradiction dans les 2 cas, donc K n'est pas récursif.

6 Automates

1. *Let* $A = \{a \in \mathbb{N} \,|\, \exists x \in \mathbb{N} : -x^5 + x + a = 0\}$. *Show that A is ND-recursive. Is A also recursive ?*

   Réponse : Notons d'abord que si A est ND-récursif, alors il est automatiquement récursif.

   Pour justifier qu'il est ND-récursif, il suffit de considérer le programme qui reçoit en entrée le coefficient a, attribue au hasard une valeur à la variable x et renvoie 1 si la formule est alors égale à 0, et 0 sinon.

   Cependant, la fonction `choose` nécessite une borne supérieure pour pouvoir tirer une valeur au hasard. On remarque que :

   - si $a = 0$, le polynôme admet comme racines 0 et 1 ;
   - si $a \neq 0$, alors $\forall x \in \mathbb{N}$ avec $x > a$, on a $-x^5 + x + a < 0$.

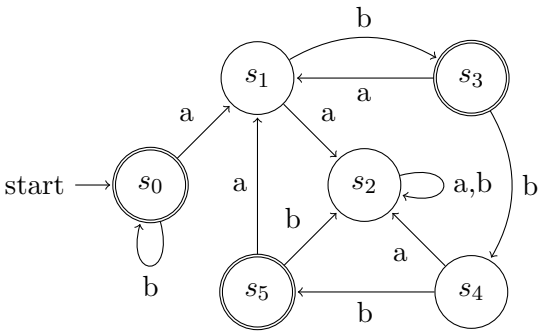
   Dès lors, le programme suivant prouve que A est ND-récursif :

$$P_{A_{ND}}(a) \equiv \left[\begin{array}{l} x = \texttt{choose}(a+1) \\ \texttt{if } -x^5 + x + a = 0 : \\ \quad \texttt{return } 1 \\ \texttt{else} : \\ \quad \texttt{return } 0 \end{array}\right.$$

2. *For the following languages, construct a deterministic finite automaton that accepts this language.*

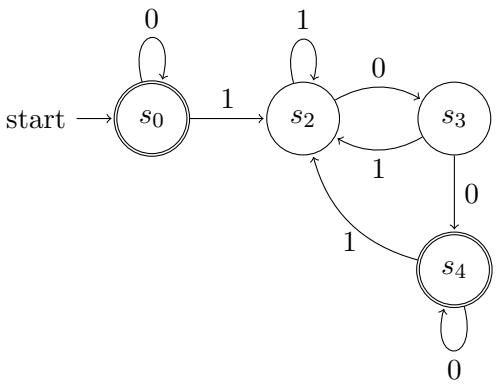
   (a) *Alphabet :* $\Sigma = \{\text{a}, \text{b}\}$
   Language : $\{w \in \Sigma^* \,|\, \text{each a in } w \text{ is followed by exactly one or three b}\}$

   Réponse :



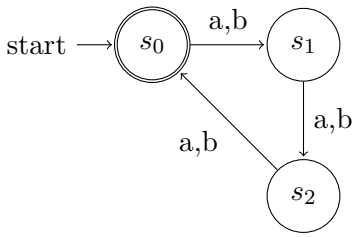
   (b) *Alphabet :* $\Sigma = \{0, 1\}$
   Language : $\{w \in \Sigma^* \,|\, \text{the decimal value of } w \text{ is divisible by 4}\}$

   Réponse :



(c) *Alphabet :* $\Sigma = \{\text{a}, \text{b}\}$
Language : $\{w \in \Sigma^* \mid \text{the length of } w \text{ is divisible by } 3\}$

Réponse :



3. *Suppose we have two languages M and L accepted by some finite automata and their corresponding diagrams. Show by construction that the following languages are also accepted by a finite automaton :*

(a) $\overline{L}$

Réponse : On transforme les états acceptants en états non acceptants et inversement.

(b) $L \cdot M$

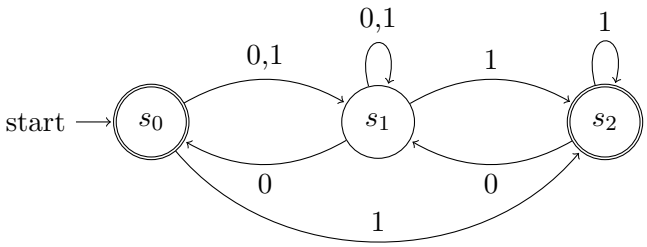
Réponse : On enlève le statut d'état acceptant aux états acceptants de l'automate qui décide L. On enlève le statut d'état initial à l'état initial de l'automate qui décide M. On crée des ε-transitions de tous les anciens états acceptants de l'automate qui décide L vers l'ancien état initial de M.

(c) $L \cup M$

Réponse : On crée un nouvel état initial avec une ε-transition vers chacun des états initiaux des deux automates fournis.

4. *Transform the following non-deterministic automata in finite deterministic automata accepting the same languages.*

(a)

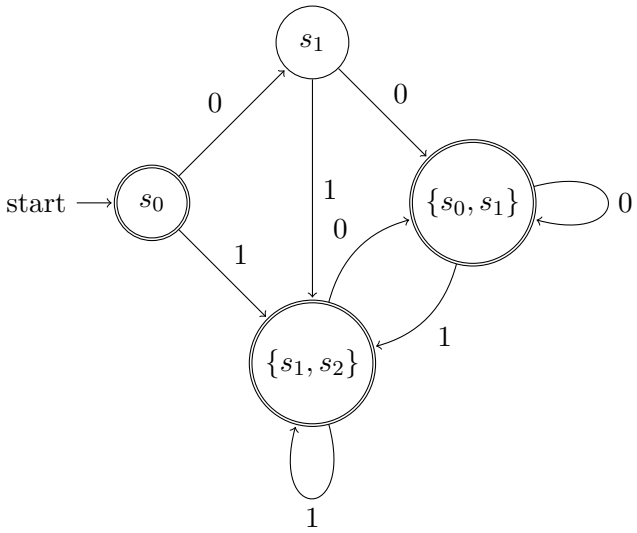


Réponse : On transforme la table de transition de la manière suivante :

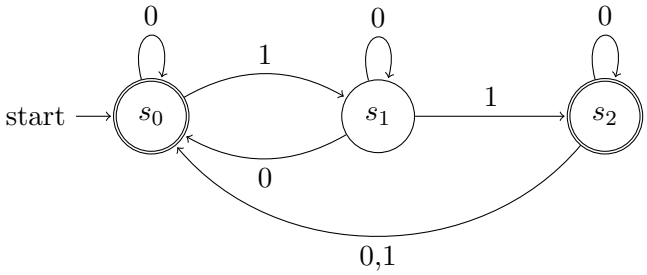
	0	1
s_0	s_1	s_1, s_2
s_1	s_0, s_1	s_1, s_2
s_2	s_1	s_2

	0	1
s_0	s_1	$\{s_1, s_2\}$
s_1	$\{s_0, s_1\}$	$\{s_1, s_2\}$
s_2	s_1	s_2
$\{s_0, s_1\}$	$\{s_0, s_1\}$	$\{s_1, s_2\}$
$\{s_1, s_2\}$	$\{s_0, s_1\}$	$\{s_1, s_2\}$

On remarque que l'état s_2 est inaccessible, on peut donc l'ignorer dans le diagramme.
L'automate correspondant est :



(b)

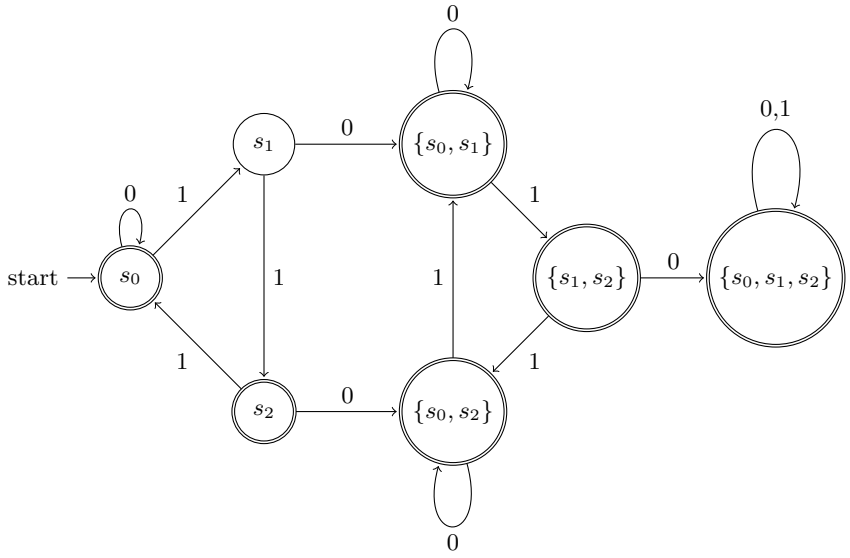


Réponse : On transforme la table de transition de la manière suivante :

	0	1
s_0	s_0	s_1
s_1	s_0, s_1	s_2
s_2	s_0, s_2	s_0

	0	1
s_0	s_0	s_1
s_1	$\{s_0, s_1\}$	s_2
s_2	$\{s_0, s_2\}$	s_0
$\{s_0, s_1\}$	$\{s_0, s_1\}$	$\{s_1, s_2\}$
$\{s_0, s_2\}$	$\{s_0, s_2\}$	$\{s_0, s_1\}$
$\{s_1, s_2\}$	$\{s_0, s_1, s_2\}$	$\{s_0, s_2\}$
$\{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$

On remarque que cette fois-ci, la nouvelle table de transition a beaucoup plus de rangées. Elle comprend même tous les sous-ensembles d'états de l'automate non-déterministe. Pour un automate non-déterministe à n états, l'automate déterministe correspondant peut donc avoir jusqu'à 2^n états.
L'automate correspondant est :


start
s_0
s_1
s_2
$\{s_0, s_1\}$
$\{s_0, s_2\}$
$\{s_1, s_2\}$
$\{s_0, s_1, s_2\}$
0
1
0,1

7 Machines de Turing

1. *For each of the following functions, build a Turing machine, using* $\Sigma = \{0, 1\}$ *:*

   (a) $f(n) = n \times 2$

   Réponse : $\Gamma = \{\mathrm{B}, 0, 1\}$

state	symbol	state	movement	symbol
start	0	start	$\rightarrow$	0
start	1	start	$\rightarrow$	1
start	B	stop	$\downarrow$	0

   (b) $f(n) = \mathrm{not}(n)$ *(f inverts the bits of n)*

   Réponse : $\Gamma = \{\mathrm{B}, 0, 1\}$

state	symbol	state	movement	symbol
start	0	start	$\rightarrow$	1
start	1	start	$\rightarrow$	0
start	B	stop	$\downarrow$	B

2. *Suppose we have two Turing machines A and B with* $\Sigma = \{\mathrm{a}, \mathrm{b}\}$*. Their starting and halting states are* (s_A, h_A) *and* (s_B, h_B)*, respectively. Explain how to combine A and B into a new Turing machine implementing the following pseudo-codes.*

   Remark. *In the pseudocode, "*`if` a*" means "if* a *is the symbol under the head", and "*`not` a*" means "the symbol under the head is not* a*"; none of which consumes a symbol.*

   (a) $\left[\begin{array}{l} \texttt{if a:}\ A \\ \texttt{else:}\ B \end{array}\right.$

   Réponse :

state	symbol	state	movement	symbol
start	a	s_A	$\downarrow$	a
start	b	s_B	$\downarrow$	b
start	B	s_B	$\downarrow$	B
h_A	a	stop	$\downarrow$	a
h_A	b	stop	$\downarrow$	b
h_A	B	stop	$\downarrow$	B
h_B	a	stop	$\downarrow$	a
h_B	b	stop	$\downarrow$	b
h_B	B	stop	$\downarrow$	B

   (b) `while not` a: A

   Réponse :

state	symbol	state	movement	symbol
start	a	stop	$\downarrow$	a
start	b	s_A	$\downarrow$	b
start	B	s_A	$\downarrow$	B
h_A	a	stop	$\downarrow$	a
h_A	b	s_A	$\downarrow$	b
h_A	B	s_A	$\downarrow$	B

3. *Are those Turing machines equivalent to the standard model of Turing machine ? Explain.*

   (a) *"Big jumps" : this Turing machine can move its head of n cells to the left and to the right. The transition function is thus :*

   $$S \times \Gamma \longrightarrow S \times (\{L, R\} \times \mathbb{N}) \times \Gamma$$

   Réponse : Vrai

   Il y a moins d'instructions à exécuter (on a donc une meilleur efficacité), mais on conserve la même puissance. En effet, on peut remplacer :

s	x	s'	$\rightarrow_n$	x'

   par :

s	x	right_{n-1}	$\rightarrow$	x'
right_{n-1}	?	right_{n-2}	$\rightarrow$	?
		$\vdots$		
right_1	?	s'	$\rightarrow$	?

   (b) *"Online" : this Turing machine can only move to the right or stay at the same cell, and cannot move to the left. The transition function is thus :*

   $$S \times \Gamma \longrightarrow S \times \{R, \downarrow\} \times \Gamma$$

   Réponse : Faux

   Ce modèle est restrictif. Voici des exemples impossibles à résoudre avec cette machine de Turing :

   - Inverser le 1$^{\text{er}}$ bit si le dernier est 0
   - Ajouter 1 à un nombre binaire
   - L'exemple donné dans le quiz

4. *Is it possible to simulate any deterministic finite automaton with a Turing machine ? Explain.*

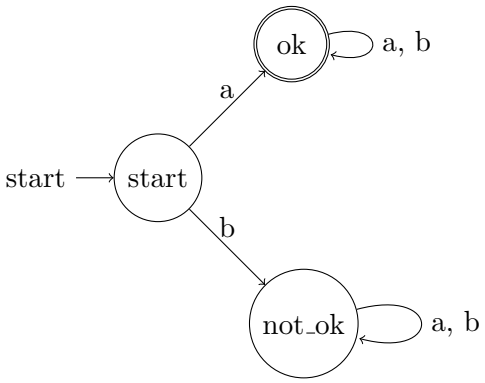
   Réponse : Vrai

   Il est toujours possible de simuler un automate avec une machine de Turing (car c'est un modèle plus puissant).

   Dans ces automates, les caractères de l'input sont lus progressivement : un à chaque transition d'état. Une fois tous les caractères lus, l'automate accepte ou rejette l'input en fonction de son état final. Une machine de Turing peut facilement simuler un tel comportement. Chaque état de l'automate équivaut à un état de la machine de Turing. Ensuite, pour chaque transition de l'automate, nous lisons la valeur courante et changeant d'état en accord avec celle-ci. Lors de cette transition, nous écrivons une valeur quelconque sur le ruban et passons à la valeur de droite. Un symbole blanc (B) est également nécessaire pour indiquer la fin de la gestion de l'input et la transition vers un état acceptant ou de rejet.

   Un exemple de transformation est présenté ci-dessous pour un automate vérifiant que l'input commence par la lettre a :

$\Sigma = \{a, b\}$



$\Gamma = \{B, a, b\}$

state	**symbol**	**state**	**mvmt**	**symbol**
start	a	ok	$\rightarrow$	a
start	b	not_ok	$\rightarrow$	b
start	B	reject	$\rightarrow$	B
ok	a	ok	$\rightarrow$	a
ok	b	ok	$\rightarrow$	b
ok	B	accept	$\rightarrow$	B
not_ok	a	not_ok	$\rightarrow$	a
not_ok	b	not_ok	$\rightarrow$	b
not_ok	B	reject	$\rightarrow$	B
accept	B	accept	$\downarrow$	B
reject	B	reject	$\downarrow$	B

8 Logique des propositions

1. *Formalisez les raisonnements suivants à l'aide de la logique des propositions, puis déterminez s'ils sont valides en vérifiant si les conclusions sont bien des conséquences logiques des prémisses.*

   (a) *Si je suis coupable, je dois être puni. Or, je dois être puni. Donc je suis coupable.*

   Réponse : Associons la variable propositionnelle A à l'affirmation "je suis coupable" et la variable propositionnelle B à l'affirmation "je dois être puni".
   Le raisonnement proposé correspond à affirmer que :

   $$(A \Rightarrow B) \wedge B \models A$$

   Or, l'interprétation qui rend A fausse et B vraie est telle que :

   i. La prémisse $(A \Rightarrow B) \wedge B$ est vraie car cette interprétation rend $(A \Rightarrow B)$ vraie et elle rend B vraie.
   ii. La conclusion A est fausse.

   En conclusion, on n'a **pas** $(A \Rightarrow B) \wedge B \models A$ et le raisonnement n'est pas valide.

   (b) *Si je suis coupable, je dois être puni. Or, je ne dois pas être puni. Donc je ne suis pas coupable.*

   Réponse : Le raisonnement proposé correspond à affirmer que :

   $$(A \Rightarrow B) \wedge \neg B \models \neg A$$

   L'interprétation qui rend A fausse et B fausse est telle que :

   i. La prémisse $(A \Rightarrow B) \wedge \neg B$ est vraie car cette interprétation rend $(A \Rightarrow B)$ vraie et elle rend $\neg B$ vraie.
   ii. La conclusion $\neg A$ est vraie.

   Toute autre interprétation rend la prémisse fausse ! En conclusion, $(A \Rightarrow B) \wedge \neg B \models \neg A$ et le raisonnement est valide.

   (c) *Si je suis coupable, je dois être puni. Or, je ne suis pas coupable. Donc je ne dois pas être puni.*

   Réponse : Le raisonnement proposé correspond à affirmer que :

   $$(A \Rightarrow B) \wedge \neg A \models \neg B$$

   Or, l'interprétation qui rend A fausse et B vraie est telle que :

   i. La prémisse $(A \Rightarrow B) \wedge \neg A$ est vraie car cette interprétation rend $(A \Rightarrow B)$ vraie et elle rend $\neg A$ vraie.
   ii. La conclusion $\neg B$ est fausse.

   En conclusion, on n'a **pas** $(A \Rightarrow B) \wedge \neg A \models \neg B$ et le raisonnement n'est pas valide.

2. *Voici quelques exemples de tautologies qui sont si importantes (d'une manière ou d'une autre) qu'elles portent un nom. En construisant les tables de vérités de ces formules, vérifiez qu'il s'agit bien de tautologies.*

   (a) *Principe du tiers exclu :* $(A \vee \neg A)$

   Réponse : Pour construire la table de vérité, on considère les différentes propositions impliquées dans la tautologie.

A	$\neg A$	$(A \vee \neg A)$
T	F	T
F	T	T

Cette formule étant vraie dans toutes ces interprétations, il s'agit bien d'une tautologie.

(b) *Loi de Morgan :* $\neg(A \vee B) \Leftrightarrow (\neg A \wedge \neg B)$

Réponse :

A	B	$(A \vee B)$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$(\neg A \wedge \neg B)$	$\neg(A \vee B) \Leftrightarrow (\neg A \wedge \neg B)$
T	T	T	F	F	F	F	T
F	T	T	F	T	F	F	T
T	F	T	F	F	T	F	T
F	F	F	T	T	T	T	T

(c) *Contraposition :* $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$

Réponse :

A	B	$(A \Rightarrow B)$	$\neg B$	$\neg A$	$(\neg B \Rightarrow \neg A)$	$(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$
T	T	T	F	F	T	T
F	T	T	F	T	T	T
T	F	F	T	F	F	T
F	F	T	T	T	T	T

(d) *Syllogisme :* $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$

Réponse :

A	B	C	$(A \Rightarrow B)$	$(B \Rightarrow C)$	$(A \Rightarrow B) \wedge (B \Rightarrow C)$	$(A \Rightarrow C)$	$(. \wedge .) \Rightarrow (A \Rightarrow C)$
T	T	T	T	T	T	T	T
F	T	T	T	T	T	T	T
T	F	T	F	T	F	T	T
F	F	T	T	T	T	T	T
T	T	F	T	F	F	F	T
F	T	F	T	F	F	T	T
T	F	F	F	T	F	F	T
F	F	F	T	T	T	T	T

3. *Pour toute formule de la logique des propositions, il existe une formule équivalente sous forme normale conjonctive. La forme normale conjonctive n'utilise que trois opérateurs logiques : la conjonction, la disjonction et la négation. Est-il possible de sélectionner seulement deux opérateurs logiques de tel sorte que toute formule de la logique des propositions admette une formule équivalente qui ne fait intervenir que ces deux opérateurs logiques ? Justifier.*

   Réponse : Oui, on peut par exemple se limiter aux opérateurs logiques $\neg$ et $\Rightarrow$. En effet, pour toute formules p et q :

   - $(p \vee q)$ est équivalent à $(\neg p \Rightarrow q)$
   - $(p \wedge q)$ est équivalent à $\neg(p \Rightarrow \neg q)$
   - $(p \Leftrightarrow q)$ est équivalent à $\neg((p \Rightarrow q) \Rightarrow \neg(q \Rightarrow p))$

4. *La conséquence logique peut-elle être vue comme une relation d'ordre sur les formules de la logique des propositions ? Plus précisément, est-elle :*

(a) *Réflexive : $p \models p$ est-il vrai pour toute formule p ?*

Réponse : Oui, car cela revient à se demander si pour toute formule p, la formule $(p \Rightarrow p)$ est une tautologie. Or, cette formule a deux interprétations possibles :

- Si p est vrai, alors $(p \Rightarrow p)$ est vrai.
- Si p est faux, alors $(p \Rightarrow p)$ est vrai.

La formule $(p \Rightarrow p)$ est donc bien une tautologie.

(b) *Transitive : $p \models q$ et $q \models r$ implique-t-il $p \models r$ pour toutes formules p, q, r ?*

Réponse : Oui, car cela revient à se demander si pour toutes formules p, q, r, si $(p \Rightarrow q)$ et $(q \Rightarrow r)$ sont des tautologies, alors $(p \Rightarrow r)$ est une tautologie. Il faut donc juste montrer qu'il n'existe pas d'interprétation qui rende p vrai et r faux.

Soit une interprétation qui rende p vrai. Puisque $(p \Rightarrow q)$ est une tautologie, une telle interprétation rend nécessairement q vrai. Or, $(q \Rightarrow r)$ est une tautologie, donc une interprétation qui rend q vrai doit nécessairement rendre r vrai également.

En conclusion, toute interprétation qui rend p vrai rend obligatoirement r vrai et il n'existe donc pas d'interprétation qui rende p vrai et r faux.

(c) *Symétrique : $p \models q$ et $q \models p$ implique-t-il $p = q$ pour toutes formules p, q ?*

Réponse : Deux réponses sont possible :

- Non si on considère l'égalité stricte (l'égalité syntaxique). Comme contre-exemple, on a : $A \models \neg\neg A$ et $\neg\neg A \models A$ mais $A \neq \neg\neg A$.
- Oui si on considère l'égalité dans le sens "être équivalent à". En effet, si $p \models q$ et $q \models p$ sont des tautologies pour deux formules p, q, alors toute interprétation qui rend p vrai rend q vrai et toute interprétation qui rend q vrai rend p vrai. Une interprétation rend donc p vrai si et seulement si elle rend q vrai, autrement dit p et q sont équivalentes.

5. *Trouvez des formules sous forme normale conjonctive équivalentes aux formules suivantes :*

(a) $(A \wedge B) \vee C$

Réponse :

- $(A \vee C) \wedge (B \vee C)$ (distributivité)

(b) $\neg(\neg A \vee B) \vee (C \Rightarrow \neg D)$

Réponse :

- $\neg(\neg A \vee B) \vee (\neg C \vee \neg D)$ (élimination des implications)
- $(\neg\neg A \wedge \neg B) \vee (\neg C \vee \neg D)$ (loi de Morgan)
- $(A \wedge \neg B) \vee (\neg C \vee \neg D)$ (double négation)
- $(A \vee \neg C \vee \neg D) \wedge (\neg B \vee \neg C \vee \neg D)$ (distributivité)

(c) $(\neg A \Rightarrow B) \Rightarrow (B \Rightarrow \neg C)$

Réponse :

- $\neg(\neg\neg A \vee B) \vee (\neg B \vee \neg C)$ (élimination des implications)
- $\neg(A \vee B) \vee (\neg B \vee \neg C)$ (double négation)
- $(\neg A \wedge \neg B) \vee (\neg B \vee \neg C)$ (loi de Morgan)
- $(\neg A \vee \neg B \vee \neg C) \wedge (\neg B \vee \neg C)$ (distributivité)

6. *En utilisant la règle de résolution, déterminez si les formules suivantes (qui sont déjà sous forme normale conjonctive) sont satisfiables :*

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

   Réponse : On peut directement appliquer deux résolutions :

   $$\frac{(A \vee B) \quad (\neg A \vee C)}{(B \vee C)} \qquad \frac{(A \vee B) \quad (\neg B)}{A}$$

   À partir des deux clauses obtenues, on peut effectuer deux nouvelles résolutions :

   $$\frac{(\neg A \vee C) \quad A}{C} \qquad \frac{(\neg B) \quad (B \vee C)}{C}$$

   Il n'y a plus d'application de la règle de résolution possible. La formule est donc satisfaisable.

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

   Réponse : On peut effectuer 4 résolutions :

   $$\frac{A \quad (\neg A \vee C)}{C} \qquad \frac{(\neg A \vee C) \quad (\neg B \vee \neg C)}{(\neg A \vee \neg B)} \qquad \frac{(\neg A \vee C) \quad (B \vee \neg C)}{(\neg A \vee B)} \qquad \frac{(\neg B \vee \neg C) \quad (B \vee \neg C)}{(\neg C \vee \neg C)}$$

   On remarque à ce stade qu'on peut déjà obtenir une contradiction (une clause fausse) :

   $$\frac{C \quad (\neg C \vee \neg C)}{\neg C} \qquad \frac{C \quad \neg C}{\bot}$$

   La formule n'est donc pas satisfaisable.

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

   Réponse : Avec ces deux clauses, on peut effectuer une résolution pour chaque variable propositionnelle :

   $$\frac{(A \vee B \vee C) \quad (\neg A \vee \neg B \vee \neg C)}{(B \vee C \vee \neg B \vee \neg C)} \qquad \frac{(A \vee B \vee C) \quad (\neg A \vee \neg B \vee \neg C)}{(A \vee C \vee \neg A \vee \neg C)} \qquad \frac{(A \vee B \vee C) \quad (\neg A \vee \neg B \vee \neg C)}{(A \vee B \vee \neg A \vee \neg B)}$$

   Comme ces trois clauses sont des tautologies, on peut les ignorer. Il n'y a alors plus d'application de la règle de résolution possible. La formule est donc satisfaisable.

9 Complexité algorithmique

1. *Pour les deux questions ci-dessous, donner au moins une raison théorique et une raison pratique pour justifier votre réponse.*

   (a) *Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme le nombre d'opérations ou le temps d'exécution dans le* meilleur des cas *?*

   Réponse : La meilleur cas peut être vu comme une borne inférieure de la complexité d'un algorithme. Mais cette borne ne nous permet pas de déduire la complexité des autres cas, et donc la classe de complexité (ni même la terminaison) de l'algorithme. Un algorithme dont le meilleur cas est $\mathcal{O}(n)$ pourrait donc très bien être intrinsèquement complexe, i.e. $\mathcal{O}(2^n)$, pour la majorité des cas.

   Si on considère le meilleur des cas, il n'y a en fait même plus aucun problème intrinsèquement complexe, puisqu'il est toujours possible de créer un algorithme qui résout un problème en un temps n pour au moins une entrée de taille n (pour tout $n \in \mathbb{N}$).

   En pratique, le meilleur cas n'est pas très intéressant, car on veut plutôt garantir une certaine rapidité d'exécution à l'utilisateur final (on considère donc les pires exécutions possible).

   (b) *Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme la* moyenne *du nombre d'opérations ou du temps d'exécution ?*

   Réponse : La complexité moyenne considère une moyenne pondérée de toutes les entrées possibles (qui sont souvent en un nombre infini et parfois même en partie inconnues !). On doit donc choisir une distribution qui représente la probabilité d'occurrence des différentes entrées. Celle-ci peut être difficile à définir précisément (et le résultat peut dépendre fortement de la distribution choisie). De plus, une telle notion a le désavantage de n'avoir aucune des propriétés élémentaires de calcul : dès que vous combinez deux algorithmes, vous devez systématiquement recalculer la complexité de l'algorithme ainsi obtenu à partir de 0.

   En pratique, on préfère considérer le pire cas, plus facile à identifier. Il se peut cependant que ce cas là soit rare, et fournit donc une estimation trop pessimiste. Dans ce cas, on tente généralement d'approximer la complexité moyenne à partir d'un cas d'exécution "typique".

   Il existe donc des situations où ce type de complexité moyenne est intéressante en pratique, mais la difficulté (théorique et pratique) de calcul de celle-ci fait qu'on préfère bien souvent la notion classique de complexité.

2. *Justifier rigoureusement les affirmations suivantes à partir de la définition de grand $\mathcal{O}$:*

   (a) $3n^3 + 2n^2 + n \in \mathcal{O}(n^3)$

   Réponse : $T(n) \in \mathcal{O}(n^3)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

   $$T(n) \leq Kn^3 \quad \forall n \geq n_0$$

   C'est le cas pour $K = 4$ et $n_0 = 3$ (voir figure 1(a)). En effet, $T(n)$ ne croit pas plus vite que $4n^3$ car $\forall n \geq 3$:

   $$\begin{aligned} & & 3n^3 &\leq 3n^3 \\ \text{et} & & 2n^2 + n &\leq n^3 \\ \implies & & 3n^3 + 2n^2 + n &\leq 4n^3 \end{aligned}$$

(b) $\log_{10}(n^{100}) \in \mathcal{O}(n)$

Réponse : $T(n) \in \mathcal{O}(n)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

$$T(n) \leq Kn \quad \forall n \geq n_0$$

C'est le cas pour $K = 1$ et $n_0 = 300$ (voir figure 1(b)). En effet, $T(n)$ ne croit pas plus vite que n car $\forall n \geq 300$:

$$\begin{aligned} & n \leq 10^{\frac{n}{100}} && (\text{car } 300 \leq 10^3) \\ \implies \quad & \log_{10}(n) \leq \frac{n}{100} \\ \implies \quad & 100 \cdot \log_{10}(n) \leq n \\ \implies \quad & \log_{10}(n^{100}) \leq n \end{aligned}$$

(c) $n^4 \in \mathcal{O}(3^n)$

Réponse : $T(n) \in \mathcal{O}(3^n)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

$$T(n) \leq K \cdot 3^n \quad \forall n \geq n_0$$

C'est le cas pour $K = 1$ et $n_0 = 8$ (voir figure 1(c)). En effet, $T(n)$ ne croit pas plus vite que n car $\forall n \geq 8$:

$$\begin{aligned} & n \leq 3^{\frac{n}{4}} && (\text{car } 8 \leq 3^2) \\ \implies \quad & (n)^4 \leq (3^{\frac{n}{4}})^4 \\ \implies \quad & n^4 \leq 3^n \end{aligned}$$

(d) $2^n \in \mathcal{O}(n!)$

Réponse : On peut montrer par récurrence que $\forall n \geq 4 : n! > 2^n$. En effet, on remarque que $4! = 24 > 16 = 2^4$. Pour tout $n \geq 4$:

$$\frac{(n+1)!}{n!} = n > 2 = \frac{2^{n+1}}{2^n}$$

ce qui signifie que $n!$ croit plus vite que 2^n (voir figure 1(d)).

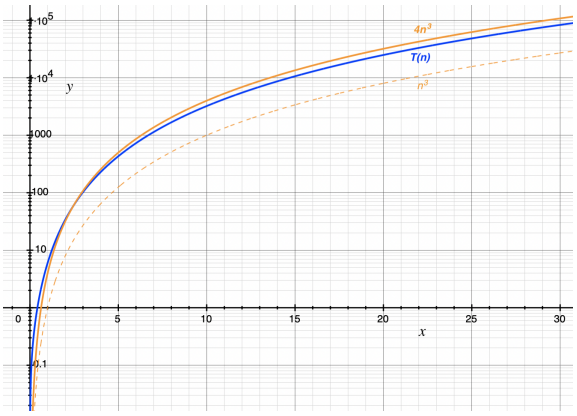
3. *Supposons avoir deux algorithmes qui ont comme input et output des matrices carrées dont la taille n peut varier (la taille de la matrice en output n'est pas nécessairement la même que celle de la matrice en input). Supposons que la complexité du premier est dans $\mathcal{O}(n^2)$ et que celle du second est dans $\mathcal{O}(n^3)$.*

   (a) *Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur cette entrée, puis applique le second algorithme sur cette entrée, puis renvoie le résultat ainsi obtenu ?*

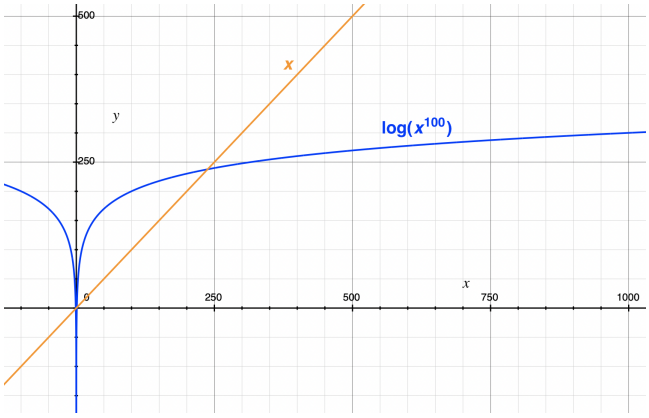
   Réponse : Le temps d'exécution de plusieurs programmes exécutés les uns après les autres équivaut à la somme des temps d'exécution de ces programmes. On a donc une complexité de la forme :

   $$T(n) = T_1(n) + T_2(n) = \mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3 + n^2) = \mathcal{O}(n^3)$$

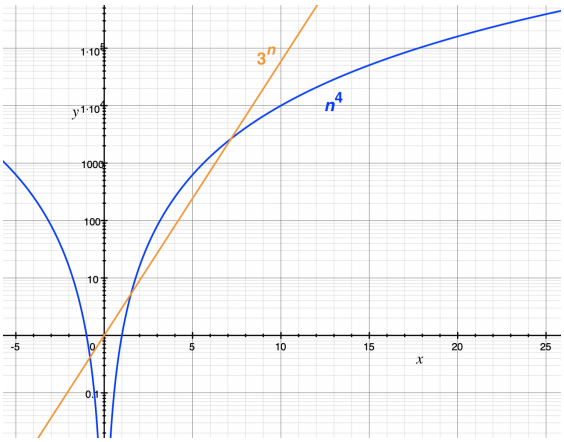
   car on se concentre sur le terme dominant et on ignore le terme $\mathcal{O}(n^2)$.



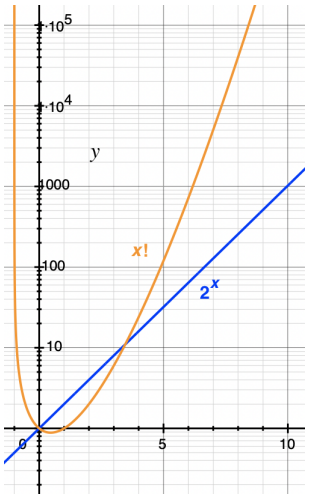
(a) $3n^3 + 2n^2 + n \in \mathcal{O}(n^3)$



(b) $\log_{10}(n^{100}) \in \mathcal{O}(n)$



(c) $n^4 \in \mathcal{O}(3^n)$



(d) $2^n \in \mathcal{O}(n!)$

FIGURE 1 : Illustrations graphiques

(b) *Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur la matrice pour chaque nombre pair plus petit ou égal à n, puis applique le second algorithme sur la matrice pour chaque nombre impair plus petit ou égal à n, puis renvoie la somme des traces des résultats obtenus ?*

Réponse : On peut exprimer la complexité de cet algorithme via la somme des complexités des algorithmes qu'il exécute :

- Si n pair :
$$T(n) = \sum_{i=0}^{\frac{n}{2}} T_1(n) + \sum_{j=0}^{\frac{n}{2}-1} T_2(n)$$
- Si n impair :
$$T(n) = T_2(n) + \sum_{i=0}^{\frac{n-1}{2}} T_1(n) + \sum_{j=0}^{\frac{n-1}{2}-1} T_2(n)$$

Dès lors, puisque que $\mathcal{O}$ est une borne supérieure, on peut donc écrire :

- Si n pair :
$$\begin{aligned} T(n) &= \sum_{i=0}^{\frac{n}{2}} \mathcal{O}(n^2) + \sum_{j=0}^{\frac{n}{2}-1} \mathcal{O}(n^3) \\ &= \frac{n}{2} \cdot \mathcal{O}(n^2) + \Big(\frac{n}{2} - 1\Big) \cdot \mathcal{O}(n^3) \\ &= \mathcal{O}\Big(\frac{1}{2} \cdot n^3\Big) + \mathcal{O}\Big(\frac{1}{2} \cdot n^4 - n^3\Big) = \mathcal{O}(n^4) \end{aligned}$$
- Si n impair :
$$\begin{aligned} T(n) &= \mathcal{O}(n^3) + \sum_{i=0}^{\frac{n-1}{2}} \mathcal{O}(n^2) + \sum_{j=0}^{\frac{n-1}{2}-1} \mathcal{O}(n^3) \\ &= \mathcal{O}(n^3) + \frac{n-1}{2} \cdot \mathcal{O}(n^2) + \Big(\frac{n-1}{2} - 1\Big) \cdot \mathcal{O}(n^3) \\ &= \mathcal{O}(n^3) + \mathcal{O}\Big(\frac{1}{2} \cdot n^3 - \frac{1}{2} \cdot n^2\Big) + \mathcal{O}\Big(\frac{1}{2} \cdot n^4 - \frac{1}{2} \cdot n^3 - n^3\Big) = \mathcal{O}(n^4) \end{aligned}$$

(c) *Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme pour obtenir une seconde matrice, puis applique le second algorithme sur cette seconde matrice, puis renvoie le résultat ainsi obtenu ?*

Réponse : La première partie de l'algorithme a une complexité dans $\mathcal{O}(n^2)$. La deuxième partie de l'algorithme a une complexité dans $\mathcal{O}(m^3)$, où m est la taille de la matrice créée par la première partie de l'algorithme. Or, il faut exprimer la classe de complexité de l'algorithme complet en terme de n, pas de m.

- Si on sait que la taille de m est bornée par le nombre d'opération de l'algorithme qui a produit la matrice de taille m, on a $m \leq n^2$ et donc la complexité de l'algorithme complet est $\mathcal{O}(n^6)$.
- Si on sait (pour une raison ou pour une autre) que m est borné en fonction de n par une fonction $f(n)$, alors la complexité de l'algorithme complet est $\mathcal{O}(f^3(n))$.
- Si on ne connait aucune borne supérieure pour m en fonction de n, on est bloqué.

4. *Est-il possible que le nombre d'opérations réalisées par un algorithme qui prend en entrée des listes de tailles variables soit égal à 100 pour toute liste dont la taille est 42 si...*

(a) La complexité de l'algorithme est dans $\mathcal{O}(2^n)$?

Réponse : Vrai
La notation $\mathcal{O}$ est une *borne supérieure*. En reprenant la définition de $\mathcal{O}$, il existe bien un facteur multiplicatif c (par exemple $c = 1$) tel que $T(42) = 100 \leq c \cdot 2^{42}$.
Par exemple, le programme qui effectue 99 opérations inutiles puis qui affiche son entrée a une complexité constante dans $\mathcal{O}(1)$, et qui est donc bien également dans $\mathcal{O}(2^n)$.

(b) La complexité de l'algorithme est dans $\Omega(2^n)$?

Réponse : Vrai
La borne inférieure de complexité de cet algorithme pourrait être n'importe laquelle. La complexité d'un algorithme n'est définie que pour des valeurs d'entrée n assez grandes (autrement dit, il s'agit d'une notion asymptotique). En ne connaissant le temps d'exécution que d'une seule entrée, on ne peut savoir comment celui-ci va évoluer lorsque n augmentera.
Par exemple, le programme détermine si une formule prédéterminée de la logique des propositions à n variables est satisfaisable par *model checking* (brute force), mais qui pour l'entrée 42 réalise à la place 99 opérations inutiles, puis renvoie 42.

(c) La complexité de l'algorithme est dans $\Theta(2^n)$?

Réponse : Vrai
Même réponse que pour la sous-question précédente.

5. *Le problème de "voyageur de commerce" ("travelling salesman problem" en anglais ; TSP) est un problème algorithmique classique.*

   Un commerçant veut passer par chacune des n villes données. Il existe une route directe entre chaque paire de villes, chaque route ayant une certaine distance en km. Une instance du problème est donc représentée par un graphe avec n nœuds (i.e. les villes), qui est complet (i.e. il y a une arrête entre chaque paire de nœuds) et pondéré (i.e. chaque arrête possède un poids, qui représente ici la distance). Afin de minimiser ses coûts, notre commerçant souhaite trouver le circuit le plus court lui permettant de revenir à sa ville de départ après être passé une et une seule fois par chacune des autres villes (voir exemple à la figure 2).

   (a) *Afin d'obtenir une solution exacte (pour toute entrée), on doit énumérer tous les circuits possibles visitant une et une seule fois chaque nœud du graphe, afin de sélectionner celui totalisant le moins de kilomètres. Quelle est la complexité d'une telle approche ?*

   Réponse : $\mathcal{O}(n!)$

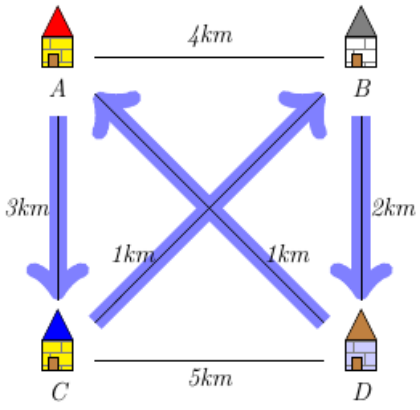


FIGURE 2 : Solution optimale d'une petite instance du TSP

- $n-1$ chemins partant de la 1ère ville ;
- pour chaque cas, $n-2$ chemins sortant ;
- etc.

(b) *Existe-t-il des cas pour lesquels on pourrait trouver la solution exacte en temps polynomial ? Si oui, donnez un exemple. Si non, justifiez.*

Réponse : Un algorithme glouton (*greedy*) consiste à prendre à chaque étape la solution localement optimale, i.e. dans ce cas-ci choisir pour l'étape suivante la ville la plus proche parmi celles restantes.

Une telle approche peut donner la solution optimale dans le cas où toutes les villes sont situées sur un cercle, par exemple.

Attention ! Cette approche ne donne en général pas la solution optimale, car les choix effectués pendant les étapes précédentes ne sont jamais remis en question.

(c) Résoudre le TSP pour des grandes instances peut être intéressant pour l'industrie : e.g. un bras de robot qui resserre une centaine de boulons par voiture dans une ligne de production (minimiser la distance à parcourir peut augmenter la productivité de la ligne). Comment feriez-vous en pratique afin de définir le trajet du robot dans un temps raisonnable ?

Réponse : En pratique, on n'a pas forcément besoin de la solution optimale car il est souvent possible de trouver une bonne approximation en temps polynomial. Par exemple :

- Utiliser un algorithme *greedy*, dans l'espoir qu'il s'approche de la solution optimale.
- Utiliser un algorithme de recherche locale, qui part d'un certain circuit (e.g. la solution *greedy*) et fait une série de permutations afin de continuellement réduire la distance.
- Etc. (Pour plus d'infos : *LINGI2261 - Artificial Intelligence*)

10 Complexité et réductions

1. *Algorithmic versus functional reduction.*

   (a) *Prove that for every set A, $A \leq_a \bar{A}$.*

   Réponse : Soit $\bar{A}$ récursif. Alors il existe un programme $P_{\bar{A}}(a)$ qui calcule :

   $$f_{\bar{A}}(a) = \begin{cases} 1 & \text{si } a \in \bar{A} \\ 0 & \text{sinon} \end{cases}$$

   À partir de ce programme, on construit :

   $$P_A(a) \equiv \texttt{return } 1 - P_{\bar{A}}(a)$$

   f_A est total et calculable, donc A est récursif, ce qui implique $A \leq_a \bar{A}$.

   (b) *Prove that for all sets A, B, if $A \leq_f B$ and B is recursively enumerable, then A is recursively enumerable.*

   Réponse : Soit B récursivement énumérable. Alors il existe un programme $P_{B_{re}}$ qui calcule :

   $$f_{B_{re}}(b) = \begin{cases} 1 & \text{si } b \in B \\ \perp & \text{sinon} \end{cases}$$

   Si $A \leq_f B$, alors il existe une fonction f totale calculable tel que $a \in A \Leftrightarrow f(a) \in B$. Cette fonction peut être calculée par un programme P_f. On peut construire :

   $$P_{A_{re}}(a) \equiv \begin{bmatrix} b = P_f(a) \\ \texttt{return } P_{B_{re}}(b) \end{bmatrix}$$

   A est donc récursivement énumérable.

   (c) *Show an example for which the relation $A \leq_f \bar{A}$ does not hold.*

   Réponse : C'est le cas pour $\bar{K} \not\leq_f K$. En effet, K est récursivement énumérable. Si $\bar{K} \leq_f K$, alors $\bar{K}$ serait aussi récursivement énumérable, ce qui impliquerait que K soit récursif.

   (d) *Show an example for which the following relation does not hold : if $A \leq_a B$ and B is recursively enumerable, then A is recursively enumerable.*

   Réponse : Par définition, $\bar{K} \leq_a K$. K est récursivement énumérable, mais pas $\bar{K}$.

2. *Prove the following relations.*

   (a) *Does $A \leq_a B$ imply $A \leq_f B$?*

   Réponse : Faux
   Par définition, $\bar{K} \leq_a K$, et on sait déjà que $\bar{K} \leq_f K$ est faux (voir exercice 1.3).

   (b) *Does $A \leq_f B$ imply $A \leq_a B$?*

   Réponse : Vrai
   Supposons $A \leq_f B$. Alors il existe une fonction f totale calculable tel que $a \in A \Leftrightarrow f(a) \in B$. Si on suppose B récursif, alors on peut déterminer si $f(a) \in B$ ou non, ce qui revient à décider $a \in A$. A est donc aussi récursif, ce qui implique $\leq_a B$.

3. *Let* $re = \{A \mid A \text{ is recursively enumerable}\}$.

   (a) *State the conditions to prove that HALT is re-complete w.r.t.* $\leq_a$.

   Réponse : Il faut prouver :

   i. $HALT \in re$

   ii. $\forall B \in re, B \leq_a HALT$

   (b) *Prove the conditions.*

   Réponse :

   i. On peut construire :

   $$P_{HALT_{re}}(n, x) \equiv \left[\begin{array}{l} P_n(x) \\ \texttt{return } 1 \end{array} \right.$$

   ce qui prouve $HALT \in re$.

   ii. Si $B \in re$, il existe un programme $P_{B_{re}}$ qui calcule :

   $$f_{B_{re}}(b) = \begin{cases} 1 & \text{si } b \in B \\ \perp & \text{sinon} \end{cases}$$

   Si on suppose $HALT$ récursif, alors on peut calculer la fonction de décision de B :

   $$P_B(b) \equiv \left[\begin{array}{l} \texttt{if } halt(B_{re}, b) \texttt{ return } 1 \\ \texttt{else return } 0 \end{array} \right.$$

   ce qui impliquerait B récursif. On a donc bien $B \leq_a HALT$.

4. *Does the class* $DTIME(n^2)$ *depend of your calculus model (e.g. Java language) ? And what about the class P ? Justify your answers.*

   Réponse :

   - La classe $DTIME(n^2)$ est spécifique à un modèle de calculabilité. En effet, elle est définie comme l'ensemble des problèmes pouvant être résolus par un programme Java ayant une complexité $\mathcal{O}(n^2)$. Les modèles de complexités sont liés entre eux par un facteur polynomial. Pour les mêmes problèmes, on aura donc une complexité $\mathcal{O}(n^i)$ (avec potentiellement $i \neq 2$) avec une machine de Turing par exemple.
   - Tous les modèles de complexité étant liés entre eux par un facteur polynomial, tout problème résolu en $\mathcal{O}(n^i)$ avec Java sera forcément résolu en $\mathcal{O}(n^j)$ par un autre modèle de complexité. La définition de P demeure donc inchangée :

   $$P = \bigcup_{i \geq 0} DTIME(n^i)$$

5. *Let A be a recursive set and f a function from* $\mathbb{N}$ *to* $\mathbb{N}$. *Are the following implication true ? Explain.*

   (a) $A \in NTIME(f) \Longrightarrow \exists c > 1 : A \in DTIME(c^f)$

   Réponse : Vrai

   On peut représenter l'ensemble des exécutions possibles d'un programme non-déterministe par un arbre d'exécution, où chaque nœud représente un choix d'exécution. Si une exécution se fait en $\mathcal{O}(f)$, parcourir cet arbre (i.e. simuler l'ensemble des exécutions possibles) se fait avec un facteur exponentiel. On est donc en $\mathcal{O}(c^f)$.

(b) $A \in DTIME(f) \Longrightarrow A \in DSPACE(f)$

Réponse : Vrai

En effet, un programme décidant A peut utiliser au plus 1 emplacement en mémoire par instruction. La complexité spatiale est donc bornée par la complexité temporelle.

11 Classes de complexité et NP-complétude

1. *Is there a problem that can be solved in polynomial time in Java, but always takes exponential time with a Turing machine ?*

   Réponse : Non, la différence est au plus d'ordre polynomiale. Il s'agit d'une thèse, car cela n'a jamais été démontré, mais on n'a jamais trouvé de contre-exemple jusqu'ici.

2. *For the following problems A and B find if $A \leq_p B$ and justify :*

   (a) *$A(S, x)$ is the problem of knowing if the maximum element of a set of integers S is greater than a value x.*
   $B(S, x)$ is the problem of knowing if the minimum element of a set of integers S is lower than a value x.

   Réponse : Soit la fonction $f(S, x)$ qui multiplie tous les éléments de S et x par -1. Il existe un programme qui calcule cette fonction dont la complexité est dans $\mathcal{O}(n)$. On peut décider A en utilisant un programme qui décide B :

   $$P_A(S, x) \equiv \texttt{return } P_B(f(S, x))$$

   (b) *$A(S_1, S_2)$ is the problem of knowing if $\sum_{x \in S_1} x = \sum_{x \in S_2} x$.*
   $B(S)$ is the problem of knowing if $\sum_{x \in S} x = 0$.

   Réponse : Soit la fonction $f(S_1, S_2)$ qui renvoie un nouvel ensemble S tel que :

   $$S = \bigcup_{x \in S_1} \{x\} \bigcup_{x \in S_2} \{-x\}$$

   Il existe un programme qui calcule cette fonction dont la complexité est dans $\mathcal{O}(n)$. On peut calculer A en utilisant un programme qui calcule B :

   $$P_A(S_1, S_2) \equiv \texttt{return } P_B(f(S_1, S_2))$$

   (c) *$A(S, y)$ is the problem of knowing if there is a subset $S^* \subseteq S$ such that $\sum_{x \in S^*} x = y$.*
   $B(S, y)$ is the problem of knowing if $\sum_{x \in S} x = y$.

   Réponse : Soit la fonction $f(S, y)$ qui renverrait un sous-ensemble de S tel que la somme de ses éléments soit égal à y s'il en existe un. On peut calculer A en utilisant un programme qui calcule B :

   $$P_A(S, y) \equiv \texttt{return } P_B(f(S, y), y)$$

   Cela dit, pour trouver un tel sous-ensemble, la fonction f devrait naïvement essayer toutes les combinaisons possibles de sous-ensembles de S, ce qui la rend de complexité du programme qui calcule cette fontion de manière triviale non-polynomiale. Jusqu'à preuve du contraire (si $P = NP$), il n'existe pas de programme avec une complexité polynomiale qui calcule cette fonction. En effet, le problème A est en fait un problème *NP*-complet.

3. *Assume you have a polynomial time algorithm to decide the satisfiability of a propositional formula. Describe how to use this algorithm to find satisfying values for the variables in polynomial time. What kind of reduction is it ? ($\leq_a$, $\leq_f$ or $\leq_p$?)*

   Réponse : On utilise une réduction polynomiale.

```
assign ← {}
if ¬P_SAT(F) then
    return null
for x ∈ Vars(F) do
    if P_SAT(F|_{assign∪{(x,true)}}) then
        assign ← assign ∪ {(x, true)}
    else
        assign ← assign ∪ {(x, false)}
return assign
```

4. *Which of the following can we infer from the fact that the Hamiltonian cycle (HC)*[1] *problem is NP-complete, if we assume that $P \neq NP$? What would your answer be if $P = NP$? Justify.*

   (a) *There does not exist a deterministic algorithm that solves the HC problem in polynomial time.*

   Réponse : Vrai, sinon $P = NP$. Faux

   (b) *There exists an algorithm that solves the HC problem in polynomial time, but no one has been able to find it yet.*

   Réponse : Faux, sinon $P = NP$. Vrai

   (c) *The HC problem is not in P.*

   Réponse : Vrai, sinon $P = NP$. Faux

   (d) *The problem of sorting an array can be reduced in polynomial time to the HC problem.*

   Réponse : Vrai, car $P \subset NP$ et HC est NP-complet. Même réponse

   (e) *All non-deterministic algorithms for the HC problem run in polynomial time.*

   Réponse : Faux, on peut écrire un algorithme moins efficace ! Même réponse

5. *The Traveling Salesman (TS) problem is the problem of finding a cycle in a graph with strictly positive weights that touches every vertex such that the sum of the weights on the cycle do not exceed a given bound. The Hamiltonian Cycle (HC) problem is the problem of finding, in a graph, a cycle that touches every vertex exactly once. Prove that TS is NP-complete by using the fact that HC is NP-complete.*

   Réponse : Soit $G = (V, E)$ un graphe. Le problème du voyageur de commerce peut être formulé comme un problème de décision : $TS(G, b) \Leftrightarrow \exists cycle \in G$ tel que $cost(cycle) \leq b$.

   Pour prouver que TS est NP-complet, il faut prouver que :

   i. TS $\in NP$

      Les solutions possibles du problème TS sont les cycles de G passant par chaque $v \in V$ dont la longueur est au plus $2|E|$. On a donc un domaine fini qui peut être généré de manière non-déterministe en un temps polynomial. On peut vérifier une solution (i.e. calculer le coût total du cycle et vérifier s'il est $\leq b$) en un temps polynomial. $\Longrightarrow$ TS $\in NP$

[1] The Hamiltonian cycle problem is the problem consisting of deciding if a given graph accepts at least one cycle that visits all vertices exactly once.

ii. HC $\leq_p$ TS

Soit $f(G)$ qui assigne un coût de 1 à toutes les arrêtes dans E. Alors on peut calculer HC en utilisant un programme qui calcule TS :

$$P_{HC} \equiv \texttt{return } P_{TS}(f(G), |V|)$$

Remarque. Formellement, le problème TS est défini pour un graphe complet. Pour être complet, la fonction $f(G)$ doit donc aussi ajouter des arrêtes avec un coût ∞ entre chaque paire de nœuds non connectés dans G.

12 Propriétés des modèles de calculabilité

1. *Prove the following implication : $CA \wedge SD \wedge U \Longrightarrow SA \wedge CD \wedge S$*

   Réponse : Considérons un formalisme de calculabilité D qui vérifie CA, SD et U. Puisque la fonction interpréteur de D est D-calculable et que toute fonction D-calculable est calculable (SD), la fonction interpréteur de D est calculable (SA).

   Puisqu'il existe un compilateur calculable capable de transformer toute machine de Turing M en un programme P de D qui calcule la même fonction (CA), toute fonction calculable (par une machine de Turing) est calculable à l'aide d'un programme de D (CD).

   Reste à démontrer S : soit une fonction $f(x, y)$ qui est D-calculable et son programme associé $P(x, y)$. Par SA, elle est calculable : notons $M(x, y)$ la machine de Turing qui la calcule. La propriété S est vraie pour les machines de Turing : il existe S telle que pour tout $x, y \in \mathbb{N}$:

   $$M(x, y) = [S(x)](y)$$

   Par CA, il existe un programme S_D de D qui calcule la même fonction, c'est-à-dire tel que pour tout $x, y \in \mathbb{N}$:

   $$P(x, y) = [S_D(x)](y)$$

   Donc D vérifie S.

2. *What is the difference between U and SA ? Does one imply the other ? If not, what property do you have to add to SA to imply U ?*

   Réponse : Aucune des deux propriétés n'implique l'autre. Donnons deux contre-exemples :

   - Les automates finis déterministes vérifient SA mais pas U.
   - Le langage Python/Java avec un oracle pour l'ensemble $\{k \in \mathbb{N} \mid \varphi_k(42) = 42\}$ vérifie U mais pas SA.

   Par contre, SA avec CD implique U (voir question 5 du quiz).

3. *Among the properties SD, CD, SA, CA, U, and S, which ones hold for the following formalisms ?*

   (a) *Java / Python*

   Réponse : SD, CD, SA, CA, U, et S

   (b) *The Oracle Turing Machines, where the oracle set is $K = \{n \mid P_n(n)$ terminates$\}$ (where P_n is the n-th Turing Machine)*

   Réponse : CD, CA, U et S

   (c) *The language BLOOP with an oracle-program, where the oracle set is $K = \{n \mid P_n(n)$ terminates$\}$ (where P_n is the n-th BLOOP program)*

   Réponse : SD, SA et S

4. *True or false ?*

   (a) *There exist languages that have the SD property, but not the SA property.*

   Réponse : Vrai

Exemple : l'ensemble des programmes Java/Python avec un bonus des programmes-oracles tels que :

$$P_i(x) \equiv \texttt{return } halt(i, 0)$$

(pour tout $i \in \mathbb{N}$, avec halt la fonction halt des programmes Java/Python). Comme pour tout $i \in \mathbb{N}$, φ_i est constante, elle est calculable. Donc ce formalisme vérifie SD.

Mais il ne vérifie pas SA car il est impossible de déterminer de façon calculable quelle sera la valeur de cette constante. Si ce formalisme de calculabilité vérifiait SA, alors cela impliquerait que l'ensemble $\{i \in \mathbb{N} \mid P_i(0) \neq \perp\}$ est récursif, ce qui est faux.

(b) *A formalism satisfying the SA, SD and U properties is a good computability formalism.*

Réponse : Faux

Contre-exemple : le formalisme qui contient un unique programme : `return 1`.