

---

## Correction des exercices de TD

---

## Table des matières

<b>1</b>	<b>Calculabilité</b>	<b>1</b>
1.1	Divers . . . . .	1
1.2	Variations sur le codage . . . . .	2
1.3	Diagonalisation . . . . .	8
1.4	Dénombrabilité . . . . .	10
1.5	Fonctions (non)-calculables . . . . .	12
1.6	Problèmes indécidables . . . . .	14
1.7	Théorème de Rice . . . . .	16
1.8	Décidabilité et récursivement énumérable . . . . .	17
1.9	Sur le point fixe . . . . .	22
<b>2</b>	<b>Complexité</b>	<b>23</b>
2.1	Rappel . . . . .	23
2.2	Autour des classes $\mathcal{P}$ et $\mathcal{NP}$ . . . . .	26
2.3	Réduction polynomiale . . . . .	30

## 1 Calculabilité

### 1.1 Divers

#### Exercice 1 - Paradoxe

Montrer que les problèmes suivants engendrent un paradoxe.

1. Le conseil municipal d'un village vote un arrêté municipal qui enjoint à son barbier (masculin) de raser tous les habitants masculins du village qui ne se rasent pas eux-même et seulement ceux-ci.
2. Un crocodile s'empare d'un bébé et dit à la mère : « si tu devines ce que je vais faire, je te rends le bébé, sinon je le dévore. ». En supposant que le crocodile tienne parole, que doit dire la mère pour que le crocodile rende l'enfant à sa mère ? Une réponse usuelle de la mère est : « Tu vas le dévorer ! »

1. Le barbier, s'il se rase, se rase lui-même, mais il est aussi rasé par le barbier. S'il ne se rase pas, il doit être rasé par le barbier, c'est à dire lui-même, donc il se rase. L'énoncé est faux (impossible).
2. Si le crocodile rend le bébé à la mère avec cette réponse, c'est qu'il comptait le dévorer. Seulement, s'il a l'intention de rendre le bébé, c'est qu'il n'a pas l'intention de le dévorer, donc il le dévorera.

## Exercice 2 - Une preuve incorrecte

Nous considérons la fonction suivante donnée par l'algorithme 1 :

**Algorithm 1:** La fonction de Collatz

```
1 begin
2   while  $n \neq 1$  do
3     if  $n \bmod 2 = 0$  then
4        $n := n/2$ 
5     else
6        $n := 3 \times n + 1$ 
```

Actuellement nous ne savons pas si cette fonction termine  $\forall n$ . Êtes-vous d'accord avec la preuve suivante ? « Si le problème de l'arrêt était décidable, il suffirait de l'appliquer à ce programme pour savoir si son exécution s'arrête. Or, on ne sait pas si son exécution s'arrête. D'où la contradiction. »

Le raisonnement est faux, la preuve est donc incorrecte. En effet, « on ne sait pas si son exécution s'arrête » signifie que soit elle s'arrête, soit elle ne s'arrête pas, on répète la question. Il n'y a pas de lien logique entre la première et la seconde phrase.

## 1.2 Variations sur le codage

### Exercice 3 - Codage de couples d'entiers

Soit  $Rang : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  tel que  $Rang(x, y) = \frac{(x+y)(x+y+1)}{2} + y$ .

1. Donner une version récursive de la fonction  $Rang$ .
2. Donner la fonction inverse.
3. Calculer  $Rang(4, 5)$ . Donner le couple pour lequel la valeur du codage est 8.

$$1. RangRec = \begin{cases} 0 & \text{si } x = 0 \text{ et } y = 0 \\ RangRec(0, x-1) + 1 & \text{si } y = 0 \\ RangRec(x+1, y-1) + 1 & \text{sinon} \end{cases}.$$

2. On pose la fonction inverse  $RangInv(n) : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ . On cherche d'abord  $x + y$ , on prend donc  $x + y = \max\{m \mid \frac{m(m+1)}{2} \leq n\}$ . On pose  $t = x + y$ . Ainsi, comme  $n = \frac{t(t+1)}{2} + y$ , on a  $y = n - \frac{t(t+1)}{2}$ . De plus, comme  $t = x + y$ , pour retrouver  $x$ , il suffit de prendre  $x = t - y$ .
3.  $Rang(4, 5) = \frac{(4+5)(4+5+1)}{2} + 5 = \frac{9 \times 10}{2} + 5 = 50$ . Pour  $n = 8$ , on cherche d'abord  $t$ . On a  $t = \max\{1, 2, 3\} = 3$ , et  $\frac{t(t+1)}{2} = 6$ . On a donc  $y = 8 - 6 = 2$  et  $x = 3 - 2 = 1$ . Le couple codé par  $n = 8$  est  $(1, 2)$ .

### Exercice 4 - Codage de triplets

Soit  $c$  la fonction de codage pour les couples d'entiers vue dans l'exercice précédent.

1. Soit  $h$  la fonction de codage pour les triplets définie par  $h(x, y, z) = c(c(x, y), z)$ . Quel est le doublet codé par 67 ? Quel est le triplet codé par 67 ?
2. Le couple  $(z, t)$  succède au couple  $(x, y)$  si  $c(z, t) = c(x, y) + 1$ . Écrire la fonction successeur qui prend en paramètre un couple et retourne le couple successeur.

1. Pour  $n = 67$ , on cherche d'abord  $t$ . On a  $t = \max\{m | m \leq 11\} = 11$ . En effet,  $\frac{t(t+1)}{2} = \frac{11 \times 12}{2} = \frac{132}{2} = 66$ . On a donc  $y = 67 - 66 = 1$  et  $x = 11 - 1 = 10$ . Comme 67 code  $(10, 1)$ , pour avoir le triplet, on veut le couple codé par  $n = 10$ . De nouveau, on cherche  $t = \max\{1, 2, 3, 4\} = 4$ . En effet,  $\frac{4 \times 5}{2} = 10$ . Donc  $y = 10 - 10 = 0$  et  $x = 4 - 0 = 4$ . Le couple codé par  $n = 10$  est  $(4, 0)$  et ainsi le triplet codé par  $n = 67$  est  $(4, 0, 1)$ .
2. La fonction  $c$  fait augmenter les couples comme une diagonale. Prenons les premiers couples :

$c(x, y)$	0	1	2	3	4	5	6	7	8	9
$(x, y)$	(0, 0)	(1, 0)	(0, 1)	(2, 0)	(1, 1)	(0, 2)	(3, 0)	(2, 1)	(1, 2)	(0, 3)

On remarque que le successeur de  $(x, y)$  est  $(x - 1, y + 1)$  (qui est d'ailleurs l'inverse de ce qu'on avait fait à l'exercice précédent pour *RangRec*), et que si  $x = 0$ , le successeur est  $(y + 1, 0)$ . Cela nous donne

$$\text{la fonction suivante : } \text{successeur}(x, y) = \begin{cases} (y + 1, 0) & \text{si } x = 0 \\ (x - 1, y + 1) & \text{sinon} \end{cases}$$

### Exercice 5 - Étude d'une équation fonctionnelle dans $\mathbb{N}$

Soit  $f$  une application de  $\mathbb{N}$  dans  $\mathbb{N}$  telle que :  $\forall (m, n) \in \mathbb{N}, f(m^2 + n^2) = f(m)^2 + f(n)^2$ . Nous voulons montrer que  $f$  est :

- l'application nulle, donnée par :  $\forall n \in \mathbb{N}, f(n) = 0$ ,
- l'application identité, donnée par :  $\forall n \in \mathbb{N}, f(n) = n$ .

Nous supposons que  $a$  est l'entier naturel  $f(1)$ .

1. Montrer que  $f(0) = 0$ . En déduire que  $\forall n \in \mathbb{N}, \text{on a } f(n^2) = f(n)^2$ .
2. Montrer alors que  $a^2 = a$ , donc que  $a$  est égal à 0 ou à 1.
3. Vérifier successivement les égalités  $f(2) = 2a$ ,  $f(4) = 4a$  et  $f(5) = 5a$ .
4. Utiliser les valeurs  $f(4)$  et  $f(5)$  pour montrer que  $f(3) = 3a$ .
5. Utiliser les valeurs de  $f(1)$  et de  $f(5)$  pour montrer que  $f(7) = 7a$ .
6. Montrer que  $f(8) = 8a$ ,  $f(9) = 9a$ ,  $f(10) = 10a$  et  $f(6) = 6a$ .
7. Observer que

$$\forall m, \text{ on a } \begin{cases} (2k)^2 + (k - 5)^2 = (2k - 4)^2 + (k + 3)^2 \\ (2k + 1)^2 + (k - 2)^2 = (2k - 1)^2 + (k + 2)^2 \end{cases}$$

Montrer que  $\forall n$ , on a  $f(n) = an$ .

8. Conclure.

1. On a  $m^2 + n^2 = 0 \Rightarrow m = 0$  et  $n = 0$  :

$$\begin{aligned} f(m^2 + n^2) &= f(m)^2 + f(n)^2 \\ \Rightarrow f(0) &= f(0)^2 + f(0)^2 \\ \Rightarrow f(0) &= 2f(0)^2 \\ \Rightarrow f(0) - 2f(0)^2 &= 0 \\ \Rightarrow f(0)(1 - 2f(0)) &= 0 \end{aligned}$$

Il y a deux cas. Soit  $f(0) = 0$ , soit  $1 - 2f(0) = 0 \Rightarrow f(0) = \frac{1}{2}$ . Or,  $f$  est une fonction de  $\mathbb{N}$  dans  $\mathbb{N}$ , donc le seul résultat possible ici est  $f(0) = 0$ . On en déduit que  $f(n^2) = f(0^2 + n^2) = f(0)^2 + f(n)^2 = f(n)^2$ .

2. On sait que  $f(n^2) = f(n)^2$ . De plus,  $1^2 = 1$ , donc  $f(1) = f(1)^2 \Rightarrow a = a^2$ . Dans les entiers, seulement 0 et 1 vérifient cette égalité.
3.  $f(2) = f(1^2 + 1^2) = f(1)^2 + f(1)^2 = a^2 + a^2 = 2a^2 = 2a$ .  
 $f(4) = f(0^2 + 2^2) = f(0)^2 + f(2)^2 = (2a)^2 = 4a^2 = 4a$ .  
 $f(5) = f(1^2 + 2^2) = f(1)^2 + f(2)^2 = a^2 + 4a^2 = 5a^2 = 5a$ .
4.  $f(5^2) = f(25) = f(3^2 + 4^2) = f(3)^2 + f(4)^2 = f(3)^2 + 16a^2$ . De plus,  $f(5^2) = f(5)^2 = (5a)^2 = 25a^2$ .  
On a donc  $f(3)^2 = 25a^2 - 16a^2 = 9a^2$ . Ainsi,  $f(3) = \sqrt{9a^2} = 3a$ .
5.  $f(5^2 + 5^2) = f(1^2 + 7^2) = f(50)$ . On a  $f(50) = f(5)^2 + f(5)^2 = 50a^2$  et  $f(50) = f(1)^2 + f(7)^2 = a^2 + f(7)^2$ .  
On a donc  $f(7)^2 = 50a^2 - a^2 = 49a^2$ . Ainsi,  $f(7) = \sqrt{49a^2} = 7a$ .
6.  $f(8) = f(2^2 + 2^2) = f(2)^2 + f(2)^2 = 4a^2 + 4a^2 = 8a^2 = 8a$ .  
 $f(9) = f(3^2 + 0^2) = f(3)^2 + f(0)^2 = 9a^2 = 9a$ .  
 $f(10) = f(3^2 + 1^2) = f(3)^2 + f(1)^2 = 9a^2 + a^2 = 10a^2 = 10a$ .  
 $f(10^2) = f(8^2 + 6^2) = f(100)$ . On a  $f(100) = f(10)^2 = f(10)^2 = 100a^2$  et  $f(100) = f(8^2 + 6^2) = f(8)^2 + f(6)^2 = 64a^2 + f(6)^2$ . On a donc  $f(6)^2 = 100a^2 - 64a^2 = 36a^2$ . Ainsi,  $f(6) = \sqrt{36a^2} = 6a$ .
7. On veut prouver l'hypothèse  $H(n) : \forall n, f(n) = an$ . On prouve ça par induction :

**Base** On a prouvé précédemment tous les cas pour  $n \leq 10$ .

**Induction** On suppose  $\forall i < n, H(i)$ . Montrons  $H(n)$ . Il y a deux cas :  $n$  pair, c'est à dire qu'il existe  $k$  tel que  $n = 2k$  ou bien  $n$  impair, c'est à dire qu'il existe  $k$  tel que  $n = 2k + 1$ .

—  $n$  est pair. On sait que  $f((2k)^2 + (k - 5)^2) = f((2k - 4)^2 + (k + 3)^2)$  :

$$\begin{aligned}
f((2k)^2 + (k - 5)^2) &= f((2k - 4)^2 + (k + 3)^2) \\
&\Rightarrow f(2k)^2 + f(k - 5)^2 = f(2k - 4)^2 + f(k + 3)^2 \\
&\Rightarrow f(2k)^2 = f(2k - 4)^2 + f(k + 3)^2 - f(k - 5)^2 \\
&\Rightarrow f(2k)^2 = a^2(2k - 4)^2 + a^2(k + 3)^2 - a^2(k - 5)^2 \\
&\Rightarrow f(2k)^2 = a^2(4k^2 - 16k + 16) + a^2(k^2 + 6k + 9) - a^2(k^2 - 10k + 25) \\
&\Rightarrow f(2k)^2 = a^2(4k^2) \\
&\Rightarrow f(2k) = \sqrt{4a^2k^2} \\
&\Rightarrow f(2k) = a2k
\end{aligned}$$

—  $n$  est impair. On sait que  $f((2k + 1)^2 + (k - 2)^2) = f((2k - 1)^2 + (k + 2)^2)$  :

$$\begin{aligned}
f((2k + 1)^2 + (k - 2)^2) &= f((2k - 1)^2 + (k + 2)^2) \\
&\Rightarrow f(2k + 1)^2 + f(k - 2)^2 = f(2k - 1)^2 + f(k + 2)^2 \\
&\Rightarrow f(2k + 1)^2 = f(2k - 1)^2 + f(k + 2)^2 - f(k - 2)^2 \\
&\Rightarrow f(2k + 1)^2 = a^2(2k - 1)^2 + a^2(k + 2)^2 - a^2(k - 2)^2 \\
&\Rightarrow f(2k + 1)^2 = a^2(4k^2 - 4k + 1) + a^2(k^2 + 4k + 4) - a^2(k^2 - 4k + 4) \\
&\Rightarrow f(2k + 1)^2 = a^2(4k^2 + 4k + 1) \\
&\Rightarrow f(2k + 1) = \sqrt{a^2(4k^2 + 4k + 1)} \\
&\Rightarrow f(2k + 1) = a(2k + 1)
\end{aligned}$$

**Conclusion** On a prouvé que  $\forall i \leq 10, H(i)$  et  $\forall i < n, H(i) \Rightarrow H(n)$ , on a donc bien  $\forall n, f(n) = an$ .

8. On a prouvé que  $\forall n, f(n) = an$ . De plus, on sait que  $a = 0$  ou  $a = 1$ . Ainsi, il y a deux cas :

- $\forall n, f(n) = 0 \times n = 0$  (application nulle),
- $\forall n, f(n) = a \times n = n$  (application identité).

On a prouvé que les deux seules applications de  $\mathbb{N} \rightarrow \mathbb{N}$  telles que  $\forall (m, n) \in \mathbb{N}, f(m^2 + n^2) = f(m)^2 + f(n)^2$  sont l'application nulle et l'application identité.

## Exercice 6 - Codage rationnels

Proposer un codage pour les nombres rationnels.

On peut proposer un codage naïf : toute fraction rationnelle  $\frac{a}{b}$  se réduit en fraction  $\frac{p}{q}$  avec  $p, q$  premiers. Pour coder les rationnels, on pourrait prendre  $c(p, q)$ . Cependant, ce codage est très inefficace, car il y a énormément de couples non premiers entre eux.

Une méthode moins naïve de faire serait de poser la fonction  $\sigma$  qui prend  $(p + q)$  et ordonne par  $p$  (numérateur) croissant lors de l'égalité :

$\sigma(p/q)$	1	2	3	4	5	6
$\frac{p}{q}$	$\frac{0}{1}$	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{2}{1}$	$\frac{1}{3}$	$\frac{3}{1}$

On remarque qu'on saute  $\frac{2}{2}$ , car on peut réduire la fraction à  $\frac{1}{1}$ .

## Exercice 7 - Codage des listes d'entiers

Pour coder les listes d'entiers, peut-on :

1. Faire la somme des entiers de la liste, et à somme égale prendre l'ordre lexicographique ?
2. Faire comme pour les mots : prendre les listes les plus courtes d'abord et à égalité de longueur l'ordre lexicographique ?

1. Non, car dans ce cas, on n'aurait que les listes qui contiennent des 0 :  $(0), (0, 0), (0, 0, 0), \dots$
2. Non, car dans ce cas, on n'aurait que les listes de longueur 1 :  $(0), (1), (2), \dots$

## Exercice 8 - Codage de listes d'entiers

On ordonne les listes de la façon suivante :

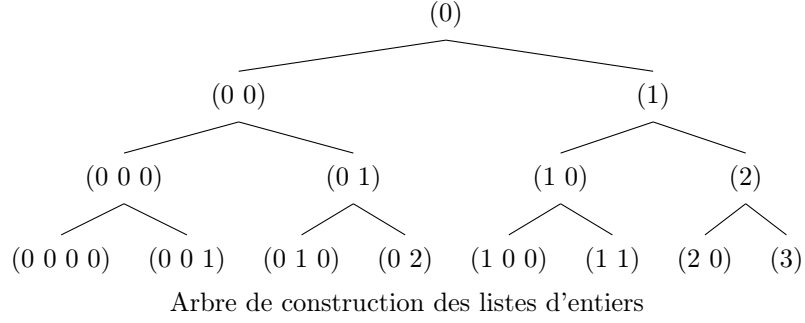
$$\sigma(l) = \text{somme des entiers de la liste} + \text{longueur de la liste}$$

Puis à valeur de  $\sigma$  égale on ordonne dans l'ordre lexicographique.

On note  $U_k$  l'ensemble des listes  $l$  telles que  $\sigma(l) = k$  et  $u_k = |U_k|$ .

1. Donner les ensemble  $U_i, i = 0, \dots, 4$ .
2. Montrer que  $u_k = 2^{k-1}, \forall k \geq 1$ .
3. Quelle est la première liste de  $U_k, \forall k \in \mathbb{N}^*$  et la dernière ?
4. Donner la fonction de codage en version itérative et récursive (resp. décodage).

1.  $U_0 = \{()\}$  la liste vide.  
 $U_1 = \{(0)\}$ .  
 $U_2 = \{(0, 0), (1)\}$ .  
 $U_3 = \{(0, 0, 0), (1, 0), (0, 1), (2)\}$ .  
 $U_4 = \{(0, 0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (2, 0), (1, 1), (0, 2), (3)\}$ .
2. Pour tout  $k \geq 1$ , on peut exhiber la méthode de construction suivante : pour chaque élément  $l$  de  $U_{k-1}$ , le fils gauche est  $l$  à laquelle est ajoutée 0 à la fin, et le fils droit est  $l$  en ajoutant 1 à son dernier élément :



On peut ainsi faire une preuve par induction :

**Base** On a prouvé le cas pour  $k \leq 4$ .

**Induction** On suppose  $\forall i \leq n, u_i = 2^{i-1}$ . Prouvons le pour  $u_{n+1}$ . Comme  $u_n = 2^{n-1}$  et que la construction est inductive sous forme d'arbre, il y a 2 fils pour chaque élément de  $U_n$  donc la taille est  $2 \times u_n = 2 \times 2^{n-1} = 2^n$ .

**Conclusion** On a montré la base, et pour tout  $k \geq 1$ , on a bien  $u_k = 2^{k-1} \Rightarrow u_{k+1} = 2^k$ .

3. La première liste de  $U_k$  est la liste  $(0, \dots, 0)$  composée de  $k$  0, et la dernière est le singleton  $(k - 1)$ .
4. On peut donner la version récursive suivante, qui se déduit pratiquement immédiatement de l'arbre de construction des listes :

**Algorithm 2:** Codage d'une liste d'entier

```

1 begin
2   if  $n = 1$  et  $e_0 = 0$  then
3     return 1
4   if  $e_{n-1} = 0$  then
5     return  $2 \times \text{codage}(e_0, \dots, e_{n-2})$ 
6   return  $2 \times \text{codage}(e_0, \dots, e_{n-1} - 1) + 1$ 

```

Le décodage est ainsi intuitif : on prend un compteur, si  $n \bmod 2 = 0$ , on ajoute 0 à la liste, sinon, tant que  $n \bmod 2 = 1$ , on ajoute 1 au compteur :

**Algorithm 3:** Décodage en liste d'entier

```

1 begin
2   if  $n = 1$  then
3     return (0)
4   if  $n \bmod 2 = 0$  then
5     return  $\text{cons}(\text{decodage}(n/2), 0)$ 
6    $L := \text{decodage}(\lfloor n/2 \rfloor)$ ;
7    $L_{[-1]} := L_{[-1]} + 1$ ;
8   return  $L$ 

```

$L_{[-1]}$  correspond au dernier élément de la liste.

## Exercice 9 - Codage d'entiers

Soit la fonction  $f$  suivante de  $\mathbb{N}^* \rightarrow \mathbb{N}$  :

$$\begin{aligned} f(n) &= k \text{ si } n = 2^k \\ f(n) &= f(n/2) \text{ si } n \text{ est pair et n'est pas une puissance de 2} \\ f(n) &= f(3n + 1) \text{ sinon} \end{aligned}$$

Nous appelons  $A_i = \{x \mid f(x) = i\}$ .

1. Donner quelques éléments de  $A_i$ ,  $\forall i \in \{1, 2, 3, 4, 5, 6\}$ .
2. Donner un algorithme qui prend  $i$  en paramètre et qui affiche tous les éléments de  $A_i$ .
3. Donner un algorithme qui affiche  $A_1 \cup A_2$ .
4. Donner un algorithme qui affiche  $A_4 \cup A_6$ .

1.  $A_1 = \{2\}$   
 $A_2 = \{4\}$   
 $A_3 = \{8\}$   
 $A_4 = \{16, 3, 5, 10, 24, 48, 20, 40, \dots\}$   
 $A_5 = \{32, \dots\}$   
 $A_6 = \{64, 21, 42, 84, \dots\}$

2. Intuitivement, on voudrait faire l'algorithme suivant :

**Algorithm 4:** Afficher  $A_i$

```
1 begin
2    $k := 0$ ;
3   while true do
4     if  $f(k) == i$  then
5       afficher( $k$ )
6      $k := k + 1$ 
```

Seulement, on ne sait pas si  $f(k)$  est calculable. Par exemple, on pourrait avoir  $f(10^{38}) = i$ ,  $f$  qui boucle pour  $k = 10^{39}$ , et  $f(10^{40}) = i$ . Dans ce cas,  $10^{40}$  ne sera jamais affiché alors qu'il fait bel et bien partie de  $A_i$ . Cet algorithme est donc faux.

Ainsi, c'est l'algorithme suivant qui affiche  $A_i$  :

<b>Algorithm 5:</b> Afficher $A_i$	
1	<b>begin</b>
2	afficher( $2^i$ );
3	$L := \emptyset$ ;
4	<b>if</b> $(2^i - 1) \bmod 3 = 0$ <b>then</b>
5	$L = \{(2^i - 1)/3\}$
6	<b>while</b> $L \neq \emptyset$ <b>do</b>
7	$n := \text{tete}(L)$ ;
8	afficher( $n$ );
9	$L := \text{queue}(L)$ ;
10	$L := \text{ajouter}(2n, L)$ ;
11	<b>if</b> $\frac{n-1}{3}$ est impair <b>then</b>
12	$L := \text{ajouter}((n-1)/3, L)$

3. Comme  $A_1$  et  $A_2$  sont finis ( $(2^i - 1) \bmod 3 \neq 0$ ), il suffit d'afficher  $2^1$  et  $2^2$  (et 1 si on considère que  $A_0$  n'existe pas).
4. Comme  $A_4$  et  $A_6$  sont infinis, on ne peut pas afficher  $A_4$  puis  $A_6$ , car on n'afficherait que l'un ou l'autre, et pas les deux. Ainsi, il faut alterner entre élément de  $A_4$  et élément de  $A_6$ .

### 1.3 Diagonalisation

La diagonalisation est un procédé assez intuitif une fois qu'il est compris. On utilise cette technique pour prouver qu'il n'existe pas d'énumération d'ensembles infinis. La procédure est simple : il suffit de poser un tableau qui énumère tous les ensembles  $e_i$  possibles, puis de prendre un ensemble contradictoire : celui qui prend la valeur  $v_i$  de l'ensemble  $e_i$ . Comme notre tableau énumère tous les ensembles possibles, cela veut dire que l'ensemble contradictoire est dans le tableau. Or, si c'est le  $k$ -ème, on aura  $v_k$  de  $e_k = v_k$  et  $v_k$  de  $e_k \neq v_k$ , ce qui amène à une contradiction !

#### Exercice 10 - Diagonalisation

1. Montrer que l'ensemble des parties d'un ensemble  $E$  infini dénombrable n'est pas dénombrable.
2. Que peut-on conclure sur la cardinalité de l'ensemble des fonctions ? Et de l'ensemble des programmes ?
3. Préciser le cas où  $E$  est un ensemble fini (donc dénombrable) ?

1. Soit  $\mathcal{P}(E)$  l'ensemble des parties de  $E$ . Supposons que tout élément de  $\mathcal{P}(E)$  est représentable par  $(P_i)_{i \in \mathbb{N}}$ . Soit  $\mathcal{X}_E$  la fonction caractéristique de  $E$ ,  $\mathcal{X}_E : E \rightarrow \{0, 1\}$ . On peut construire le tableau suivant :

	$e_0$	$e_1$	$e_2$	$\dots$	$e_i$	$\dots$
$P_0$	0	1	0	$\dots$	1	$\dots$
$P_1$	1	1	0	$\dots$	1	$\dots$
$P_2$	0	0	0	$\dots$	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$P_i$	1	0	1	$\dots$	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

On prend l'ensemble  $P \in \mathcal{P}(E)$  tel que  $P = \{i \mid e_i \notin P_i\}$ . On suppose que  $\exists i. P = P_i$ , car  $\forall k \in \mathbb{N}. P_k \in \mathcal{P}(E)$ . Si  $P = P_i$ , alors si  $e_i \in P_i$ ,  $e_i \notin P$  par construction, donc  $P \neq P_i$ . De même, si  $e_i \notin P_i$ , par construction,  $e_i \in P$ , donc  $P \neq P_i$ . On ne peut pas trouver  $i$  tel que  $P = P_i$ , donc le tableau ne contient pas tous les éléments de  $\mathcal{P}(E)$ , donc l'ensemble des parties d'un ensemble  $E$  infini dénombrable n'est pas dénombrable ( $|\mathcal{X}_E| > |\mathbb{N}|$ ).



2. L'ensemble des fonctions caractéristiques est un sous-ensemble de l'ensemble des fonctions :  $\mathcal{X} \subset \mathcal{F}$  car  $\mathcal{X} : \mathbb{N} \rightarrow \{0, 1\}$  et  $\mathcal{F} : \mathbb{N} \rightarrow \mathbb{N}$ . Comme  $|\mathcal{X}| > \mathbb{N}$  et  $|\mathcal{F}| > |\mathcal{X}|$  (car  $\mathcal{X} \subset \mathcal{F}$ ), on a  $|\mathcal{F}| > |\mathbb{N}|$ , donc l'ensemble des fonctions n'est pas dénombrable.

L'ensemble des programmes, lui, est dénombrable. On peut par exemple prendre le taille puis l'ordre lexicographique à taille égale. Vu qu'on travaille sur un alphabet fini, cet ensemble sera fini.

3. Si  $E$  est fini,  $\mathcal{P}(E)$  est dénombrable. En effet, on ne peut pas construire de tableau infini, on ne peut donc pas appliquer la technique de diagonalisation et on conclut pratiquement immédiatement que c'est dénombrable.

### Exercice 11 - Diagonalisation

Montrer que l'ensemble des sous-ensembles d'un ensemble dénombrable n'est pas dénombrable. Pour cela, considérer un ensemble dénombrable  $A = \{a_0, a_1, a_2, \dots\}$ , et  $S$  l'ensemble de ses sous-ensembles.

L'ensemble des sous-ensembles d'un ensemble est l'ensemble des parties d'un ensemble  $\mathcal{P}(A)$ . On vient de démontrer cette exacte propriété dans l'exercice précédent.

### Exercice 12 - Diagonalisation

1. Soit une suite quelconque d'ensembles  $E_i \subset \mathbb{N}$ . Construire un ensemble qui n'appartient pas à cette suite (en vous inspirant de la diagonalisation).
2. Que pouvons nous conclure sur l'ensemble des sous-ensembles de  $\mathbb{N}$ ?

1. On prend l'ensemble  $E = \{e_i \mid \mathcal{X}_{E_i}(e_i) = 0\}$ . Cet ensemble est différent de tous les ensembles de  $E_i$ , et n'est donc pas dans la suite.
2. L'ensemble des sous-ensembles de  $\mathbb{N}$  n'est pas dénombrable.

### Exercice 13 - Diagonalisation

Montrer que  $[0, 1[$  n'est pas dénombrable.

On travaille en binaire dans la suite de cet exercice. On pose la fonction  $d_i(x)$  qui renvoie le  $i$ -ème décimal après la virgule de  $x$ . Si  $[0, 1[$  est dénombrable, on peut construire un tableau qui contient tous les éléments de  $[0, 1[$ . On prend tous les éléments qui ont des chiffres après la virgule ( $x_i = \sum_{k=0}^{\infty} d_k(x_i) \times 2^{-k-1}$ ) :

	$d_0$	$d_1$	$d_2$	$\dots$	$d_i$	$\dots$
$x_0$	<span style="border: 1px solid black;">1</span>	1	0	$\dots$	1	$\dots$
$x_1$	1	<span style="border: 1px solid black;">1</span>	0	$\dots$	1	$\dots$
$x_2$	0	0	<span style="border: 1px solid black;">0</span>	$\dots$	0	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$x_i$	1	0	1	$\dots$	<span style="border: 1px solid black;">1</span>	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Soit le réel  $x$  suivant :  $x = \sum_{k=0}^{\infty} ((-d_k(x_k) + 1) \times 2^{-k-1})$ , autrement dit,  $d_k(x) = 0$  si  $d_k(x_k) = 1$  et  $d_k(x) = 1$  sinon. Comme le tableau contient tous les éléments de  $[0, 1[$ ,  $\exists i. x = x_i$ . Or,  $d_i(x_i) \neq d_i(x)$  par construction, donc  $x \neq x_i$ . Contradiction, il n'y a pas de  $i$  tel que  $x = x_i$ , et dans ce cas, le tableau n'énumère pas tous les éléments de  $[0, 1[$ , donc  $[0, 1[$  n'est pas dénombrable.

## Exercice 14 - Diagonalisation

On considère l'ensemble  $U$  des suites  $(u_n)_{n \in \mathbb{N}}$  à la valeur dans  $\{0, 1\}$ , c'est à dire  $\forall n \in \mathbb{N}$ . Montrer que  $U$  n'est pas dénombrable.

	$u_0$	$u_1$	$u_2$	$\dots$	$u_i$	$\dots$
$U_0$	<span style="border: 1px solid black;">0</span>	1	0	$\dots$	0	$\dots$
$U_1$	0	<span style="border: 1px solid black;">1</span>	1	$\dots$	1	$\dots$
$U_2$	1	1	<span style="border: 1px solid black;">1</span>	$\dots$	1	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$U_i$	0	1	0	$\dots$	<span style="border: 1px solid black;">1</span>	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Soit la suite  $V$  suivante :  $V(i) = 0$  si  $U_i(i) = 1$  et  $V(i) = 1$  sinon. On suppose  $\exists i. V = U_i$ . Or,  $U_i(i) \neq V(i)$  par construction. Donc,  $V \neq U_i$ , et ainsi l'ensemble  $U$  n'est pas dénombrable.

## 1.4 Dénombrabilité

### Exercice 15 - Ensemble fini/infini

Un ensemble est fini si on ne peut pas le mettre en bijection avec une partie stricte de lui-même. Il est infini sinon.  
Montrer que l'ensemble des entiers est infini.

On peut mettre l'ensemble des entiers en bijection avec une partie stricte de lui-même : soit la fonction  $f : \mathbb{N} \rightarrow \mathbb{N}^*$  où  $f(n) = n + 1$ .  $f$  est en bijection avec une partie stricte de  $\mathbb{N}$ . On peut exhiber la fonction inverse  $q : \mathbb{N}^* \rightarrow \mathbb{N}$  avec  $q(n) = n - 1$ . L'ensemble des entiers est donc bien infini.

### Exercice 16 - Taille des ensembles

Soit  $E$  un ensemble, et soit  $\mathcal{P}(E)$  l'ensemble des parties de  $E$ . Montrer que  $|E| < |\mathcal{P}(E)|$ .

On sait que  $|E| \neq |\mathcal{P}(E)|$ . On veut montrer que  $|E| \leq |\mathcal{P}(E)|$ . Pour ce faire, il suffit de trouver  $\varphi : E \rightarrow \mathcal{P}(E)$  injective, ou bien  $\varphi : \mathcal{P}(E) \rightarrow E$  surjective. On prend par exemple  $\varphi : E \rightarrow \mathcal{P}(E)$  avec  $\varphi(e) = \{e\}$ .

## Exercice 17 - Dénombrabilité

1. Donner les bijections :
  - (a) de  $\mathbb{N}$  sur  $\mathbb{N} - \{0\}$ .
  - (b) de  $\mathbb{N}$  sur  $2\mathbb{N}$ .
  - (c) de  $\mathbb{N}$  sur  $\mathbb{Z}$ .
2. Est-ce que la fonction  $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$  est une bijection de  $\mathbb{N}$  sur  $\mathbb{Z}$  ?
3. Montrer que tout sous-ensemble  $X \subset \mathbb{N}$  est dénombrable.
4. Il existe une application  $f : X \rightarrow \mathbb{N}$  qui est injective si et seulement si  $X$  est dénombrable.
5. Un produit fini d'ensembles dénombrables est dénombrable.
6. Il existe une application  $f : \mathbb{N} \rightarrow X$  qui est surjective si et seulement si  $X$  est dénombrable.
7. Soit  $E$  un ensemble dénombrable infini. Alors il existe une bijection de  $\mathbb{N}$  sur  $E$ . Autrement dit, on peut numéroter les éléments de  $E$ , i.e. écrire  $E = \{e_0, e_1, \dots, e_n, \dots\}$ .
8. Montrer que  $\mathbb{Q}$  est dénombrable.
9. Soit  $(E_n)_{n \in \mathbb{N}}$  une famille dénombrable de sous-ensembles dénombrables d'un ensemble  $E$ . Montrer que la réunion  $\bigcup_{n \in \mathbb{N}} E_n$  est dénombrable.
10. Soit  $A = \mathbb{Q} \cap [0, 1]$  et  $B = \mathbb{Q} \cap ]0, 1[$ . Existe-t'il une bijection de  $A$  vers  $B$  ?

1. (a)  $\varphi : \mathbb{N} \rightarrow \mathbb{N} - \{0\}$  avec  $\varphi(n) = n + 1$ .  
 (b)  $\varphi : \mathbb{N} \rightarrow 2\mathbb{N}$  avec  $\varphi(n) = 2n$ .  
 (c)  $\varphi : \mathbb{N} \rightarrow \mathbb{Z}$  avec  $\varphi(n) = \frac{-(n+1)}{2}$  si  $n$  est impair et  $\varphi(n) = \frac{n}{2}$  sinon.
2. Oui, cette fonction est une bijection de  $\mathbb{N}$  sur  $\mathbb{Z}$ , c'est celle que j'ai décrit en 1.(c).
3. Pour montrer que tout sous-ensemble  $X \subset \mathbb{N}$  est dénombrable, il suffit d'exhiber une bijection entre  $X$  et  $\mathbb{N}$ . Soit  $X = \{x_0, x_1, \dots, x_i, \dots\}$ . Soit  $f : X \rightarrow \mathbb{N}$  l'application suivante :  $f(x_i) = i$ . On a  $f^{-1}(i) = x_i$ . Tout sous-ensemble de  $\mathbb{N}$  est donc bien dénombrable.
4.  $\Rightarrow$  On suppose qu'il existe une application  $f : X \rightarrow \mathbb{N}$  injective, et on veut montrer que  $X$  est dénombrable. On sait que  $|X| \leq \mathbb{N}$  par l'existence de la fonction injective. Ainsi, il existe une fonction injective  $g : \mathbb{N} \rightarrow X$  telle que  $g(i) = x_i$  (avec  $x_i \in X, \forall i$ ). Par le théorème de Cantor-Bernstein, il existe donc une bijection entre  $X$  et  $\mathbb{N}$ , ce qui montre que  $X$  est dénombrable.  
 $\Leftarrow$   $X$  est dénombrable, donc il existe une bijection entre  $X$  et  $\mathbb{N}$ . Ainsi, il existe deux fonctions injectives  $g : X \rightarrow \mathbb{N}$  et  $h : \mathbb{N} \rightarrow X$ . On pose  $g = f$  qui vérifie l'existence d'une application injective de  $X$  dans  $\mathbb{N}$ .
5. Pour prouver que le produit fini d'ensembles dénombrables est dénombrables, prenons  $E = \{E_0, E_1, \dots, E_n\}$  ces ensembles dénombrables. Soient  $F = \{f_0, f_1, \dots, f_n\}$  les bijections de ces ensembles. On peut exhiber la bijection suivante :  $h(x_0, x_1, \dots, x_n) = c(\dots(c(c(f_0(x_0), f_1(x_1)), \dots), f_n(x_n)), \dots)$ , avec  $c$  la fonction qui met en bijection  $\mathbb{N}^2$  avec  $\mathbb{N}$ . On a exhibé une bijection entre  $E_0 \times E_1 \times \dots \times E_n$  et  $\mathbb{N}$ , le produit cartésien des ensembles de  $E$  est donc dénombrable.
6.  $\Rightarrow$  On suppose qu'il existe une application  $f : \mathbb{N} \rightarrow X$  surjective, et on veut montrer que  $X$  est dénombrable. C'est à dire que  $|X| \leq \mathbb{N}$ . De plus, si  $|X| \leq \mathbb{N}$ , alors il existe une application injective de  $X$  dans  $\mathbb{N}$ . Par la question (4),  $X$  est dénombrable.  
 $\Leftarrow$   $X$  est dénombrable, c'est à dire qu'il existe une fonction  $g : \mathbb{N} \rightarrow X$  bijective. Comme celle-ci est bijective, elle est injective et surjective. Soit  $f = g$ , cette application est bien surjective, donc il existe bien une application surjective  $f : \mathbb{N} \rightarrow X$  si  $X$  est dénombrable.
7. Oui, c'est la définition de dénombrable.

8. Évident (on a déjà exhibé la bijection de  $\mathbb{Q}$  avec  $\mathbb{N}$  précédemment, exercice 6).
9. On peut remarquer que  $|\bigcup_{n \in \mathbb{N}} E_n| \leq |\prod_{n \in \mathbb{N}} E_n|$  et que comme le second est dénombrable, le premier l'est aussi. De la même manière que ce dernier, on peut construire une bijection de  $E = \bigcup_{n \in \mathbb{N}} E_n$  dans  $\mathbb{N}$  en diagonalisant les éléments des ensembles : soit  $e_{i,j}$  le  $j$ -ème élément du  $i$ -ème ensemble de  $E$ . On a :

$$\begin{array}{cccccc} & \vdots & \vdots & \vdots & \vdots & \vdots \\ e_{0,j} & e_{1,j} & \dots & e_{i,j} & \dots & \\ & \vdots & \vdots & \vdots & \vdots & \vdots \\ e_{0,1} & e_{1,1} & \dots & e_{i,1} & \dots & \\ e_{0,0} & e_{1,0} & \dots & e_{i,0} & \dots & \end{array}$$

En appliquant l'application inverse  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ , on associe chaque élément  $e_{i,j}$  à un entier de  $\mathbb{N}$ .

10. On peut exhiber la bijection suivante :  $f : \mathbb{Q} \cap [0, 1] \rightarrow \mathbb{Q} \cap ]0, 1[$  avec  $f(x) = \frac{x+1}{3}$ . Pour  $x = 0$ ,  $f(x) = \frac{1}{3} \in ]0, 1[$  et pour  $x = 1$ ,  $f(x) = \frac{2}{3} \in ]0, 1[$ . De plus,  $\forall x \in \mathbb{Q} \cap ]0, 1[, f(x) = \frac{p+q}{3 \times q}$  et ainsi  $0 < f(x) < 1$ .

## 1.5 Fonctions (non)-calculables

### Exercice 18 - Calculabilité

Soit  $f : \mathbb{N} \rightarrow \{0, 1\}$  une fonction totale non calculable.

1. Rappeler la définition d'une fonction totale et d'une fonction non calculable.
2. Construire une fonction  $g : \mathbb{N} \rightarrow \mathbb{N}$  totale, croissante et non calculable à partir de  $f$ .

1. Une fonction totale est une fonction définie pour tout  $n$ . Une fonction non-calculable est une fonction qui ne peut pas être calculée par une procédure automatique. Autrement dit,  $f$  est non calculable si et seulement si  $\forall n. f(n) \downarrow, \nexists p \forall n. p(n) = f(n)$ .
2. On peut construire la fonction  $g$  suivante :  $g(n) = \sum_{i=0}^n f(i)$ . Cette fonction est forcément croissante, car l'ensemble d'arrivée de  $f$  est  $\{0, 1\}$  (donc  $f(i) + f(i-1) \geq f(i-1)$ ),  $\forall i. f(i) \downarrow$  par hypothèse, donc  $(\sum_{i=0}^n f(i)) \downarrow$ .  
Enfin, supposons que  $g$  soit calculable. On a  $g(n-1) = \sum_{i=0}^{n-1} f(i)$  et  $g(n) = \sum_{i=0}^n f(i)$ , donc  $f(n) = g(n) - g(n-1)$ , si  $g$  est calculable, alors  $f$  est calculable. Or,  $f$  n'est pas calculable, donc  $g$  ne l'est pas non plus (contraposée).

### Exercice 19 - Calculabilité

Montrer que l'inverse d'une fonction  $f$  calculable et bijective est calculable.

$f$  calculable, cela signifie qu'il existe une procédure automatique  $p$  telle que  $p(n) = f(n)$ . Si on trouve une procédure automatique  $q$  qui calcule  $g$ , alors  $g$  est calculable. Soit la procédure suivante :

```
int q(int n) {
    for (int i = 0; ; ++i) {
        if (p(i) == n)
            return i;
    }
}
```

Comme  $f$  est bijective et totale,  $q$  termine forcément : pour n'importe quelle entrée  $n$ ,  $p(n)$  termine, et comporte un unique antécédent dans  $\mathbb{N}$ .

Dans le cas où  $f$  est injective,  $q$  ne termine pas forcément : la procédure  $p$  peut boucler pour certains  $n$ . Dans le cas où  $f$  est surjective,  $q$  renverra  $\min\{x \mid p(x) = n\}$  le minimum des antécédents de  $n$  par  $p$ .

### Exercice 20 - Calculabilité

Montrer qu'une fonction  $f$  totale  $\mathbb{N} \rightarrow \mathbb{N}$  est calculable si et seulement si son graphe

$$G = \{(x, f(x)) \mid x \in \mathbb{N}\}$$

est décidable.

Si  $E \subset \mathbb{N}$  est décidable, c'est qu'on peut écrire une procédure  $p$  qui calcule la fonction caractéristique  $\mathcal{X}_E$  de cet ensemble :  $\exists p \forall n \in \mathbb{N}. p(n) = \begin{cases} 1 & \text{si } n \in E \\ 0 & \text{sinon} \end{cases}$

$\Rightarrow$  On suppose que  $f$  totale est calculable. Montrons que le graphe  $G_f$  de  $f$  est décidable. La procédure  $p$  calcule  $f$ . Soit  $q$  la procédure qui calcule la fonction caractéristique  $\mathcal{X}_{G_f}$  :

```
int q(int x, int y) {
    return y == p(x);
}
```

Comme  $p$  est calculable,  $q$  l'est aussi. On a trouvé une procédure  $q$  qui calcule la fonction caractéristique  $\mathcal{X}_{G_f}$ , donc si  $f$  est calculable, alors  $G_f$  est décidable.

$\Leftarrow$  On suppose que  $G_f$  est décidable. Montrons que la procédure  $f$  du graphe  $G_f$  est calculable. La procédure  $q$  calcule la fonction caractéristique  $\mathcal{X}_{G_f}$ . Soit  $p$  la procédure qui calcule  $f$  :

```
int p(int x) {
    for (int i = 0; ; ++i) {
        if (q(x, i)) return i;
    }
}
```

Comme  $q$  est calculable, alors  $p$  l'est aussi. On a trouvé une procédure  $p$  qui calcule  $f$ , donc si  $G_f$  est décidable, alors  $f$  est calculable.

### Exercice 21 - Calculabilité

Soit  $E$  un ensemble et  $\phi$  une fonction telle que  $\phi(n)$  est égale au nombre d'éléments de  $E$  strictement inférieurs à  $n$ . Montrer que  $\phi$  totale est calculable si et seulement si  $E$  est décidable.

$\Rightarrow$  On suppose que  $\phi$  est calculable. Montrons que  $E$  est décidable. La procédure  $p$  calcule  $\phi$ . Soit  $q$  la procédure qui calcule la fonction caractéristique  $\mathcal{X}_E$  :

```
int q(int x) {
    return p(x) != p(x + 1); // On pourrait aussi dire p(x) == p(x + 1) - 1
}
```

$p$  termine et définie pour tout  $x$ , donc  $q$  termine et définie pour tout  $x$ . On a trouvé une procédure  $q$  qui calcule la fonction caractéristique  $\mathcal{X}_E$ , donc si  $\phi$  calculable, alors  $E$  est décidable.

$\Leftarrow$  On suppose que  $E$  est décidable, c'est à dire qu'il existe une procédure automatique  $q$  telle que  $q$  calcule  $\mathcal{X}_E$ . Montrons que  $\phi$  est calculable. Soit  $p$  la procédure qui calcule  $\phi$  :

```

int p(int x) {
    if (x == 0) 0;
    if (x == 1) return q(0);
    return q(x - 1) + p(x - 1);
}

```

Comme  $q$  termine et définie pour tout  $n \in \mathbb{N}$ ,  $p$  termine. On a trouvé une procédure  $p$  qui calcule  $\phi$ , donc si  $E$  décidable, alors  $\phi$  est calculable.

## 1.6 Problèmes indécidables

### Exercice 22 - Variantes du problème de l'arrêt

1. SELF-HALT : le programme SELF-HALT( $p$ ) s'arrête sur  $p$  si HALT( $p, p$ ) s'arrête où HALT( $p, v$ ) désigne le problème de l'arrêt pour un programme  $p$  appliquée à des données  $v$ .
2. ANTI-SELF-HALT : le programme ANTI-SELF-HALT( $p$ ) s'arrête si et seulement si  $p$  ne s'arrête pas.
3. Montrer que les problèmes suivants sont indécidables.
  - (a) HALT $_{\exists}$  : le problème de l'arrêt existentiel, existe-t-il une entrée pour laquelle le programme s'arrête ?
  - (b) HALT $_{\forall}$  : le problème de l'arrêt universel, le programme s'arrête-t-il pour toutes les entrées ? La réduction est la même que la précédente.
  - (c) NEGVAL : le problème du test de valeur négative, la variable  $v$  du programme prend-elle une valeur négative au cours du calcul ? Ce problème est à rapprocher de « cet indice de tableau évolue-t-il toujours dans les bornes du tableau ? »
  - (d) EQUIV : problème du test d'équivalence de programmes, les programmes  $P_1$  et  $P_2$  ont-ils le même comportement pour toutes les entrées ?  
Pour illustrer l'intérêt de ce problème : on peut se poser la question concernant d'un programme source et d'un programme objet correspondant produit par un compilateur. Certaines optimisations tendantes changent le comportement du programme.
  - (e) RETURN $_0$  : problème du test de rendu nul, existe-t-il une entrée pour laquelle le programme retourne la valeur 0.

1. Par définition, le problème SELF-HALT est indécidable.
2. Par définition, le problème ANTI-SELF-HALT est indécidable.
3. (a) Supposons l'existence de la procédure automatique  $p_{HE}$  qui résout HALT $_{\exists}$ , c'est à dire que  $\forall p \in P. p_{HE}(p) = 1$  s'il existe une entrée  $v$  telle que  $p(v)$  s'arrête et 0 sinon. Soit  $\gamma$  la procédure automatique complémentaire de HALT $_{\exists}$  suivante :

```

int gamma(procedure x) {
    if (p_HE(x)) while(1) ;
    else return 0;
}

```

$\gamma$  prend en entrée une procédure automatique, et si pour une donnée  $v$  cette procédure automatique termine, alors  $\gamma$  ne termine pas. Si la procédure automatique ne termine pour aucune donnée, alors  $\gamma$  termine.

Si  $p_{HE}$  existe, alors  $\gamma$  existe. Si on appelle  $\gamma(\gamma)$ , que se passe-t-il ? Si  $\gamma$  termine pour une donnée, il va renvoyer 0. Seulement, s'il fait cela, c'est que  $\gamma$  ne s'arrête pour aucune donnée.

Si **gamma** ne s'arrête pas, c'est à dire qu'il rentre dans une boucle infinie, alors il retourne 0 et alors **gamma** s'arrête.

Par contradiction, **gamma** ne peut pas exister, et ainsi,  $p_{HE}$  n'existe pas non plus.

- (b) Supposons l'existence de la procédure automatique  $p_{HA}$  qui résout  $HALT_{\forall}$ , c'est à dire que  $\forall p \in P. p_{HA}(p) = 1$  si  $p$  s'arrête pour toutes les entrées  $v$  et  $p_{HA}(p) = 0$  sinon. Soit **gamma** la procédure automatique complémentaire de  $HALT_{\forall}$  suivante :

```
int gamma(procedure x) {
    if (p_HA(x)) while (1) ;
    else return 0;
}
```

**gamma** prend en entrée une procédure automatique, et si pour toute donnée  $v$  cette procédure automatique termine, alors **gamma** ne termine pas. Si la procédure automatique ne termine pas pour une donnée, alors **gamma** termine.

Si  $p_{HA}$  existe, alors **gamma** existe. Si on appelle **gamma**(**gamma**), que se passe-t-il ? Si **gamma** termine pour toute donnée, alors par définition, **gamma** boucle et ainsi  $p_{HA}(\text{gamma}) = 0$ .

Si **gamma** boucle pour une donnée, alors  $p_{HA}(\text{gamma}) = 0$  et **gamma** termine, donc  $p_{HA}(\text{gamma}) = 1$ .

Par contradiction, **gamma** ne peut pas exister, et ainsi,  $p_{HA}$  n'existe pas non plus.

- (c) Supposons l'existence de la procédure automatique  $p_{NV}$  qui résout  $NEGVAL$ , c'est à dire que  $\forall p \in P. p_{NV}(p, v) = 1$  si  $v$  prend une valeur négative au cours du calcul, 0 sinon. Soit **gamma** la procédure automatique complémentaire de  $NEGVAL$  suivante :

```
void gamma(procedure x, int v) {
    if (!p_NV(x, v)) {
        v = -1;
        return;
    }
}
```

Si on appelle **gamma**(**gamma**, 1), il y a deux cas :

- Si  $v$  ne prend pas de valeur négative durant le calcul, alors  $v = -1$  et  $v$  prend une valeur négative pendant le calcul de **gamma**, ce qui est une contradiction.
- Si  $v$  prend une valeur négative durant le calcul, alors  $v$  n'est pas touchée donc  $v = 1$  et  $v$  ne prend pas de valeur négative durant le calcul de **gamma**, ce qui est une contradiction.

**gamma** ne peut pas être calculée, donc  $p_{NV}$  ne peut pas l'être non plus.

- (d) Supposons l'existence de la procédure automatique  $p_{EQ}$  qui résout  $EQUIV$ , c'est à dire que  $\forall p, q \in P. p_{EQ}(p, q) = 1$  si  $p$  et  $q$  sont équivalents à chaque étape de calcul, 0 sinon. Soit **gamma** la procédure automatique complémentaire de  $EQUIV$  suivante :

```
int gamma(procedure p, procedure q) {
    if (p_EQ(p, q)) return random();
    return 1;
}
```

Si on appelle **gamma**(**gamma**, **gamma**), il y a deux cas :

- Si **gamma** et **gamma** sont équivalents à chaque étape de calcul, alors **gamma** et **gamma** renvoient une valeur différente ( $\frac{1}{\infty} \approx 0$  chances d'avoir un entier égal), ce qui est une contradiction.
- Si **gamma** et **gamma** ne sont pas équivalents à chaque étape de calcul, alors **gamma** et **gamma** renvoient 1, et sont équivalents à chaque étape de calcul, ce qui est une contradiction.

**gamma** ne peut pas être calculé, donc  $p_{EQ}$  ne peut pas l'être non plus.

- (e) Supposons l'existence de la procédure automatique  $p_{R0}$  qui résout  $\text{RETURN}_0$ , c'est à dire que  $\forall p \in P. p_{R0}(p, v) = 1$  si  $p$  renvoie 0 sur l'entrée  $v$ , 0 sinon. Soit **gamma** la procédure automatique complémentaire de  $\text{RETURN}_0$  suivante :

```
int gamma(procedure x, int v) {
    if (p_R0(x, v)) return 1;
    else return 0;
}
```

Si on appelle  $\text{gamma}(\text{gamma}, 0)$ , il y a deux cas :

- Si **gamma** renvoie 0, alors **gamma** va renvoyer 1, ce qui est une contradiction.
- Si **gamma** renvoie 1, alors **gamma** va renvoyer 0, ce qui est une contradiction.

**gamma** ne peut pas être calculée, donc  $p_{R0}$  non plus.

## 1.7 Théorème de Rice

### Exercice 23 - Calculabilité

En vous inspirant du théorème de Rice, donnez le prédicat (indécidable) et la fonction contradictoire qui prouve par l'absurde le résultat d'indécidabilité pour chacun des exemples suivants : on ne peut décider si une procédure calcule

1. une fonction totale
2. une fonction injective
3. une fonction croissante
4. une fonction à valeurs bornées

1. Soit le prédicat  $P$  tel que  $P(p) = 1$  si  $p$  calcule une fonction totale, et 0 sinon. Le prédicat n'est pas trivial : soit **rac** la procédure qui ne calcule pas une fonction totale :

```
int rac(int n) {
    r = 0;
    while (r * r != n) r += 1;
    return r;
}
```

$P(\text{rac}) = 0$ , car si un nombre n'a pas de racine carrée entière, alors cette procédure ne s'arrêtera pas, elle n'est pas totale. Soit la procédure **id** qui calcule une fonction totale :

```
int id(int n) { return n; }
```

$P(\text{id}) = 1$ , car la procédure automatique termine pour n'importe quel élément de  $\mathbb{N}$ . Le prédicat  $P$  n'est pas trivial, donc on ne peut pas décider si une procédure calcule une fonction totale.

2. Soit le prédicat  $P$  tel que  $P(p) = 1$  si  $p$  calcule une fonction injective, et 0 sinon. Le prédicat n'est pas trivial : soit  $p_0$  la procédure qui ne calcule pas une fonction injective :

```
int p_0(int n) {
    if (n % 2 == 0) return n;
    else return 2*n;
}
```

$P(p_0) = 0$ , car  $p_0(3) = p_0(6) = 6$ , la fonction a deux antécédents pour la même image, elle n'est donc pas injective. Soit la procédure **id** qui calcule une fonction injective :

```
int id(int n) { return n; }
```



$P(\text{id}) = 1$ , car la procédure automatique associe une seule image pour chaque antécédent de  $\mathbb{N}$ . Le prédicat  $P$  n'est pas trivial, donc on ne peut pas décider si une procédure calcule une fonction injective.

3. Soit le prédicat  $P$  tel que  $P(p) = 1$  si  $p$  calcule une fonction croissante, et 0 sinon. Le prédicat n'est pas trivial : soit  $p_0$  la procédure qui ne calcule pas une fonction croissante :

```
int p_0(int n) {
    return n % 2;
}
```

$P(p_0) = 0$ , car  $p_0(2) = 0 < p_0(1) = 1$ , la fonction n'est donc pas croissante. Soit la procédure  $\text{id}$  qui calcule une fonction croissante :

```
int id(int n) { return n; }
```

$P(\text{id}) = 1$ , car  $\text{id}(n) = \text{id}(n-1) + 1$ . Le prédicat  $P$  n'est pas trivial, donc on ne peut pas décider si une procédure calcule une fonction croissante.

4. Soit le prédicat  $P$  tel que  $P(p) = 1$  si  $p$  calcule une fonction à valeurs bornées, et 0 sinon. Le prédicat n'est pas trivial : soit  $\text{id}$  la procédure qui ne calcule pas une fonction à valeurs bornées :

```
int id(int n) { return n; }
```

$P(\text{id}) = 0$ , car on peut toujours trouver une valeur de  $\text{id}$  supérieure à une valeur  $k$  fixée, la fonction n'est donc pas à valeurs bornées. Soit la procédure  $p_1$  qui calcule une fonction à valeurs bornées :

```
int p_1(int n) { return n % 17; }
```

$P(p_1) = 1$ , car  $0 \leq p_1(n) < 17$ . Le prédicat  $P$  n'est pas trivial, donc on ne peut pas décider si une procédure calcule une fonction à valeurs bornées.

## 1.8 Décidabilité et récursivement énumérable

### Exercice 24 - Récursivement énumérable

Soit  $A$  un ensemble énumérable et  $P : A \rightarrow \text{Bool}$  un programme total tel que  $\forall a \in A. P(a) \downarrow$ . Alors l'ensemble  $B := \{a \in A \mid P(a) = \text{Vrai}\}$  est énumérable.

On rappelle que si  $E$  est énumérable, cela signifie qu'il existe une procédure  $p_E$  qui affiche tous les éléments de  $E$ . Si on peut construire cette procédure, on obtient ce qu'on appelle une preuve constructive.

$A$  est énumérable, c'est à dire qu'il y a une procédure  $p_A$  qui affiche tous les éléments de  $A$ . On pose la procédure  $q_A : \mathbb{N} \rightarrow \text{Bool} : \begin{cases} 1 & \text{si } n \in \mathbb{N} \text{ est affiché par } p_A \\ 0 & \text{sinon} \end{cases}$

Cette procédure termine : comme tous les éléments de  $A$  sont affichés par  $p_A$  dans l'ordre  $(a_0 < a_1 < \dots < a_i < \dots)$ , si on trouve  $k$  tel que  $n = a_k$ ,  $q_A$  renvoie vrai, et si on ne trouve pas ce  $k$  mais que pour un certain  $i$ ,  $a_i > n$ ,  $q_A$  renvoie faux. On peut alors construire le programme  $p_B$  qui énumère  $B$  suivant :

```
void p_B() {
    for (int i = 0; ; ++i) {
        if (q_A(i) && P(i)) {
            afficher(i);
        }
    }
}
```

La procédure  $p_B$  termine car  $q_A(n)$  totale et termine  $\forall n \in \mathbb{N}$  et renvoie vrai si et seulement si  $n \in A$ , et  $\forall a \in A. P(a) \downarrow$ .

## Exercice 25 - Calculabilité

Soit  $E$  l'ensemble  $val(f)$  où  $f$  est calculable et partielle.

Montrer que  $E$  est récursivement énumérable (inspirez-vous du fait que l'arrêt en  $t$  unités de temps est décidable).

Comme  $f$  est calculable, on sait qu'il existe une procédure automatique  $p_f : \mathbb{N} \rightarrow \mathbb{N}$  qui termine pour tout  $n \in D_f$ . Soit  $A = \{n \in \mathbb{N} | p(n) \downarrow\}$  et  $E = \{p_f(n) | p(n) \text{ s'arrête en } n \text{ étapes}\}$ . On a  $E \subseteq A$ . Montrons d'abord que  $A$  est énumérable. Il suffit de trouver la procédure automatique  $p_A$  qui énumère  $A$ . Comme  $f$  est partielle, si on envoie  $m \notin D_f$  à  $p_f$ , on ne sait pas si la procédure termine. Ainsi, on définit une surcouche en étapes de  $p_f : p_{f_t} : \mathbb{N} \times \mathbb{N} \rightarrow \text{Bool}$ .  $p_{f_t}(n, t)$  renvoie vrai si  $p_f(n)$  termine en  $t$  étapes, faux sinon. On peut ainsi définir  $p_A$  :

```
void p_A() {
    for (int i = 0; ; ++i) {
        for (int k = 0; k <= i; ++k) {
            if (p_f_t(k, i))
                afficher p_f(k);
        }
    }
}
```

Comme  $p_{f_t}$  termine pour n'importe quelles valeurs et que  $p_f$  termine en particulier pour les valeurs où  $p_{f_t}$  termine en un nombre d'étapes finies,  $p_A$  énumère bien tous les éléments de  $A$ . On peut cependant remarquer que le même élément peut être affiché plusieurs fois.

La procédure automatique  $p_B$  est assez similaire à  $p_A$ , et énumère tous les éléments de  $B$  pour les mêmes raisons :

```
void p_B() {
    for (int i = 0; ; ++i) {
        if (p_f_t(i, i)) {
            afficher p_f(i);
        }
    }
}
```

## Exercice 26 - Calculabilité

Soit  $f$  une fonction calculable, un ensemble  $B$  et son image réciproque par  $f$ ,  $A$  :

$$A = f^{-1}(B) = \{x | f(x) \in B\}$$

1. Rappeler la définition d'un ensemble décidable et d'un ensemble récursivement énumérable.
2. A-t-on  $B$  décidable implique  $A$  décidable ?
3. A-t-on  $B$  récursivement énumérable implique  $A$  récursivement énumérable ?

1. Un ensemble est décidable s'il y a une procédure automatique qui calcule sa fonction caractéristique. Un ensemble est récursivement énumérable s'il y a une procédure automatique qui calcule sa fonction semi-caractéristique (donc énumère tous les éléments de l'ensemble).
2.  $A$  est décidable si et seulement si  $f$  est totale. En effet, dans ce cas, on peut écrire la procédure suivante : `bool X_A(int n) { return X_B(p_f(n)); }` (avec  $X_B$  la fonction caractéristique de  $B$  et  $p_f$  la procédure automatique qui calcule la fonction  $f$ ). On voit bien que si  $f$  n'est pas totale,  $X_A$  ne termine pas toujours.

3. En utilisant une surcouverte de  $p_f$  avec le nombre d'étapes, comme à l'exercice précédent, et qu'on définit  $q_B$  de manière analogue à l'exercice 24 ( $q_B : \mathbb{N} \rightarrow \text{Bool} : \begin{cases} 1 & \text{si } n \in \mathbb{N} \text{ est affiché par } p_B \\ 0 & \text{sinon} \end{cases}$ ) si  $B$  est récursivement énumérable, alors  $A$  l'est aussi :

```
void p_A() {
    for (int i = 0; ; ++i) {
        for (int k = 0; k <= i; ++k) {
            if (q_B(f(i, k)))
                afficher(i)
        }
    }
}
```

### Exercice 27 - Calculabilité

1. Montrer qu'un ensemble énuméré par une fonction calculable strictement croissante  $f$  est décidable.
2. En déduire que tout ensemble récursivement énumérable non décidable contient un sous-ensemble infini et décidable.

1.  $f$  est calculable, et par définition, comme l'ensemble est énuméré par une fonction, celle-ci est totale. On peut donc écrire la fonction caractéristique  $\mathcal{X}_E$  de l'ensemble  $E$  énuméré par  $f$  :

```
bool X_E(int n) {
    for (int i = 0; ; ++i) {
        if (p_f(i) == n) return true;
        if (p_f(i) > n) return false;
    }
}
```

2. Si l'ensemble  $A$  est énumérable non décidable, ça veut dire qu'il existe une procédure automatique  $p_A$  qui affiche tous les éléments de cet ensemble. On suppose que  $p_f : \mathbb{N} \rightarrow \mathbb{N}$  stocke les éléments de  $A$  au fur et à mesure de l'énumération, et qu'à chaque fois que la procédure est appelée, le premier élément stocké est renvoyé et supprimé. On peut écrire la procédure automatique suivante qui calcule la fonction caractéristique d'un sous-ensemble infini décidable :

```
bool X_B(int n) {
    int dernier = 0;
    for (int i = 0; ; ++i) {
        if (p_f(i) > dernier) {
            if (p_f(i) == n) return true;
            if (p_f(i) > n) return false;
            dernier = p_f(i);
        }
    }
}
```

### Exercice 28 - Calculabilité

Montrer que tout ensemble récursivement énumérable peut-être énuméré par une fonction sans répétition.

Il suffit de garder en mémoire tous les éléments de l'ensemble qui ont été affichés, et si un élément l'a déjà été, on ne l'affiche pas.

### Exercice 29 - Calculabilité

Soient  $A$  et  $B$  deux ensembles décidables.

1. Est-on sûrs que le complémentaire de  $A$  ( $\bar{A}$ ) est décidable ?
2. Est-on sûrs que l'union de  $A$  et  $B$  est décidable ?
3. Est-on sûrs que l'intersection de  $A$  et  $B$  est décidable ?
4. Même question en remplaçant décidables par récursivement énumérables.

On suppose  $A$  et  $B$  décidables, il existe les procédures automatiques  $p_A$  et  $p_B$  telles que  $\forall n \in \mathbb{N}. p_A(n) \downarrow$  et  $p_B(n) \downarrow$  qui calculent la fonction caractéristique de ces deux ensembles.

1. Oui, on peut exhiber la procédure automatique qui correspond à la fonction caractéristique de  $\bar{A}$  :  

```
bool p_Abar(int x) { return !p_A(x); }
```
2. Oui, on peut exhiber la procédure automatique qui correspond à la fonction caractéristique de  $A \cup B$  :  

```
bool p_AUB(int x) { return p_A(x) || p_B(x); }
```
3. Oui, on peut exhiber la procédure automatique qui correspond à la fonction caractéristique de  $A \cap B$  :  

```
bool p_AIB(int x) { return p_A(x) && p_B(x); }
```

On suppose maintenant  $A$  et  $B$  récursivement énumérables. C'est à dire qu'il existe deux procédures automatiques  $p_A$  et  $p_B$  qui affichent tous les éléments de  $A$  et respectivement  $B$  dans la sortie standard.

1. On ne peut rien dire sur  $\bar{A}$ . Cependant, si celui-ci est bien récursivement énumérable,  $A$  (et  $\bar{A}$ ) sont décidables :

```
bool P_Aprime(int n) {
    int nbrE = 1;
    while (1) {
        Calculer P_A en nbrE;
        if (P_A affiche n) { return true; }
        Calculer P_Abar en nbrE;
        if (P_Abar affiche n) { return false; }
        ++nbrE;
    }
}
```

2. Oui, on peut exhiber la fonction qui énumère tous les éléments de  $A \cup B$  :

```
void P_AUB() {
    int n = 0;
    while (1) {
        for (int i = 0; i < n; ++i) {
            if (P_A affiche i en n etapes) { afficher(i); }
            else if (P_B affiche i en n etapes) { afficher(i); }
        }
        ++n;
    }
}
```

3. En utilisant la même astuce, on peut afficher tous les éléments de  $A \cap B$  :

```

void P_AIB() {
    int n = 0;
    while (1) {
        for (int i = 0; i < n; ++i) {
            if ((P_A affiche i en n etapes) &&
                (P_B affiche i en n etapes)) {
                afficher(i);
            }
        }
        ++n;
    }
}

```

### Exercice 30 - Calculabilité

1. Soit  $A$  un ensemble décidable de couples d'entiers. Montrer que la projection de  $A$  à savoir  $E = \{x | \exists y. (x, y) \in A\}$  est récursivement énumérable.
2. Montrer que réciproquement, tout ensemble récursivement énumérable est la projection d'un ensemble décidable.

1. On peut exhiber la fonction semi-caractéristique  $p_E$  de  $E$  :

```

void p_E(int x) {
    for (int i = 0; ; ++i) {
        if (p_A(x, i)) return 1;
    }
}

```

Comme  $A$  est décidable, il est assez évident que cette procédure termine pour tout  $x$  qui a une image dans  $A$ , et elle ne termine pas sinon.

2. On peut construire  $p_B : \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{B}$  tel que vrai si  $(i, j) \in B$  et faux sinon :

```

bool p_B(int x, int y) {
    return P_A(x) == y;
}

```

### Exercice 31 - Concept de la réduction

Pour deux sous-ensembles  $A$  et  $B$  de  $\mathbb{N}$ , on dit que  $A$  se réduit à  $B$  (ce qu'on note  $A \propto B$ ) s'il existe une fonction totale  $f$  telle que  $\forall x \in \mathbb{N}, x \in A \Leftrightarrow f(x) \in B$ .

1. Montrer que si  $B$  est décidable et  $A \propto B$ , alors  $A$  est décidable.
2. Montrer que si  $B$  est récursivement énumérable et  $A \propto B$ , alors  $A$  est récursivement énumérable.

1. On peut exhiber la procédure automatique qui calcule la fonction caractéristique de  $A$  :

```

bool p_A(int x) {
    return p_B(p_f(x));
}

```

Comme  $B$  décidable,  $p_B$  termine. Comme  $f$  totale et calculable,  $p_f$  termine pour tout entier. Donc  $p_A$  termine pour tout entier, et  $p_A$  calcule la fonction caractéristique de  $A$ , donc  $A$  est décidable.

2. On peut exhiber la procédure automatique qui calcule la fonction semi-caractéristique de  $A$  :

```
bool p_A(int x) {
    return p_B(p_f(x));
}
```

Comme  $B$  récursivement énumérable,  $p_B$  termine si  $f(x) \in B$ . Comme  $f$  est totale et calculable,  $p_f$  termine pour tout entier. Donc,  $p_A$  termine pour tout entier  $n \in A$ .  $p_A$  calcule bien la fonction semi-caractéristique de  $A$ .

## 1.9 Sur le point fixe

### Exercice 32 - Exemples

Donner les points fixes pour les fonctions suivantes :

1.  $\mathbb{N} \rightarrow \mathbb{N}, x \mapsto x + 1$
2.  $\mathbb{N} \rightarrow \mathbb{N}, x \mapsto 0 * x$
3.  $\mathbb{N} \rightarrow \mathbb{N}, x \mapsto \begin{cases} x/2 & \text{si } x \text{ est pair} \\ x & \text{sinon} \end{cases}$
4.  $\mathbb{N} \rightarrow \mathbb{N}, x \mapsto \begin{cases} x & \text{si } x \geq n \\ x + 1 & \text{sinon} \end{cases}$
5.  $\mathbb{N}^* \rightarrow \mathbb{N}, x \mapsto \begin{cases} x & \text{si } x \geq n \\ x - 1 & \text{sinon} \end{cases}$
6.  $\mathbb{N}^* \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}, \langle x, y \rangle \mapsto \begin{cases} \langle x - 1, y + 1 \rangle & \text{si } x > 0 \\ \langle x, y \rangle & \text{sinon} \end{cases}$

Les points fixes sont l'ensemble  $F = \{x | x = f(x)\}$

1.  $F = \emptyset$
2.  $F = \{0\}$
3.  $F = \{x | x \bmod 2 \neq 0\}$
4.  $F = \{x | x \geq n\}$
5.  $F = \{x | x \geq n\}$
6.  $F = \{\langle 0, y \rangle | y \in \mathbb{N}\}$

## 2 Complexité

### 2.1 Rappel

#### Exercice 33 - Une certaine idée de la complexité

Soit la fonction C suivante :

```
int pf(int x) {  
    int y = pg(x);  
    return ph(y);  
}
```

1. Quelle est la complexité du calcul de **pf** si **pg** est de complexité  $O(n^4)$ , **ph** de complexité linéaire et si  $g(n) < n^2$  ( $g$  étant la fonction calculée par **pg**) ?
2. Si **ph** s'exécute en temps polynomial, à quelle condition le calcul de **pf** se fait-il en temps polynomial ?
3. Si les hypothèses de la question précédente est vérifiée, que peut-on en déduire si la fonction  $h$  calculée par **ph** se calcule en temps polynomial ?
4. Soit le calcul de Fibonacci en utilisant directement la formule de récurrence :  $f_0 = 1, f_1 = 1, \forall n > 1. f_n = f_{n-1} + f_{n-2}$ . Montrons que le nombre d'additions nécessaires pour faire le calcul est compris entre  $\sqrt{2}^n$  et  $2^n$  ?
5. Comment améliorer pour que ce nombre soit en  $O(n)$  ? Peut-on déduire qu'il existe un algorithme qui calcule  $f_n$  avec un nombre d'additions polynomial par rapport à la taille de la donnée ? Pourquoi ?
6. Questions difficiles : comment calculer  $f_n$  avec un nombre d'additions et de multiplications polynomial par rapport à la taille de la donnée ? Peut-on trouver un algorithme qui s'exécute en temps polynomial par rapport à la taille de la donnée ?

1. On sait que :

$$\begin{aligned}O(\text{pg}) &\subseteq O(n^4) \\O(\text{ph}) &\subseteq O(n) \\O(\text{pf}) &\subseteq O(\text{pg}) + O(\text{ph})\end{aligned}$$

De plus, comme  $g(n) < n^2$ , la taille de la donnée en entrée de **ph** est donc au plus de  $n^2$  de la taille de l'entrée de **pf**, donc  $O(\text{ph}(\text{pg})) = O(n^2)$ .

Ainsi :

$$\begin{aligned}O(\text{ph}) &\subseteq O(\text{pg}) + O(\text{ph}) \\&\subseteq O(n^4) + O(n^2) \\&\subseteq O(n^4)\end{aligned}$$

2. On sait que **ph** se calcule en temps polynomial. Pour que **pf** se calcule aussi en temps polynomial, il faut que :
  - $n$  soit de taille polynomiale ;
  - **pg** se calcule en temps polynomial ;
  - $g(n)$  renvoie une donnée de taille polynomiale.
3. Si la fonction  $h$  calculée par **ph** se calcule en temps polynomial, et que les hypothèses précédentes sont vérifiées, alors  $f$  se calcule en temps polynomial.

4. La propriété est fausse : on peut citer le contre-exemple suivant : pour  $n = 2$ ,  $f_2 = f_1 + f_0$ . Pour 0 et 1, il n'y a pas d'addition, donc le nombre d'additions pour  $f_2$  est 1. Or,  $\sqrt{2^2} = 2$  et  $2^2 = 4$ , et comme  $1 < \sqrt{2^2}$ , on n'a pas  $\sqrt{2^2} \not\leq 1 \not\leq 2^2$ .

On peut cependant affirmer que le nombre d'addition nécessaires reste en  $O(2^n)$  avec cette formule de récurrence (c'est assez évident si on représente les additions sous forme d'arbre binaire).

5. Pour que le nombre d'additions soit en  $O(n)$ , il suffit stocker chaque valeur de la suite dans un tableau, et faire appel à ces éléments pour calculer le nombre qui nous intéresse :

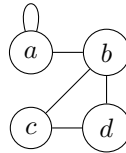
```
int fibonacci(int n) {
    int F[n]; F[0] = 1; F[1] = 1;
    for (int i = 2; i <= n; ++i)
        F[i] = F[i - 1] + F[i - 2];
    return F[n];
}
```

On ne peut cependant pas déduire qu'il existe un algorithme qui calcule  $f_n$  avec un nombre d'additions polynomial par rapport à la taille de la donnée. En effet, comme la taille est  $O(\log(n))$ , le nombre d'additions reste exponentiel : soit  $k = \log(n)$  la taille de  $n$ . L'algorithme s'exécute en temps  $O(n)$ , mais comme  $n = 2^k$ , sur la taille de l'entrée, cet algorithme fait  $O(2^k)$  additions. Le nombre d'additions est exponentiel par rapport à la taille de l'entrée.

### Exercice 34 - Sur le codage d'un graphe

Donner la taille en nombre de bits pour coder un graphe en utilisant une matrice d'adjacence et listes chaînées.

Soit un graphe  $G = (V, E)$  avec  $V$  les sommets et  $E$  les arêtes. Soit  $n = |V|$  et  $m = |E|$ . Ce graphe peut être représenté de deux manières : par une matrice d'adjacence  $\mathcal{M}_{n,n}$  où chaque case  $\mathcal{M}_{[i,j]}$  est 1 si  $(i, j) \in E$  et 0 sinon, ou bien un tableau de listes chaînées :  $T_{[i]}$  contient tous les sommets tels que  $(i, v) \in E$  (pour  $v \in V$ ). Par exemple, soit  $G_0$  le graphe non-orienté suivant :



Représentation graphique de  $G_0$

On a  $G_0 = (\{a, b, c, d\}, \{(a, a), (a, b), (b, c), (b, d), (c, d)\})$ . En donnant les indices suivants aux différents sommets :  $a = 0, b = 1, c = 2, d = 3$ , on peut définir la matrice  $\mathcal{M}_{G_0}$  et la liste d'adjacence  $\mathcal{L}_{G_0}$  :

$a$	1	1	0	0
$b$	1	0	1	1
$c$	0	1	0	1
$d$	0	1	1	0

$\mathcal{M}_{G_0}$

$a$	0	1	
$b$	0	2	3
$c$	1	3	
$d$	1	2	

$\mathcal{L}_{G_0}$

Le nombre de bits nécessaires pour coder  $\mathcal{M}_G$  pour n'importe quel graphe  $G$  est donc  $n^2$  (car la matrice sera toujours de taille  $n \times n$ , quelles que soient les arêtes), et le nombre de bits nécessaires pour coder  $\mathcal{L}_G$  est en  $O(n + m)$  (qui se réduit à  $n^2$  si le graphe est complet).

Le compromis est d'utiliser une matrice d'adjacence lorsque le graphe est dense, et une liste d'adjacence dans le cas contraire.



### Exercice 35 - Certificat

Si pour un problème  $\Pi$  vous avez un certificat polynomial pour une réponse positive et un certificat polynomial pour une réponse négative. Que pouvez-vous conclure ? Justifiez votre réponse.

On connaît un algorithme simple en  $O(\sqrt{n})$  pour savoir si un nombre  $n$  est premier. Peut-on en déduire que savoir si un nombre est premier s'exécute en temps sous-linéaire ?

On peut savoir si  $n$  peut s'écrire comme le produit de deux nombres premiers et on connaît un algorithme en  $O(\sqrt{n})$ . Peut-on en déduire que ce problème est dans  $P$  ? Quel serait l'impact si ce problème était dans  $P$  ?

Si pour un problème, on a un certificat polynomial pour une réponse positive et négative, on peut seulement dire que le problème appartient à la classe  $NP$ .

Pour l'algorithme des nombres premiers en  $O(\sqrt{n})$ , l'algorithme est sous-linéaire sur la donnée, mais pour calculer la complexité, il faut calculer selon la taille de la donnée. Comme pour l'exercice 33, la taille de la donnée est en  $O(\log(n))$ , donc par le même raisonnement, si on pose  $k = \log(n)$ , l'algorithme est en complexité  $O(\sqrt{2^k})$  (car  $n = 2^k$ ), qui est exponentiel.

Par le même argument que dans le paragraphe précédent, on ne peut pas déduire que ce problème est dans  $P$ . Si ce problème était dans  $P$ , comme il se réduit à  $SAT$ , alors on aurait  $P = NP$ , et les systèmes informatiques sécurisés du monde entier s'effondreraient.

### Exercice 36 - Puissance de calcul

Tous les 4 ans, la puissance des machines est multipliée environ par 8. Vous avez deux algorithmes  $A$  et  $B$  l'un dont le temps d'exécution est proportionnel à  $n^3$  et l'autre dont le temps d'exécution est proportionnel à  $2^n$ . Avec les deux algorithmes vous traitiez un problème de taille  $n = 10$  en 1s, il y a 40 ans. Quelle est la taille des problèmes que vous êtes capables de traiter aujourd'hui avec chacun des deux algorithmes en 1s ?

Commençons par compter le nombre de calcul dans les procédures  $A$  et  $B$ . On pose  $C_A$  le nombre de calculs dans la procédure  $A$  (en  $O(n^3)$ ) et  $C_B$  le nombre de calculs dans la procédure  $B$  (en  $O(2^n)$ ) il y a 40 ans :

$$C_A = 10^3 = 1000$$

$$C_B = 2^{10} = 1024 \simeq 1000$$

La puissance des machines étant multipliée par 8 tous les 4 ans, en 40 ans, la puissance des machines à donc été multipliée par  $8^{10}$ . On remarque que :

$$\begin{aligned} 8^{10} &= (2^3)^{10} \\ &= (2^{10})^3 \\ &\simeq (10^3)^3 \\ &= 10^9 \end{aligned}$$

Il suffit maintenant de multiplier  $C_A$  et  $C_B$  par ce coefficient et d'inverser les résultats trouvés par la complexité de la fonction pour trouver le nombre de données traitées en 1 seconde par les algorithmes  $A$  et  $B$  de nos jours :

$$\begin{aligned}
n_A^3 &= 10^9 \times 10^3 \\
n_A &= \sqrt[3]{10^9} \times \sqrt[3]{10^3} \\
n_A &= 10^3 \times 10 \\
n_A &= 10^4
\end{aligned}$$

L'algorithme  $A$  traite 1000 fois plus de données qu'il y a 40 ans, il en traite 10000 au lieu de 10 en une seconde.

$$\begin{aligned}
2_B^n &= 10^9 \times 2^{10} \\
n_B &= \log(10^9 \times 2^{10}) \\
n_B &= \log(10^9) + \log(2^{10}) \\
n_B &= 9\log(10) + 10 \\
n_B &\simeq 40
\end{aligned}$$

L'algorithme  $B$  traite une trentaine de données en plus qu'il y a 40 ans (40 au lieu de 10 en une seconde).

## 2.2 Autour des classes $\mathcal{P}$ et $\mathcal{NP}$

### Exercice 37 - 2-SATISFAISABILITÉ

1. Montrer en calculant le nombre de clauses créées et le nombre de variables ajoutées que la réduction de SATISFAISABILITÉ à 3-SATISFAISABILITÉ vue en cours est bien polynomiale.
2. Sachant que 2-SATISFAISABILITÉ peut-être résolu en temps polynomial, appliquer l'algorithme pour les 2 instances suivantes :
  - $\phi_1 = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_4) \wedge (x_1 \vee x_4) \wedge (x_1 \vee x_3) \wedge (x_4 \vee \neg x_2)$
  - $\phi_2 = (x_1 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge (x_1 \vee x_4) \wedge (x_4 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3)$
3. Quel problème d'optimisation pouvons-nous étudier dans le cas où la réponse est négative à l'existence d'une solution pour 2-SATISFAISABILITÉ ?

1. Une instance du problème  $I$  contient  $n$  clauses. Chaque clause  $i$  ( $0 < i \leq n$ ) contient un nombre fini  $k_i$  de littéraux. On pose  $r = \sum_{i=1}^n k_i$ . On veut montrer que la réduction vue en cours est polynomiale. On rappelle la réduction vue en cours :

Considérons une clause  $C_i = \{l_{i_1}, l_{i_2}, \dots, l_{i_k}\}$  de  $\phi$  :

- Si  $k_i = 1$ , alors nous introduisons deux nouvelles variables  $y_{i_1}$  et  $y_{i_2}$  et construisons la formule  $\phi'_i = (l_{i_1} \vee y_{i_1} \vee y_{i_2}) \wedge (l_{i_1} \vee y_{i_1} \vee \neg y_{i_2}) \wedge (l_{i_1} \vee \neg y_{i_1} \vee y_{i_2}) \wedge (l_{i_1} \vee \neg y_{i_1} \vee \neg y_{i_2})$
- Si  $k_i = 2$ , alors nous introduisons une nouvelle variable  $y_{i_1}$  et construisons la formule  $\phi'_i = (l_{i_1} \vee l_{i_2} \vee y_{i_1}) \wedge (l_{i_1} \vee l_{i_2} \vee \neg y_{i_1})$
- Si  $k_i = 3$ , alors la clause reste inchangée
- Si  $k > 3$ , nous introduisons  $k - 3$  nouvelles variables  $y_{i_1}, y_{i_2}, \dots, y_{i_{k-3}}$  et construisons alors la formule  $\phi'_i = (l_{i_1} \vee l_{i_2} \vee y_{i_1}) \wedge_{1 \leq j \leq k-4} (\neg y_{i_j} \vee l_{j+2} \vee y_{i_{j+1}}) \wedge (\neg y_{i_{k-3}} \vee l_{i_{k-1}} \vee l_{i_k})$ . La formule  $\phi'_i$  est la conjonction des formules obtenues comme décrit ci-dessus.

Ainsi, il y a 4 cas à considérer :

- Soit  $k_i = 1$ , et dans ce cas, deux nouvelles variables sont générées, et trois nouvelles clause sont créées (tandis que l'ancienne est transformée)  $\Rightarrow k_i \rightarrow 12$ .
- Soit  $k_i = 2$ , et dans ce cas, une nouvelle variable est générée, et une nouvelle clause est créée (tandis que l'ancienne est transformée)  $\Rightarrow k_i \rightarrow 6$ .
- Soit  $k_i = 3$ , et dans ce cas, aucune nouvelle variable ni aucune nouvelle clause ne s'ajoute à la liste.
- Soit  $k_i > 3$ , et dans ce cas,  $k_i - 3$  nouvelles variables sont créées et  $k_i - 3$  clauses sont ajoutées  $\Rightarrow k_i \rightarrow 3(k_i - 2)$ .

Dans le pire des cas, le nombre de clause généré est linéaire (entre  $\times 3$  et  $\times 12$ ). La réduction est donc bien polynomiale.

2. L'algorithme polynomial de 2-SATISFAISABILITÉ est un algorithme qui transforme les clauses en implication logique, afin de les placer sur un graphe orienté, une implication logique représentant une arête, et chaque variable propositionnelle étant un sommet.

En effet, on remarque qu'en logique, la clause  $(a \vee b)$  est équivalente à  $\neg a \Rightarrow b$ , ou encore à  $\neg b \Rightarrow a$  (par sémantique, ou bien par contraposée).

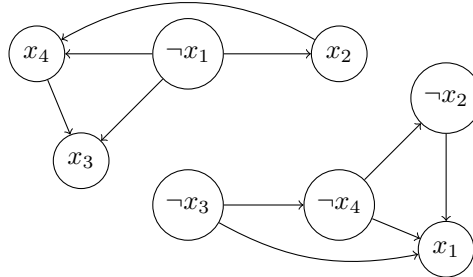
Le but de l'algorithme est d'abord de trouver des contradictions : si dans une composante fortement connexe du graphe se trouvent  $a$  et  $\neg a$ , alors on a une formule du style  $a \Leftrightarrow \neg a$ , ce qui est absurde.

Ensuite, une valeur de vérité est assignée à chaque sommet ( $\top$  pour les sommets de polarité positive, et  $\perp$  pour les sommets de polarité négative) tant qu'il n'y a pas d'affectation pour tous les sommets.

En suivant cet algorithme sur  $\phi_1$ , la transformation sera :

$$\begin{aligned} \phi_1 = & ((\neg x_1 \Rightarrow x_2) \vee (\neg x_2 \Rightarrow x_1)) \wedge ((\neg x_3 \Rightarrow \neg x_4) \vee (x_4 \Rightarrow x_3)) \wedge ((\neg x_1 \Rightarrow x_4) \vee (\neg x_4 \Rightarrow x_1)) \\ & \wedge ((\neg x_1 \Rightarrow x_3) \vee (\neg x_3 \Rightarrow x_1)) \wedge ((\neg x_4 \Rightarrow \neg x_2) \vee (x_2 \Rightarrow x_4)) \end{aligned}$$

Ce qui nous donne le graphe suivant :

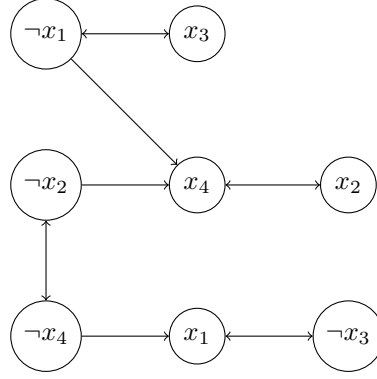


Dans ce graphe, il y a 2 composantes connexes, et chaque composante fortement connexe est composée d'un seul sommet (il n'y a pas de cycle). On peut en déduire que la formule est satisfiable. Une affectation possible est  $x_1 = \top$ ,  $x_2 = \top$ ,  $x_3 = \top$ ,  $x_4 = \top$ .

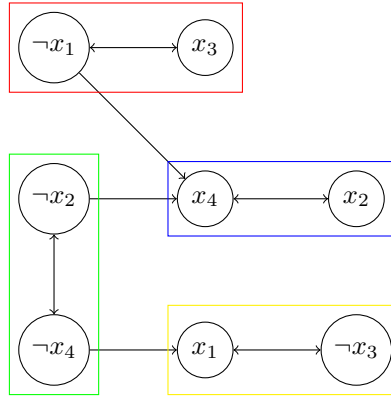
Le cas de  $\phi_2$  est plus intéressant :

$$\begin{aligned} \phi_2 = & ((\neg x_1 \Rightarrow x_3) \vee (\neg x_3 \Rightarrow x_1)) \wedge ((\neg x_2 \Rightarrow x_4) \vee (x_4 \Rightarrow x_2)) \wedge ((\neg x_1 \Rightarrow x_4) \vee (\neg x_4 \Rightarrow x_1)) \\ & \wedge ((\neg x_4 \Rightarrow \neg x_2) \vee (x_2 \Rightarrow x_4)) \wedge ((x_1 \Rightarrow \neg x_3) \vee (x_3 \Rightarrow \neg x_1)) \end{aligned}$$

Qui nous donne le graphe suivant :



Le graphe est connexe, et en particulier, on observe les composantes fortement connexes suivantes :



- La composante rouge contient  $\neg x_1$  et  $x_3$ , il n'y a pas d'équivalence absurde.
- La composante verte contient  $\neg x_2$  et  $\neg x_4$ , il n'y a pas d'équivalence absurde.
- La composante bleue contient  $x_2$  et  $x_4$ , il n'y a pas d'équivalence absurde.
- La composante jaune contient  $x_1$  et  $\neg x_3$ , il n'y a pas d'équivalence absurde.

Il n'y a aucune composante fortement connexe qui contient une équivalence absurde, donc la formule est satisfiable. En particulier, l'affectation suivante rend la formule satisfiable :  $x_1 = \top$ ,  $x_2 = \top$ ,  $x_3 = \perp$ ,  $x_4 = \top$ .

3. Si l'existence d'une solution pour 2-SATISFAISABILITÉ est négative, le problème d'optimisation à étudier est le problème du nombre maximum de clauses satisfiables (ou MAX-SAT).

### Exercice 38 - Algorithme non-déterministe pour le problème de 3-COLORATION

Proposer un algorithme non-déterministe linéaire pour le problème de 3-COLORATION.

On suppose que le graphe  $G = (V, E)$  est coloriable, et que la procédure automatique DEVINECOULEUR( $u$ )

renvoie la bonne couleur pour le sommet  $u$  du graphe.

**Algorithm 6:** Algorithme non-déterministe pour la 3-COLORATION

```

1 begin
2   forall  $u \in V$  do
3      $\text{COULEUR}(u) := \text{DEVINECOULEUR}(u)$ 
4   forall  $(u, v) \in E$  do
5     if  $\text{COULEUR}(u) = \text{COULEUR}(v)$  then
6       return False
7   return True

```

Cet algorithme est bien linéaire : il est de complexité  $O(\max(n, m))$ .

### Exercice 39 - Classification dans $\mathcal{NP}$ ou dans $\mathcal{P}$

Classer les problèmes suivants en fonction de  $\mathcal{P}$  et  $\mathcal{NP}$  :

PROBLÈME P1

**Entrée :**  $G = (V, E)$  un graphe non orienté.

**Question :** Existe-t-il un cycle de longueur égale à  $\left\lfloor \frac{|V|}{2} \right\rfloor$  ?

PROBLÈME P2

**Entrée :**  $G = (V, E)$  un graphe non orienté.

**Question :** Existe-t-il un cycle de longueur égale à 4 ?

PROBLÈME P3

**Entrée :**  $G = (V, E)$  un graphe non orienté.

**Question :** Existe-t-il un chemin simple entre  $u$  et  $v$  de longueur inférieure ou égale à  $k$  ?

PROBLÈME P4

**Entrée :**  $G = (V, E)$  un graphe non orienté et un entier  $k$ .

**Question :** Existe-t-il un arbre couvrant tous les sommets de  $G$  ayant moins de  $k$  feuilles ?

Pour prouver qu'un problème est dans  $\mathcal{NP}$ , il suffit d'exhiber un certificat positif polynomial (c'est à dire qu'on peut vérifier la solution donnée en temps polynomial).

S'il est dans  $\mathcal{P}$ , alors il existe un algorithme connu qui résoud le problème en temps polynomial.

**PROBLÈME P1** La vérification de ce problème peut être fait en temps polynomial. Il suffit de vérifier, grâce au graphe et à la solution, que c'est bien un cycle et que la taille de la solution est  $\left\lfloor \frac{|V|}{2} \right\rfloor$ .

Cependant, ce problème n'est pas dans  $\mathcal{P}$ . En effet, il se réduit au cycle hamiltonien d'un des sous-graphes connexes de  $G$  de  $\left\lfloor \frac{|V|}{2} \right\rfloor$  sommets.

**PROBLÈME P2** Ce problème est dans  $\mathcal{P}$ . On peut exhiber une procédure automatique qui calcule le

résultat attendu en temps polynomial :

**Algorithm 7:** Cycle de longueur égale à 4 dans un graphe non orienté

```

1 begin
2   for  $i = 0$  à  $n$  do
3     for  $j = 0$  à  $n$  do
4       for  $x = 0$  à  $n$  do
5         for  $y = 0$  à  $n$  do
6           if  $\{(i, j), (j, x), (x, y), (y, i)\} \subseteq E$  then
7             return True
8   return False

```

**PROBLÈME P3** Ce problème est dans  $\mathcal{P}$ . En effet, il suffit de lancer un algorithme de plus court chemin (Dijkstra, en  $O(|E| + |V|\log(|V|))$ , s'il n'y a pas d'arête à poids négatifs, Bellman-Ford, en  $O(|V||E|)$ , sinon).

**PROBLÈME P4** Ce problème se réduit à trouver une chaîne hamiltonienne, qui est NP-complet.

## 2.3 Réduction polynomiale

### Exercice 40 - Réduction de Karp : une vision algorithmique

Supposons que le problème  $A$  est NP-complet. Soit  $\alpha$  une instance du problème  $A$ .  $\pi'$  est une réduction polynomiale de  $A$  vers  $B$ , plus formellement,  $O(|\alpha|^{c_1}) \Rightarrow |\pi'(\alpha)| \in O(|\alpha|^{c_1})$ .  $\pi'(\alpha)$  est ainsi une instance du problème  $B$ . De plus,  $\forall \beta$  instance de  $B$ , il y a une solution polynomiale avec l'algorithme  $P_B$ .

Quelle est la complexité de l'algorithme  $A$ . Dédurre la classe du problème  $A$ .

Comme toute instance du problème  $B$  se résout en temps polynomial, et qu'il existe une réduction en temps polynomial pour toute instance du problème  $A$  vers le problème  $B$ ,  $A$  est polynomial. Comme  $\mathcal{P} \neq \mathcal{NP}$ , alors il y a une contradiction :  $A$  n'est pas NP-complet, mais NP-difficile (donc  $B$  est aussi NP-difficile).

### Exercice 41 - Concept de la réduction (suite)

Pour réduire un problème  $A$  à un problème  $B$ , il suffit de montrer que la résolution de  $B$  permet de résoudre  $A$  à condition qu'une solution à  $B$  soit disponible.

Pour illustrer, supposons que  $A$  est le problème suivant :  $A(n)$  = le plus petit nombre premier plus grand que  $n$ , et  $B$  le problème de décision  $B = \{n | n \text{ is prime}\}$ .

1. Donner pour quelques valeurs de  $n$  la valeur  $A(n)$ .
2. Proposer une réduction du problème  $A$  au problème  $B$ .

1.  $A(5) = A(6) = 7$  ou bien  $A(11) = 13$  ou encore  $A(29) = 31$ .
2. Supposons que la procédure automatique  $P_B$  calcule  $B$ . La réduction suivante transforme toute instance du problème  $A$  en problème  $B$  :

```

int P_A(int n) {
    n = n + 1;
    while (!P_B(n)) ++n;
    return n;
}

```

De manière équivalente, supposons que  $P_A$  est la procédure automatique qui calcule  $A$ . La réduction suivante transforme toute instance du problème  $B$  en problème  $A$  :

```
bool P_B(int n) {
    if (n == 0 || n == 1) return false;
    return P_A(n - 1) == n;
}
```

Ces deux problèmes se réduisent l'un à l'autre, ils sont donc de la même classe de complexité. Cependant, on n'est pas sûrs que  $A$  se réduise à  $B$  en temps polynomial : rien ne nous assure qu'il n'y a pas  $2^n$  entiers entre le  $n$  et  $n + 1$ -ème nombre premier.

## Exercice 42 - Réduction

Montrer que les deux problèmes PROBLÈME DU CARRÉ D'UN ENTIER et PROBLÈME DE LA MULTIPLICATION se réduisent l'un à l'autre. L'addition, la soustraction et la division sont des opérations autorisées.

PROBLÈME DE LA MULTIPLICATION

**Entrée :** Soient  $a \in \mathbb{N}$  et  $b \in \mathbb{N}$ .

**Question :** Peut-on multiplier  $a$  et  $b$  ?

PROBLÈME DU CARRÉ D'UN ENTIER

**Entrée :** Soient  $a \in \mathbb{N}$ .

**Question :** Peut-on élever  $a$  au carré ?

On suppose qu'on peut élever au carré un entier. On peut réduire le PROBLÈME DE LA MULTIPLICATION en PROBLÈME DU CARRÉ D'UN ENTIER en remarquant que :

$$(a + b)^2 = a^2 + 2ab + b^2 \Rightarrow ab = \frac{(a + b)^2 - a^2 - b^2}{2}$$

De manière équivalente, on peut réduire le PROBLÈME DU CARRÉ D'UN ENTIER en PROBLÈME DE LA MULTIPLICATION en multipliant le nombre donné par lui-même.

## Exercice 43 - Réduction entre deux problèmes polynomiaux

2-SATISFAISABILITÉ

**Entrée :** Étant donné une formule conjonctive  $\phi$  sur  $n$  variables et  $m$  clauses chacune de taille deux.

**Question :** Existe-t-il une affectation de vérité aux variables qui satisfasse  $\phi$  ?

2-COLORATION

**Entrée :** Soit  $G = (V, E)$  et deux couleurs.

**Question :** Existe-t-il une 2-coloration valide, i.e. une fonction totale  $f : \mathbb{N} \rightarrow \{1, 2\}$  telle que  $f(u) \neq f(v), \forall (u, v) \in E$  ?

1. Montrer qu'il existe une réduction polynomiale entre 2-COLORATION et 2-SATISFAISABILITÉ.
2. Conclure sur la complexité du problème 2-SATISFAISABILITÉ.

Pour connaître la classe de complexité maximum de 2-SATISFAISABILITÉ, sachant que 2-COLORATION est dans  $\mathcal{P}$ , il faut trouver une réduction polynomiale entre 2-SATISFAISABILITÉ et 2-COLORATION pour prouver que 2-SATISFAISABILITÉ est aussi dans  $\mathcal{P}$  :

- 1.