Exercices 36 à 39 (correction)

Exercice 36 (graphes). Proposer un pseudo-code pour chacune des fonctions définies ci-dessous :

(1) La fonction **simple(G)** prend en entrée un graphe G et renvoie le booléen `vrai` si le graphe G est simple, `faux` sinon.

On balaye tous les sommets de G, et pour chaque sommet s on teste si s est voisin de s (si cette situation se produit, par définition G n'est pas simple)

```
fonction simple(G)
   pour tout sommet s de G
      si s est voisin de s
         retourner faux
   retourner vrai
```

(2) La fonction **dmax(G)** prend en entrée un graphe G et renvoie le degré maximal obtenu parmi tous les sommets de G.

On calcule le degré (nombre de voisins) de chaque sommet de G, et on prend la valeur maximale rencontrée.

```
fonction dmax(G)
   max <- 0
   pout tout sommet s de G
      d <- nombre de voisins de s
      si d>max
         max <- d
   retourner max
```

(3) La fonction **régulier(G)** prend en entrée un graphe G et renvoie le booléen `vrai` si le graphe G est régulier, `faux` sinon.

On vérifie que tous les sommets de G ont le même degré (c'est-à-dire le même nombre de voisins). Pour cela, on balaie tous les sommets de G, et on utilise une variable D, initialisée avec la valeur (symbolique) `rien`, qui reçoit le degré du premier sommet rencontré. À partir du moment où D a été mise à jour, on vérifie que chaque degré calculé est bien égal à D.

```
fonction régulier(G)
   D <- rien
   pour tout sommet s de G
      d <- nombre de voisins de s
      si D=rien
         D <- d
      sinon si D différent de d
        retourner faux
   retourner vrai
```

(4) La fonction **complet(G)** prend en entrée un graphe G et renvoie le booléen `vrai` si le graphe G est complet, `faux` sinon.

Nous avons déjà vu cette fonction dans le cours, il suffit de vérifier que chaque sommet est bien connecté à tous les autres. Pour cela, on balaie les sommets de G, et pour chaque sommet s on vérifie que l'ensemble des voisins de s, complété par s lui-même, donne bien l'ensemble de tous les sommets du graphe. Si ce n'est pas le cas, le graphe n'est pas complet et on retourne la valeur `faux`.

```
fonction complet(G)
   S <- ensemble formé par les sommets de G
   pour tout sommet s de G
      si voisins(s) U {s} est différent de S
         retourner faux
   retourner vrai
```

(5) La fonction **symétrique(G)** prend en entrée un graphe orienté G et renvoie le booléen `vrai` si le graphe G est symétrique (au sens où l'arête $A \to B$ existe ssi l'arête $B \to A$ existe), `faux` sinon.

On balaie toutes les arêtes du graphe (en balayant tous les sommets et tous les voisins de chaque sommet), et on vérifie que l'arête inverse est bien présente. Si ce n'est pas le cas pour une arête, on retourne la valeur `faux`.

```
fonction symétrique(G)
   pour tout sommet s de G
      pour tout voisin t de s
         si s n'est pas voisin de t
            retourner faux
   retourner vrai
```

(6) La fonction **adjacence(G)** prend en entrée un graphe orienté G et renvoie sa matrice d'adjacence.

On commence par lire dans une variable n la taille du graphe (nombre de sommets), et on construit, coefficient par coefficient, la matrice d'adjacence du graphe, qui est de taille $n \times n$, en l'initialisant à 0 partout, et en plaçant la valeur 1 à chaque coefficient (i, j) pour lequel l'arête $i \to j$ est présente dans le graphe (définition de la matrice d'adjacence du graphe).

```
fonction adjacence(G)
   // on suppose que les sommets de G sont numérotés à partir de 1
   n <- nombre de sommets de G
   A <- matrice nulle de taille n x n
   pour i=1,2,...n
      pour j=1,2,...n
         si j est voisin de i
            A(i,j) <- 1
   retourner A
```

Exercice 37 (graphes). Soit G un graphe orienté de matrice d'adjacence A (on suppose que les sommets sont numérotés de 1 à n). Trouver, pour chacune des propritétés ci-dessous, une caractérisation de la propriété au moyen de A (par exemple: G est simple ssi tous les coefficients diagonaux de A sont nuls).

(1) Le graphe G est régulier

Un graphe est régulier si tous ses sommets ont le même degré (c'est-à-dire le même nombre de voisins). Or, le degré du sommet i est simplement donné par la somme des coefficients de la ligne i de la matrice d'adjacence. La condition "G régulier" se réécrit donc

$$\forall i \in \{1, 2, \dots, n\}, \quad \sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} a_{1j}$$

(2) Le graphe G est complet (on suppose que G est simple)

Comme G est simple, il est complet si et seulement si le degré de chaque sommet est $n-1$. Par conséquent, la condition "G complet" se réécrit

$$\forall i \in \{1, 2, \dots, n\}, \quad \sum_{j=1}^{n} a_{ij} = n - 1$$

(3) Il existe un chemin dans G reliant le sommet 1 au sommet n

Si $B = A^k$, d'après le cours b_{1n} compte le nombre de chemins de longueur k allant du sommet 1 au sommet n. Autrement dit, $b_{1n} > 0$ si et seulement si il existe un chemin **de longueur k** allant du sommet 1 au sommet n. Or s'il existe un chemin allant du sommet 1 au sommet n, on peut, quitte à supprimer des boucles inutiles, le réduire à un chemin qui ne repasse pas deux fois par le même sommet, donc de longueur au plus $n-1$. On en déduit qu'il existe un chemin allant du sommet 1 au sommet n si et seulement si le coefficient $(1, n)$ de A^k est strictement positif pour un certain $k \in \{1, 2, \dots, n-1\}$. Cette condition peut s'écrire plus simplement

$$b_{1n} > 0,$$

en posant

$$B = A + A^2 + A^3 + ... + A^{n-1}.$$

(4) Le graphe G est connexe

Le graphe G est connexe si et seulement si il existe un chemin allant du sommet i au sommet j pour tout couple (i, j) tel que $1 \leqslant i, j \leqslant n$. D'après la question précédente, cette condition s'écrit

$$\forall i \in \{1, 2, \ldots, n\},\ \forall j \in \{1, 2, \ldots, n\}, \quad b_{ij} > 0,$$

en posant

$$B = A + A^2 + A^3 + ... + A^{n-1}$$

(dit autrement: tous les coefficients de la matrice $A + A^2 + A^3 + ... + A^{n-1}$ doivent être strictement positifs).

(5) Pour tout couple (x, y) de sommets de G, la distance de x à y est au plus égale à d

Il s'agit d'une variante de la question précédente, où l'on limite la longueur des chemins à d. La condition s'écrit donc toujours

$$\forall i \in \{1, 2, \ldots, n\},\ \forall j \in \{1, 2, \ldots, n\}, \quad b_{ij} > 0,$$

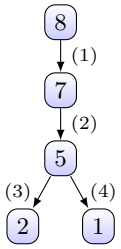
mais en posant cette fois

$$B = A + A^2 + A^3 + ... + A^{d}.$$

Exercice 38 (jeu à deux joueurs). Alice et Bernard décident de jouer au jeu suivant. À tour de rôle, chaque joueur reçoit un entier n, choisit un entier k selon certaines règles, et donne le nombre $n - k$ à l'autre joueur, qui continue de même. Le premier joueur qui ne peut pas jouer a perdu. Les règles imposés pour le choix de k sont les suivantes :

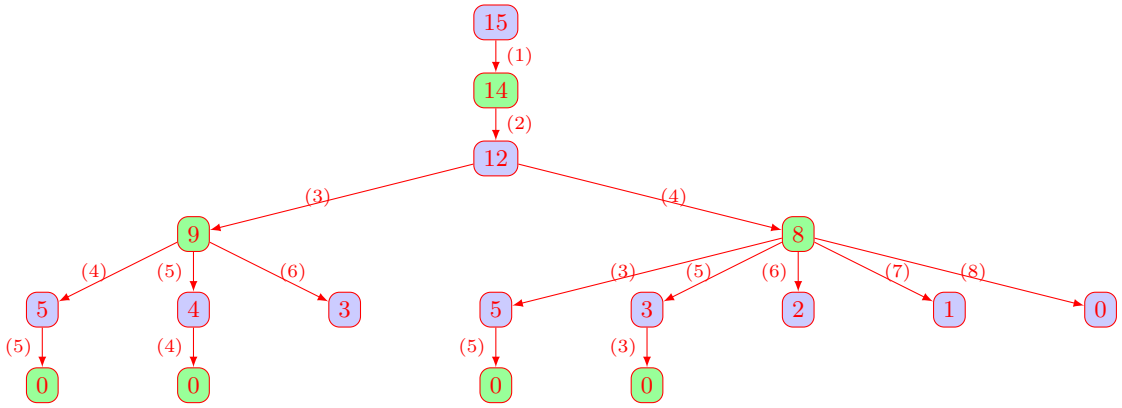
- le nombre k est compris entre 1 et le double du nombre choisi par l'autre joueur au tour précédent (pour Alice, qui joue en premier, le choix $k = 1$ est imposé);
- le nombre k n'a jamais été joué depuis le début de la partie;
- le résultat $n - k$ doit être positif ou nul.

On convient que Alice commence systématiquement le jeu, et que sa valeur de départ (n) est fixée d'un commun accord avec Bernard. Par exemple, si Alice commence le jeu avec $n = 8$, alors elle doit soustraire $k = 1$ (choix forcé au premier coup); Bernard reçoit donc l'entier 7, et est forcé de jouer $k = 2$ (car $k = 1$ a déjà été choisi); Alice reçoit alors l'entier 5, et peut jouer $k = 3$ ou $k = 4$. Dans les deux cas, Bernard reçoit un entier inférieur ou égal à 2, et ne peut donc plus jouer, donc Alice gagne. On peut résumer cette analyse pour $n = 8$ avec l'arbre ci-dessous, qui représente toutes les parties possibles, au nombre de 2 ici (l'étiquette de chaque nœud indique la valeur de n, et l'étiquette sur chaque arête indique le choix de k. Les feuilles correspondent aux situations perdantes (le joueur ne peut plus jouer).



(1) Dessiner l'arbre correspondant lorsque la valeur de départ est $n = 15$, puis lorsque la valeur de départ est $n = 16$ (on rapelle que c'est toujours Alice qui commence, avec cette valeur de n). Indiquer comment Bernard peut forcer le gain pour $n = 15$ (bien que ce ne soit pas lui qui commence), et comment Alice peut forcer le gain pour $n = 16$.

Pour $n = 15$, l'arbre de toutes les parties possibles est le suivant :

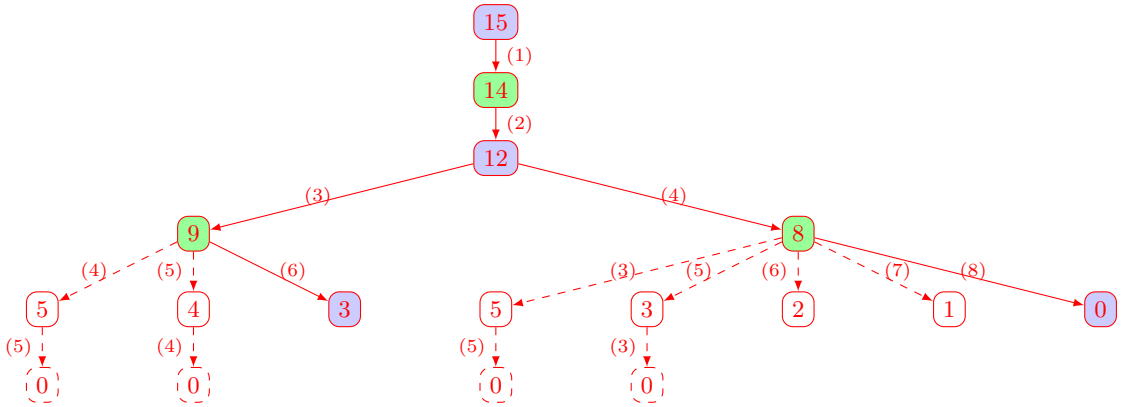


Les nœuds sur fond bleu correspondent aux moments où Alice doit jouer, les nœuds sur fond vert aux moments où Bernard doit jouer.

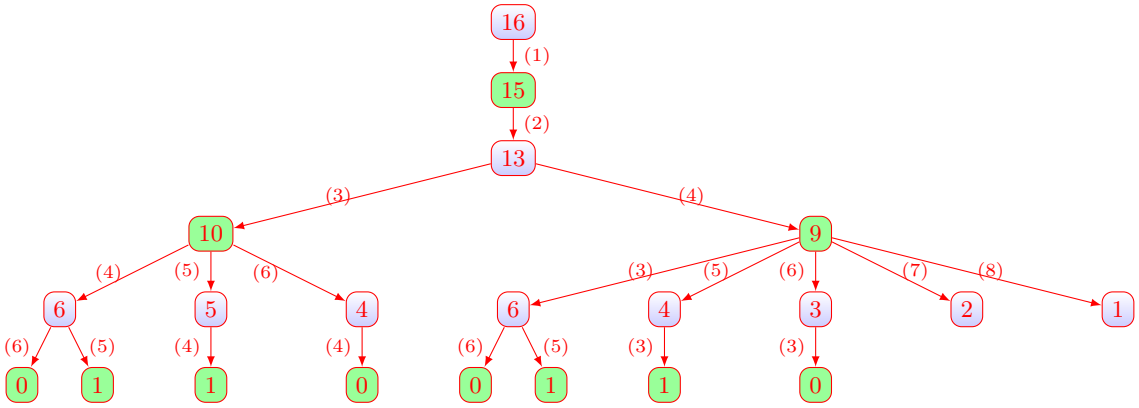
Bernard dispose d'une stratégie gagnante pour la raison suivante :

- Au premier coup, Alice est forcée de jouer $1 \to 15 - 1 = 14$
- Bernard joue 2 (forcé) $\to 14 - 2 = 12$
- Alice a alors le choix entre 3 et 4, mais :
  - si elle joue 3, $12 - 3 = 9$ et Bernard peut alors jouer $6 \to 9 - 6 = 3$ et Alice perd (plus de coup possible)
  - si elle joue 4, $12 - 4 = 8$ et Bernard peut alors jouer $8 \to 8 - 8 = 0$ et Alice perd également.

Si l'on se réfère à l'arbre des parties possibles, l'existence d'une stratégie gagnante pour Bernard se résume au fait qu'en ne conservant qu'une seul branche descendante pour chaque nœud vert (la sélection de cette branche correspondant au choix d'un coup par Bernard), on peut aboutir à un arbre dont toutes les feuiles sont bleues (c'est-à-dire correspondant à des situations perdantes pour Alice). C'est effectivement le cas, comme le montre le schéma ci-dessous, dans lequel on a simplement indiqué en blanc et en pointillés les parties de l'arbre qui n'étaient pas atteignables compte tenu des choix stratégiques de Bernard.



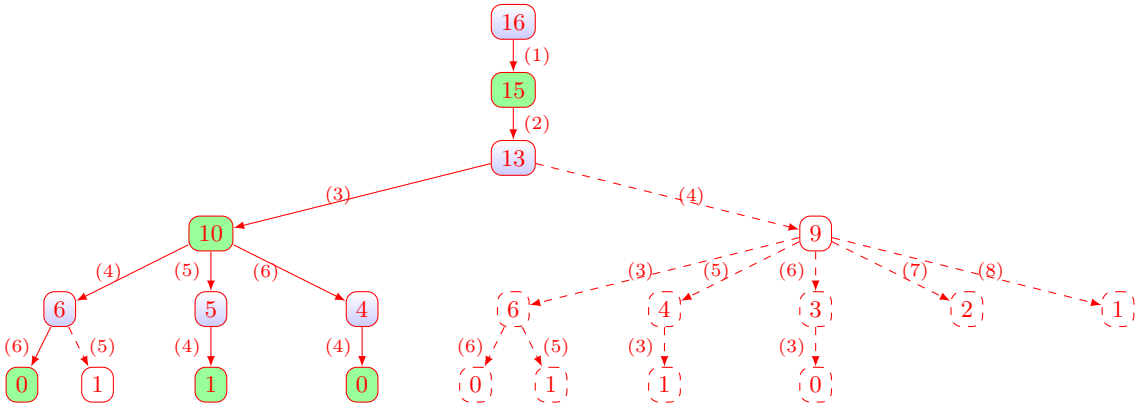
Pour $n = 16$, l'arbre de toutes les parties possibles est le suivant :



Cette fois-ci, c'est Alice qui dispose d'une stratégie gagnante pour la raison suivante :

- Alice joue 1 (forcé) $\rightarrow 16-1=15$
- Bernard joue 2 (forcé) $\rightarrow 15-2=13$
- Alice choisit de jouer 3 $\rightarrow 13-3=10$
- Bernard a alors le choix entre 4, 5 ou 6
  - s'il joue 4, $10-4=6$ et Alice gagne en jouant 6 $(\rightarrow 6-6=0)$
  - s'il joue 5, $10-5=5$ et Alice gagne en jouant 4 $(\rightarrow 5-4=1)$
  - s'il joue 6, $10-6=4$ et Alice gagne en jouant 4 $(\rightarrow 4-4=0)$

Là encore, on peut résumer cette stratégie en "effaçant" les parties de l'arbre qui ne sont pas atteintes en raison des choix stratégiques d'Alice. Comme on le voit, le sous-arbre correspondant n'a que des feuilles vertes, donc correspond bien à une victoire systématique d'Alice.



(2) Soit **arbre**(n, E, m) la fonction qui prend en entrée un entier $n \geqslant 1$, un ensemble d'entiers E et un entier m, et renvoie l'arbre associé à toutes les fins de parties possibles à partir de la valeur n, de l'ensemble de coup joués E (éventuellement vide) et de la contrainte $k \leqslant m$ pour le coup à venir.

Montrer que l'arbre renvoyé par **arbre**(n, E, m) a pour racine un sommet d'étiquette n et dont les enfants sont les sous-arbres retournés par **arbre**$(n-k, E \cup \{k\}, 2k)$ pour $k \in \mathbb{N}, 1 \leqslant k \leqslant n, k \leqslant m, k \notin E$.

L'entier n représente le total disponible, E l'ensemble des entiers k déjà joués aux coups précédents, et m la borne maximale sur k due au coup précédent (ou imposée au départ). La relation proposée décrit alors simplement les règles du jeu: le joueur suivant peut passer de n à $n-k$ en enlevant k, à condition que $1 \leqslant k \leqslant n$, $k \leqslant m$ et $k \notin E$. Le sous-arbre enfant correspondant est alors obtenu avec **arbre**$(n-k, E \cup \{k\}, 2k)$

(3) En déduire une implémentation Python de la fonction précédente sous la forme d'une fonction définie par l'en-tête

```
def arbre(n,E=set(),m=1):
```

On pourra utiliser la fonction `creer_arbre()` ci-dessous pour la création des arbres :

```
def creer_arbre(e,L=[]):
    """
    Retourne un arbre dont la racine a pour étiquette e,
    et pour enfants les éléments de la liste L (éventuellement vide)
    signature: étiquette x liste d'arbres -> arbre
    """
    return [e]+L
```

On implémente la relation récursive de la question précédente : pour créer l'arbre que doit renvoyer `arbre(n,E,m)`, on construit la liste `L` de tous les arbres retournés par `arbre(n-k,E∪{k},2k)` pour $k \in \mathbb{N}, 1 \leqslant k \leqslant n, k \leqslant m, k \notin E$, et l'on retourne ensuite, grâce à l'appel `creer_arbre(n,L)`, un arbre dont la racine a pour étiquette `n` et pour enfants les éléments de `L`.

```
def arbre(n,E=set(),m=1):
    L = []
    for k in range(1,m+1):
        if k<=n and k not in E:
            L.append(arbre(n-k,E|{k},2*k))
    return creer_arbre(n,L)
```

Compte-tenu des valeurs par défaut de `E` et `m`, comment interprète-t-on le résultat retourné par l'appel `arbre(n)` ?

Les valeurs par défaut $E = \{\}$ et $m = 1$ correspondent au contraintes du premier coup, donc `arbre(n)` renvoie l'arbre de toutes les parties qui commencent avec la valeur `n`.

(4) Vérifier les résultats obtenus à la question 1 en examinant les résultats retournés par `arbre(15)` `arbre(16)`.

```
>>> arbre(15)
[15, [14, [12, [9, [5, [0]], [4, [0]], [3]], [8, [5, [0]], [3, [0]], [2], [1], [0]]]]]
>>> arbre(16)
[16, [15, [13, [10, [6, [1], [0]], [5, [1]], [4, [0]]], [9, [6, [1], [0]], [4, [1]],
[3, [0]], [2], [1]]]]]
```

Ces listes correspondent bien aux arbres de la question 1

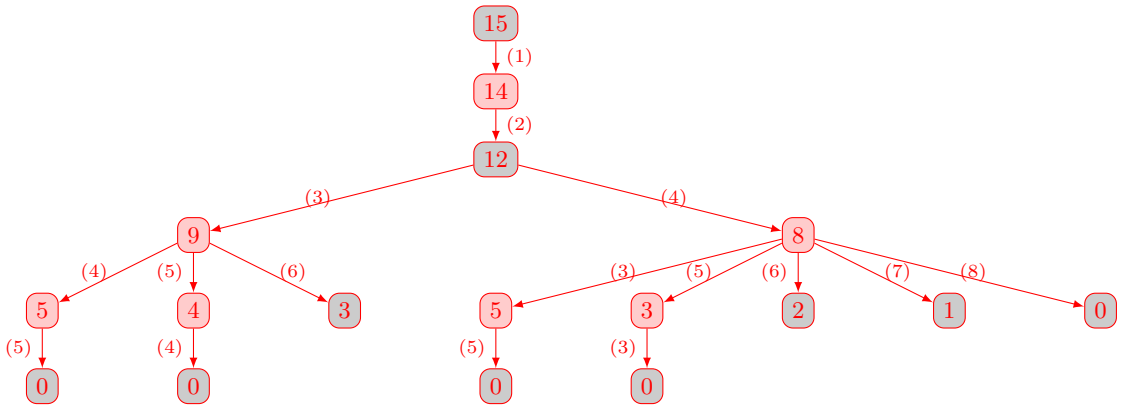
(5) Soit A l'arbre des parties possibles pour un entier n donné. On définit une fonction booléenne g sur les nœuds de A avec la règle récursive suivante :

- si $g(y) =$ faux pour au moins un enfant y de x, alors $g(x) =$ `vrai`;
- dans tous les autres cas (et en particulier lorsque x est une feuille de A), $g(x) =$ `faux`.

Calculer la fonction g sur tous les nœuds des arbres de la question 1. Comment s'interprète la valeur que prend g pour la racine de A ?

Reprenons les arbres de la questions 1, et colorions les nœuds en fonction de la valeur de g: rouge si g est `vrai`, gris si g est `faux`. Pour colorier les nœuds, on applique la définition de g: on commence par colorier en gris (valeur `faux`) toutes les feuilles de l'arbre, puis on remonte progressivement vers la racine en prenant tour à tour chaque nœud dont toutes les feuilles sont coloriées, et en le coloriant :

- en rouge si au moins l'un de ses enfants est gris;
- en gris sinon.

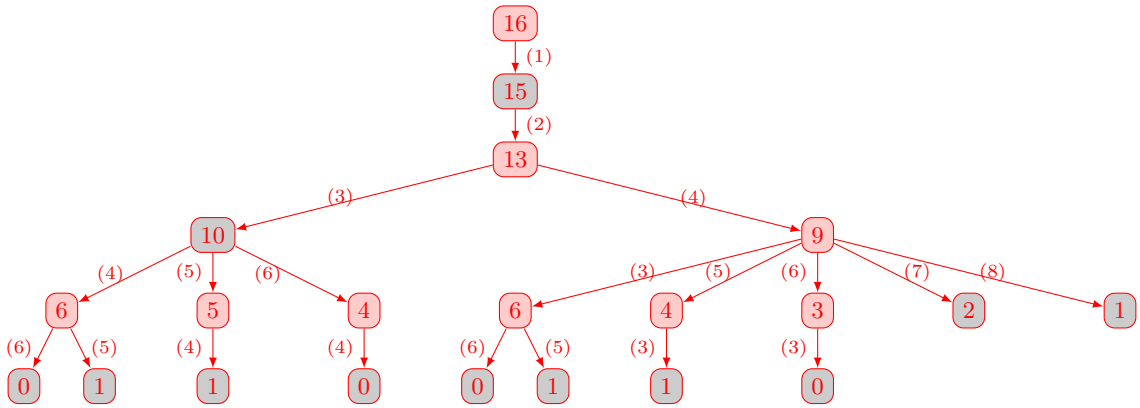


La valeur de g en un nœud de l'arbre permet de décider, pour toutes les parties qui arrivent à ce nœud, si le joueur dont c'est le tour de jouer à une stratégie gagnante ou non. En effet:

- si le nœud est une feuille, le joueur courant a perdu;
- si g prend la valeur `faux` pour au moins l'un des enfants du nœud considéré, cela signifie que le choix, pour le joueur courant, du coup correspondant à cet enfant, amène son adversaire vers une situation perdante (c'est-à-dire pour lequel il n'a pas de stratégie gagnante). Autrement dit, ce choix donne au joueur courant le début d'une stratégie gagnante.

Pour $n = 15$ (arbre ci-dessus), la racine est grise (g `faux`) donc Alice n'a pas de stratégie gagnante et par conséquent Bernard a une stratégie gagnante (qui consiste à jouer, à chaque étape, un coup qui amène Alice sur un nœud gris).

Pour $n = 16$, on obtient le coloriage de l'arbre des parties possibles suivant :



La racine est rouge (g `vrai`) donc Alice a une stratégie gagnante (qui consiste à jouer, à chaque étape, un coup qui amène Bernard sur un nœud gris).

(6) Écrire une fonction Python `g(A)` qui prend en entrée un arbre A, et renvoie la valeur de la fonction g pour la racine de A. Pour le parcours de l'arbre A, on utilisera la fonction `enfants()` ci-dessous, qui retourne la liste des enfants de la racine d'un arbre (voir cours).

```
def enfants(A):
    "retourne la liste des enfants de la racine de l'arbre A"
    return A[1:]
```

l'implémentation récursive de la fonction g est très simple : si l'arbre est une feuille, on retourne `False`, sinon on calcule récursivement g sur tous les enfants de la racine et l'on retourne `True` si et seulement si g prend la valeur `False` pour au moins l'un des enfants. On peut pour cela utiliser la fonction `any` vue au chapitre 2 (p. 59).

```
def g(A):
    if len(enfants(A))==0: return False
    return any(not g(x) for x in enfants(A))
```

Vérifier les résultats de la question 1 en calculant `g(arbre(15))` et `g(arbre(16))`.

```
>>> g(arbre(15))
False
>>> g(arbre(16))
True
```

L'appel `g(arbre(n))` permet de savoir, pour une valeur de n donnée, si Alice a une stratégie gagnante (réponse `True`) ou si, au contraire, c'est Bernard qui dispose d'une telle stratégie (réponse `False`). On obtient bien les valeurs attendues : `False` pour $n = 15$ (stratégie gagnante pour Bernard) et `True` pour $n = 16$ (stratégie gagnante pour Alice).

(7) Déterminer tous les entiers $n \leqslant 60$ pour lesquels Alice a une stratégie gagnante.

On affiche la liste des entiers $n \leqslant 60$ pour lesquels `g(arbre(n))` prend la valeur `True` :

```
>>> print([n for n in range(1,61) if g(arbre(n))])
[1, 2, 6, 7, 8, 9, 16, 17, 18, 19, 20, 21, 22, 31, 32, 33, 34, 35, 36, 37, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58]
```

Exercice 39 (mesure de volumes).

On dispose de 2 récipients, un récipient A de 3 litres et un récipient B de 7 litres, tous deux initialement vides. Si l'on ne dispose d'aucun autre récipient ni moyen de mesurer des volumes, on peut néanmoins mesurer un volume d'un litre en procédant comme suit:

- remplir B (complètement)
- vider B dans A (jusqu'à ce que A soit plein)
- jeter le contenu de A
- vider B dans A (jusqu'à ce que A soit plein)

$\rightarrow$ il reste alors 1 litre dans B

Si l'on note sous la forme d'un couple (x, y) une configuration donnée (A contient x litres et B contient y litres), alors l'algorithme précédent peut s'écrire sous la forme d'une suite de configurations :

$$(0,0) - (0,7) - (3,4) - (0,4) - (3,1)$$

(1) Proposer un schéma similaire pour mesurer un volume de 2 litres.

En démarrant comme pour un volume de 1 litre (exemple de l'exercice):

(0,0) - (0,7) - (3,4) - (0,4) - (3,1) - (0,1) - (1,0) - (1,7) - (3,5) - (0,5) - (3,2)

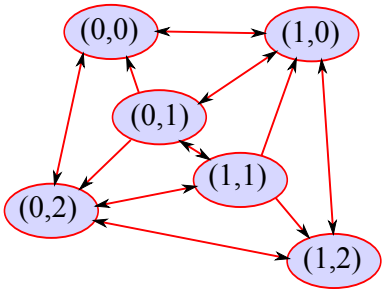
ou plus simplement

(0,0) - (3,0) - (0,3) - (3,3) - (0,6) - (3,6) - (2,7)

(2) Étant donnés deux récipients A et B de contenances respectives a et b litres (avec a et b entiers et $1 \leqslant a \leqslant b$), on souhaite déterminer le graphe (simple orienté) des configurations accessibles à partir de la configuration vide. Un sommet du graphe représente une configuration, et chaque arête correspond à l'une des manipulations élémentaires suivantes :

- remplir (complètement) l'un des récipients;
- jeter le contenu d'un des récipients;
- vider (au maximum) l'un des récipients dans l'autre (à la fin, le premier est vide ou le deuxième est plein).

Par exemple, si $a = 1$ et $b = 2$, on obtient le graphe orienté ci-dessous :



Proposer le pseudo-code d'une fonction **configurations(a,b)** qui prend en entrée les deux entiers précédents a et b, et renvoie le graphe des configurations accessibles à partir de la configuration $(0,0)$. On pourra construire ce graphe avec un parcours en largeur à partir de la configuration $(0,0)$, c'est-à-dire en traitant dans une boucle les sommets d'un ensemble N (initialisé avec $\{(0,0)\}$), et en construisant dans la boucle la version de l'ensemble N destinée à la boucle suivante, formée à partir des sommets rencontrés mais pas encore présents dans G.

En suivant le principe décrit dans l'algorithme (parcours en largeur vu en cours à partie du sommet $(0,0)$, on aboutit à la construction suivante :

```
fonction configurations(a,b)
    // retourne le graphe des configurations accessibles depuis (0,0)
    G <- graphe vide (dictionnaire)
    N <- {(0,0)}
    tant que N est non vide
        M <- ensemble vide
        pour tout élément (x,y) de N
            fabriquer C, l'ensemble de toutes les configurations valides
              que l'on peut obtenir à partir de (x,y)
            ajouter à M tous les éléments de C qui ne sont pas des clés de G
            ajouter à G la clé (x,y) associée à la valeur C (ensemble)
        N <- M
    retourner G
```

(3) Implémenter la fonction précédente en Python (on représentera la graphe sous la forme d'un dictionnaire, comme nous l'avons vu en cours), et vérifier qu'elle retourne le bon résultat pour $a = 1$ et $b = 2$.

Par rapport au pseudo-code de la question précédente, il faut détailler la construction de l'ensemble `C`, qui s'obtient en considérant toutes les actions élémentaires possibles à partir d'une configuration (x, y): vider B dans A (si A n'est pas plein et B n'est pas vide), vider A dans B (si B n'est pas plein et A n'est pas vide), jeter le contenu de A (si A n'est pas vide), jeter le contenu de B (si B n'est pas vide), remplir A (si A n'est pas plein), remplir B (si B n'est pas plein).

```
def configurations(a,b):
    "retourne le graphe des configurations accessibles depuis (0,0)"
    G = dict()
    N = {(0,0)}
    while N: # tant qu'il reste des voisins à explorer
        M = set() # liste des voisins qui seront explorés à la prochaine itération
        for x,y in N:
            C = set() # voisins de la configuration (x,y)
            if x<a and y>0: # vide B dans A
                d = min([y,a-x])
                C.add((x+d,y-d))
            if x>0 and y<b: # vide A dans B
                d = min([x,b-y])
                C.add((x-d,y+d))
            if x>0: # vide A
                C.add((0,y))
            if y>0: # vide B
                C.add((x,0))
            if x<a: # remplit A
                C.add((a,y))
            if y<b: # remplit B
                C.add((x,b))
            # on complète les sommets à traiter à la prochaine itération
            M |= {z for z in C if z not in G}
            # on ajoute le sommet (x,y) avec référence à tous ses voisins
            G[(x,y)] = C
        N = M # pour l'itération suivante
    return G

>>> configurations(1,2)
{(0, 1): {(1, 0), (0, 0), (1, 1), (0, 2)},
 (1, 2): {(1, 0), (0, 2)},
 (0, 0): {(1, 0), (0, 2)},
 (1, 0): {(0, 1), (1, 2), (0, 0)},
 (0, 2): {(1, 2), (0, 0), (1, 1)},
 (1, 1): {(0, 1), (1, 2), (1, 0), (0, 2)}}
```

On peut vérifier que ce dictionnaire décrit bien le graphe donné dans l'énoncé de la question 2.

(4) Implémenter une fonction Python `volumes(a,b)`, qui renvoie l'ensemble des volumes entiers accessibles pour deux récipients de tailles respectives a et b. Cette fonction pourra faire appel à la fonction `configurations()` implémentée à la question 3. Examiner (après tri) le résultat produit par `volumes(a,b)` pour plusieurs valeurs de a et b, et établir une conjecture empirique sur l'expression générale de `volumes(a,b)`.

L'ensemble retourné par l'appel `volumes(a,b)` se construit simplement en considérant toutes les valeurs de x et de y obtenues pour les clés (x, y) du dictionnaire retourné par l'appel `configurations(a,b)`.

```
def volumes(a,b):
    G = configurations(a,b)
    return {x for x,y in G}|{y for x,y in G}
```

On peut alors afficher le résultat donné par `volumes(a,b)` pour tous les couples (a, b) tels que $2 \leqslant a < b \leqslant 19$) :

```
for a in range(2,10):
    for b in range(a+1,20):
        print(a,b,sorted(volumes(a,b)))
```

```
2 3 [0, 1, 2, 3]
2 4 [0, 2, 4]
2 5 [0, 1, 2, 3, 4, 5]
2 6 [0, 2, 4, 6]
2 7 [0, 1, 2, 3, 4, 5, 6, 7]
2 8 [0, 2, 4, 6, 8]
2 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 10 [0, 2, 4, 6, 8, 10]
2 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
2 12 [0, 2, 4, 6, 8, 10, 12]
2 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
2 14 [0, 2, 4, 6, 8, 10, 12, 14]
2 15 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
2 16 [0, 2, 4, 6, 8, 10, 12, 14, 16]
2 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
2 18 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
2 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
3 4 [0, 1, 2, 3, 4]
3 5 [0, 1, 2, 3, 4, 5]
3 6 [0, 3, 6]
3 7 [0, 1, 2, 3, 4, 5, 6, 7]
3 8 [0, 1, 2, 3, 4, 5, 6, 7, 8]
3 9 [0, 3, 6, 9]
3 10 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3 12 [0, 3, 6, 9, 12]
3 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
3 14 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
3 15 [0, 3, 6, 9, 12, 15]
3 16 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
3 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
3 18 [0, 3, 6, 9, 12, 15, 18]
3 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
4 5 [0, 1, 2, 3, 4, 5]
4 6 [0, 2, 4, 6]
4 7 [0, 1, 2, 3, 4, 5, 6, 7]
4 8 [0, 4, 8]
4 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4 10 [0, 2, 4, 6, 8, 10]
4 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
4 12 [0, 4, 8, 12]
4 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
4 14 [0, 2, 4, 6, 8, 10, 12, 14]
4 15 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
4 16 [0, 4, 8, 12, 16]
```

```
4 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
4 18 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
4 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
5 6 [0, 1, 2, 3, 4, 5, 6]
5 7 [0, 1, 2, 3, 4, 5, 6, 7]
5 8 [0, 1, 2, 3, 4, 5, 6, 7, 8]
5 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 10 [0, 5, 10]
5 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
5 12 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
5 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
5 14 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
5 15 [0, 5, 10, 15]
5 16 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
5 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
5 18 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
5 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
6 7 [0, 1, 2, 3, 4, 5, 6, 7]
6 8 [0, 2, 4, 6, 8]
6 9 [0, 3, 6, 9]
6 10 [0, 2, 4, 6, 8, 10]
6 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
6 12 [0, 6, 12]
6 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
6 14 [0, 2, 4, 6, 8, 10, 12, 14]
6 15 [0, 3, 6, 9, 12, 15]
6 16 [0, 2, 4, 6, 8, 10, 12, 14, 16]
6 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
6 18 [0, 6, 12, 18]
6 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
7 8 [0, 1, 2, 3, 4, 5, 6, 7, 8]
7 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
7 10 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
7 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
7 12 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
7 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
7 14 [0, 7, 14]
7 15 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
7 16 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
7 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
7 18 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
7 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
8 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
8 10 [0, 2, 4, 6, 8, 10]
8 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
8 12 [0, 4, 8, 12]
8 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
8 14 [0, 2, 4, 6, 8, 10, 12, 14]
8 15 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
8 16 [0, 8, 16]
8 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
8 18 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
8 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
9 10 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
9 11 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
9 12 [0, 3, 6, 9, 12]
9 13 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
9 14 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
9 15 [0, 3, 6, 9, 12, 15]
9 16 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
9 17 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
9 18 [0, 9, 18]
9 19 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

On constate que pour la plupart des couples (a, b) avec $a < b$, tous les volumes de 0 à b sont réalisables. Les exceptions surviennent lorsque a et b sont divisibles par un même entier au moins égal à deux. Plus

précisément, si l'on note d le pgcd de a et b, on remarque que dans tous les cas les volumes réalisables sont exactement les multiples de d compris entre 0 et b.

Le fait que l'on ne puisse pas obtenir autre chose que des multiples de d vient simplement du fait que tous les volumes produits par les opérations élémentaires restent des multiples de d.

Le fait qu'on puisse obtenir effectivement tous les multiples est moins évident (mais on ne demande pas de preuve ici, seulement une "conjecture")

(5*) Implémenter une fonction Python `etapes(a,b)`, qui renvoie un dictionnaire donnant, pour chaque volume entier v accessible, le nombre minimal d'étapes à réaliser pour obtenir v (dans A ou B). En déduire le nombre d'étapes minimales nécessaire pour mesurer un volume de 16 litres à partir de deux récipients de contenances respectives 31 litres et 37 litres.

On reprend le code de la fonction `configurations()`, avec essentiellement trois modifications :

- il est inutile de construire le graphe G (non retourné par le fonction), donc on se contente de construire l'ensemble de ses sommets (appelé G aussi dans la fonction);
- on rajoute une variable n, qui compte (en commençant à 0) le numéro du tour de boucle de la boucle principale (boucle `while N:`);
- dans la boucle sur les éléments (x, y) de N, on rajoute, si elles ne sont pas déjà présentes, les clés x et y dans le dictionnaire que l'on veut construire, avec comme valeur n.

```
def etapes(a,b):
    "retourne le nombre d'étapes pour chaque volume accessible"
    V = dict()
    G = set() # G est l'ensemble des sommets déjà traités (on ne construit pas le dictionnaire
    N = {(0,0)}
    n = 0 # numéro du tour de boucle principal (nombre d'étapes)
    while N: # tant que N n'est pas vide
        M = set()
        for x,y in N:
            # ajout des volumes x et y s'ils ne sont pas déjà présents dans V
            if x not in V:
                V[x] = n
            if y not in V:
                V[y] = n
            # construction de C
            C = set()
            if x<a and y>0: # vide B dans A
                d = min([y,a-x])
                C.add((x+d,y-d))
            if x>0 and y<b: # vide A dans B
                d = min([x,b-y])
                C.add((x-d,y+d))
            if x>0: # vide A
                C.add((0,y))
            if y>0: # vide B
                C.add((x,0))
            if x<a: # remplit A
                C.add((a,y))
            if y<b: # remplit B
                C.add((x,b))
            # mise à jour du prochain ensemble de sommets
            M |= C-G
            G |= C
        N = M
        n += 1
    return V
```

```
>>> E = etapes(31,37)
>>> print(E)
{0: 0, 1: 20, 2: 42, 3: 64, 4: 46, 5: 24, 6: 2, 7: 16, 8: 38, 9: 60, 10: 50, 11: 28,
12: 6, 13: 12, 14: 34, 15: 56, 16: 54, 17: 32, 18: 10, 19: 8, 20: 30, 21: 52, 22: 58,
23: 36, 24: 14, 25: 4, 26: 26, 27: 48, 28: 62, 29: 40, 30: 18, 31: 1, 32: 24, 33: 46,
34: 66, 35: 44, 36: 22, 37: 1}
>>> print(E[16])
```

Il faut au minimum 54 étapes pour obtenir 16 litres à partir de 31 et 37 litres

---

Indications

37.1 Comment s'interprète la quantité $\sum_{j=1}^{n} a_{ij}$?

37.3 Interpréter en termes de dénombrement la valeur de b_{1n}, où $B = (b_{ij})_{i,j}$ est la matrice définie par $B = A + A^2 + A^3 + \ldots + A^{n-1}$

37.4 Interpréter en termes de dénombrement la valeur de b_{ij}, où $B = (b_{ij})_{i,j}$ est la matrice définie par $B = A + A^2 + A^3 + \ldots + A^{n-1}$

37.5 Interpréter en termes de dénombrement la valeur de b_{ij}, où $B = (b_{ij})_{i,j}$ est la matrice définie par $B = A + A^2 + A^3 + \ldots + A^{d}$

38.1 Pour $n = 15$ les deux premiers coups sont forcés (Alice: $k = 1$ puis Bernard: $k = 2$), ensuite Alice a le choix entre $k = 3$ et $k = 4$. Montrer que dans les deux cas, Bernard peut trouver un coup qui mène Alice à une position perdante.

38.3 L'appel arbre(n,E,max) constuit un arbre dont la racine a pour étiquette n et pour enfants les sous-arbres obtenus par des appels récursifs de type arbre(n-k,E—k,2*k) pour les valeurs admissibles de k.

Le paramètre n représente le total disponible, l'ensemble E les entiers k déjà joués, et m est la borne maximale sur k due au coup précédent (ou imposée au départ)

38.5 Si x est un nœud de l'arbre, quel est le lien entre $g(x)$ et la possibilité de gagner à partir du nœud x ? (commencer par le cas des feuilles)

38.6 Utiliser une définition récursive de g.

39.1 Commencer comme pour un volume de 1 litre, et poursuivre ensuite les manipulations de base sans revenir en arrière jusqu'à tomber sur une configuration satisfaisante (2 litres dans l'un des récipients)

39.2 Compléter le pseudo-code suivant :

fonction configurations(a, b)
- // retourne le graphe des configurations accessibles depuis $(0, 0)$
- $G \leftarrow$ graphe vide (dictionnaire)
- $N \leftarrow$ .....
- tant que $N \neq \emptyset$
  - $M \leftarrow$ .....
  - pour tout élément (x, y) de N
    - fabriquer C, l'ensemble de toutes les configurations valides que l'on peut obtenir à partir de (x, y)
    - ajouter à M .....
    - ajouter à G la clé (x, y) associée à la valeur .....
  - $N \leftarrow M$
- retourner G

39.3 La construction de C (voir pseudo-code partiel ci-dessus) peut commencer comme ceci :

```
C = set()
if x<a and y>0: # vide B dans A
    d = min([y,a-x])
    C.add((x+d,y-d))
if y>0: # vide B
    C.add((x,0))
if y<b: # remplit B
    ......
......
```

39.4 La fonction `volumes()` récupère le dictionnaire G construit par `configurations(a,b)` et renvoie l'ensemble de tous les composantes (x et y) de tous les couples (x, y) contenus dans les clés de G.

39.5 On peut construire la fonction `etapes()` en modifiant légèrement la fonction `configurations()`. Dans la boucle sur les éléments (x, y) de N, on rajoute, si elles ne sont pas déjà présentes, les clés x et y dans le dictionnaire que l'on veut construire, avec comme valeur le numéro (commençant à 0) de tour de boucle principal (boucle "tant que $N \neq \emptyset$").