Devoir Surveillé d'ITC N°1
le 20/12/2024
MPSI & PCSI

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

- Les signatures des fonctions devront toutes être écrites.
- Les codes devront être les plus optimisés possibles, lorsque une telle optimisation semble évidente (par exemple, au moins de boucles s'arrêtant le plus tôt possible).

Problème Jeu du chocolat empoisonné

Partie I — Questions préliminaires au problème

1. Soit une liste `L` supposée existante. Proposer deux instructions différentes permettant d'ajouter la liste `[1,2]` à la liste `L` (pour une liste `L` égale à `[[0,0],[0,1]]`, on aura donc, après exécution d'une de ces instructions, modifié la liste `L` en `[[0,0],[0,1],[1,2]]`).
2. On considère une liste `L` d'entiers qui contient un certain nombre de zéros, tous situés successivement en fin de liste. Écrire une fonction `enleve_0(L:list)->list` qui prend en argument la liste `L` et qui renvoie une nouvelle liste contenant les mêmes valeurs que celles de la liste `L`, à l'exception des valeurs nulles. Ainsi `enleve_0([2,1,0,0])` devra renvoyer la liste `[2,1]`. *Cette fonction ne devra pas modifier la liste* `L` *passée en argument.*

Partie II — Étude du jeu du chocolat empoisonné

On s'intéresse au jeu du chocolat empoisonné (ou *chomp* en anglais).

Ce jeu se joue à deux joueurs que l'on notera J_1 et J_2 par la suite. On dispose d'une tablette de chocolat, de taille $n \times m$, contenant n lignes et m colonnes, pour laquelle le carré situé en haut à gauche est empoisonné (les joueurs le savent). Chaque carré de la tablette est repéré par un couple d'entiers (i,j), avec $0 \leq i \leqslant n-1$, $0 \leq j \leqslant m-1$; ici i désigne le numéro de ligne (ordonnée sur les graphiques ci-dessous), et j le numéro de colonne (abscisse sur les graphiques ci-dessous).

À tour de rôle, chaque joueur, en commençant par J_1, doit *jouer un coup*, ce qui consiste à choisir un carré d'indices (i_c, j_c), à le manger, et à manger également tous les carrés présents dans la tablette d'indices (i,j) vérifiant $i \geq i_c$ et $j \geq j_c$ (c'est-à-dire tous les carrés qui se situent en dessous et à droite du carré choisi). Le joueur qui mange le carré empoisonné meurt et est déclaré perdant. Il n'est pas possible de passer son tour.

La figure ci-dessous montre le début d'une partie pour une tablette de taille initiale 3×5.

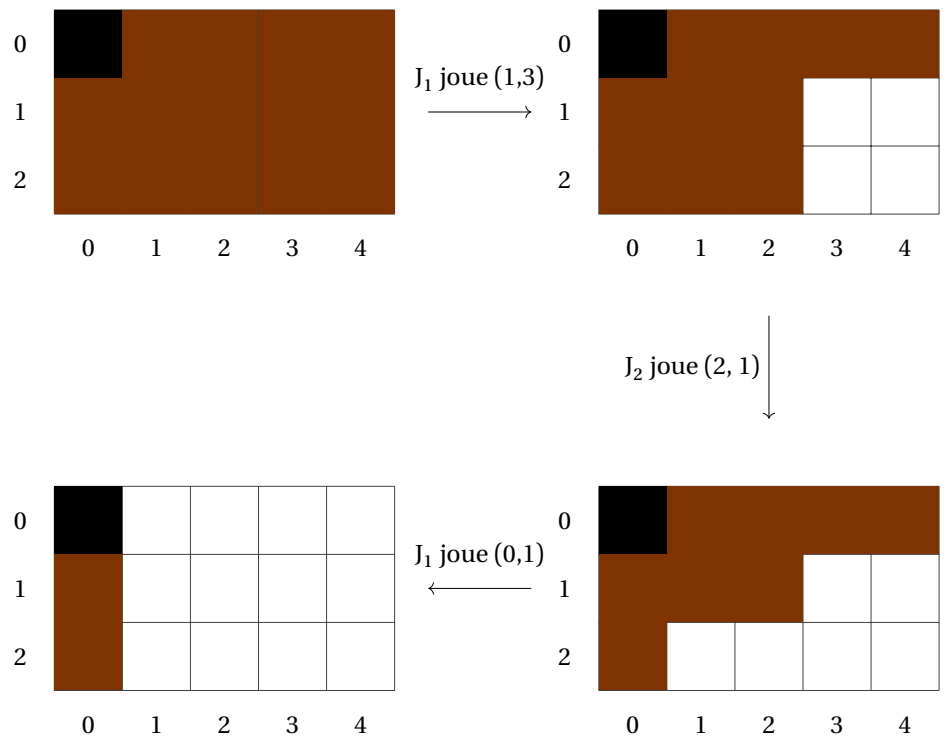

FIGURE 1. – Exemple de partie

Au cours d'une partie, au moment de jouer, chaque joueur est en face d'une tablette entamée, appelée *configuration*, et notée $\mathscr{C}$. La *configuration* initiale de la tablette (à laquelle est confrontée J_1 pour la première fois), correspond ainsi à une tablette comportant $n \times m$ carrés de chocolat disposés en n lignes et m colonnes.

Sans trop formaliser le problème,

- on dit qu'une configuration est *gagnante* si la personne qui doit jouer à partir de cette configuration peut effectuer une série de coups qui lui permettent de gagner, et ceci quels que soient les coups que pourra effectuer son adversaire. Le premier coup d'une telle série sera appelé *coup gagnant*.
- On dit qu'une configuration est *perdante* si elle n'est pas gagnante ; c'est-à-dire si le joueur qui doit jouer à partir de cette configuration ne peut s'opposer à la victoire de son adversaire quelque soit le ou la série de coups qui sera effectuée par le joueur.

Afin de pouvoir coder des fonctions associées à ce jeu, on choisit d'utiliser les structures de données suivantes :

- un coup (i,j) sera représenté sous la forme d'une liste à deux éléments `[i,j]`,
- une configuration $\mathscr{C}$ sera représentée par une liste d'entiers contenant le nombre de carrés de chocolat présents dans chaque ligne (de la ligne 0 à la dernière ligne de la configuration). Ainsi les quatre configurations successives décrites en Figure 1 sont :
  - `[5,5,5]`
  - `[5,3,3]`
  - `[1,1,1]`
  - `[5,3,1]`

Par convention, une telle liste ne contiendra jamais de 0. On écrira donc `[2, 1]` au lieu de `[2, 1, 0, 0]`.

3. On donne quatre exemples de configurations :

a) Ligne 1×5 :

b) Configuration « en L » avec deux branches de même longueur contenant chacune trois carrés :

c) Configuration « en L » avec deux branches de longueurs différentes (cinq carrés sur la ligne, trois sur la colonne) :

d) Tablette carrée de taille 4×4 :

Pour chaque configuration :

3.1) Donner la liste associée à la configuration.

3.2) Préciser le caractère gagnant ou perdant de la configuration (pour les justifications, on peut supposer que c'est à J_1 de jouer, et décrire les coups successifs de J_1 et J_2 permettant de conclure).

4. Écrire le script d'une fonction `tablette(n:int,m:int)->list` qui prend en argument deux entiers n et m et qui renvoie la liste représentant la configuration correspondant à une tablette de n lignes et m colonnes.

5. On désire écrire le script d'une fonction `affiche(conf:list)->None` qui prend en argument une liste `conf` correspondant à une configuration $\mathscr{C}$ donnée, et qui fournit un affichage de cette configuration , le caractère `'o'` correspondant au carré empoisonné, le caractère `'*'` correspondant à un carré standard. Ainsi, on aura par exemple :

```
>>> affiche([5,3,1])
o****
***
*
```

Pour cela, on peut utiliser la fonction `print()` au sein d'une boucle **`for`**. À titre d'exemple, on donne le script suivant :

```
def exemple_affiche()->None:
    L = ['a','b','c']
    for el in L:
        print(el)
```

dont l'exécution fournit le résultat suivant :

```
>>> exemple_affiche()
a
b
c
```

Écrire le script de la fonction `affiche(conf:list)->None` demandée.

6. Écrire le script d'une fonction `nb_coups(conf:list)->int` qui prend en argument la liste `conf` correspondant à une configuration donnée, et qui renvoie le nombre de coups, c'est à dire le nombre de carrés mangeables, associé à cette configuration. On rappelle que le carré empoisonné n'est pas mangeable.

7. Écrire le script d'une fonction `coups_jouables(conf:list)->list` qui prend en argument une liste correspondant à une configuration donnée, et qui renvoie la liste de tous les coups jouables de cette configuration (c'est-à-dire la liste des coordonnées `[i,j]` des carrés de la configuration). La liste renvoyée est donc une liste de listes à deux éléments, et on convient du fait que `[0,0]` n'est pas un coup jouable. Cette fonction ne devra pas modifier la liste `conf` passée en argument.

Quand on joue le coup $[i_c, j_c]$ sur une configuration $\mathscr{C}$ donnée, le nombre de carrés de chocolat des lignes d'indices i strictement inférieurs à i_c n'est pas modifié, alors que pour les autres lignes, ce nombre devient égal à j_c lorsque j_c est strictement inférieur au nombre de carrés sur la ligne i, et n'est pas modifié sinon.

8. On donne ci-dessous le début du script d'une fonction `joue(conf:list,coup:list)->list` qui prend en argument une liste `conf` correspondant à une configuration $\mathscr{C}$ donnée, une liste `coup` constituée de deux entiers tels que cette liste désigne un coup jouable de la configuration $\mathscr{C}$, et qui renvoie une liste correspondant à la nouvelle configuration obtenue une fois le coup joué. Cette liste ne devra contenir aucun zéro.

```
def joue(conf:list,coup:list)->list:
    nl = len(conf) # nombre de lignes total de la configuration
    [ic,jc] = coup # coordonnées du coup joué
    n_conf = [] # nouvelle configuration
    #
    # A compléter
    #
    return n_conf
```

Recopier et compléter le script de cette fonction pour qu'elle renvoie la liste demandée.

Le caractère perdant ou gagnant d'une configuration peut être défini plus rigoureusement de la manière suivante :

- La configuration où il ne reste que le carré empoisonné est perdante.
- Pour une configuration $\mathscr{C}$ donnée,
  - s'il existe un coup jouable permettant de passer de $\mathscr{C}$ à une configuration perdante (pour l'adversaire donc), alors $\mathscr{C}$ est gagnante,
  - si tout coup jouable à partir de $\mathscr{C}$ amène dans une configuration gagnante (pour l'adversaire là encore), alors $\mathscr{C}$ est perdante.

9. Recopier et compléter le code de la fonction récursive `gagnante(conf:list)->bool` suivante, basée sur la définition précédente, qui prend en argument une liste `conf` correspondant à une configuration $\mathscr{C}$ et qui renvoie **`True`** si $\mathscr{C}$ est gagnante, et **`False`** sinon.

```
def gagnante(conf:list)->bool:
    # cas de base
    if conf == [1]:
        return ######
    # autres cas
    else:
        LC = coups_jouables(######)
        for coup in LC:
            n_conf = joue(########)
            if ######## == False:
                ######
        ######
```

10. On modifie la fonction précédente en rajoutant, au début de la fonction, les deux lignes suivantes :

```
global c
c += 1
```

10.1) Préciser le rôle de l'instruction **`global`** `c`.

10.2) On initialise `c` à 0, puis on appelle la fonction `gagnante`. Donner une interprétation du contenu de la variable `c` après exécution de `gagnante`.

11. Le nombre nb de configurations différentes atteignables à partir d'une tablette de taille 4×5 peut se calculer à l'aide de la formule suivante

$$nb = \left(\sum_{a=1}^{5} \sum_{b=0}^{a} \sum_{c=0}^{b} \sum_{d=0}^{c} 1 \right) - 1.$$

11.1) Écrire le script d'une fonction `nb_4_5()->int`, sans argument, qui renvoie le nombre nb défini précédemment.

11.2) On donne le résultat de l'exécution des lignes suivantes :

```
>>> c = 0
>>> gagnante(tablette(4,5))
>>> print(c)
95440
>>> nb_4_5()
124
```

Commenter le résultat obtenu.

Afin d'éviter les calculs redondants, on souhaite mémoriser le caractère gagnant ou perdant des configurations déjà rencontrées. Pour cela, on va utiliser un dictionnaire appelé *dictionnaire des configurations* :

- dont les clés correspondent aux configurations,
- dont les valeurs sont des booléens (`True` si la configuration est gagnante, `False` sinon).

Les clés d'un dictionnaire ne peuvent pas être de type `list`, et chaque configuration est stockée sous forme de liste. Il est donc nécessaire de disposer d'une fonction qui convertit le contenu d'une liste. On choisit ici de convertir en chaine de caractère, à l'aide d'une fonction `conf_to_str(conf:list)->str` dont on donne quelques exemples de fonctionnement ci-dessous.

```
>>> conf_to_str([1])
'1'
>>> conf_to_str([3,2])
'3-2'
>>> conf_to_str([5,2,1])
'5-2-1'
```

12. Écrire le script de la fonction `conf_to_str(conf:list)->str` décrite précédemment.

On rappelle que pour un dictionnaire passé en argument d'une fonction, toute modification du dictionnaire effectuée à l'intérieur de la fonction se répercute sur le contenu du dictionnaire une fois la fonction terminée[1]. Ainsi, pour une fonction

```
def f(d:dict)->None:
    d['1'] = False
```

on aura le comportement suivant :

```
>>> d = {}
>>> f(d)
>>> d
{'1': False}
```

Notez bien la modification de `d` en dehors de la fonction. On dit que `d` a été modifié par effet de bord lors de l'exécution de `f`.

13. On désire écrire le script d'une fonction `gagnante_memo(conf:list,d:dict)->bool` qui prend en argument une liste `conf` correspondant à une configuration donnée, un dictionnaire `d` correspondant à un dictionnaire des configurations, qui procède récursivement comme la fonction `gagnante`, et qui renvoie `True` si `conf` est gagnante, et `False` sinon. Afin d'éviter des appels récursifs redondants, cette fonction devra commencer par tester si la configuration apparaît déjà dans le dictionnaire, auquel cas elle renverra la valeur correspondante. Dans le cas contraire, la fonction évaluera récursivement le caractère gagnant ou perdant de la configuration, et la configuration et la valeur associée seront rajoutées au dictionnaire avant de renvoyer le résultat.
Recopier et compléter le code le la fonction suivante :

```
def gagnante_memo(conf:list,d:dict)->bool:
    s = conf_to_str(conf)
    if s in d:
        return #####
    else:
        LC = coups_jouables(####)
        for coup in LC:
            n_conf = joue(######)
            n_conf_str = conf_to_str(n_conf)
            if ####### == False:
                ######
                ######
            else:
                ######
        return #####
```

14. On modifie la fonction `gagnante_memo` en rajoutant au début de la fonction les instructions suivantes :

```
global c
c += 1
```

1. ceci est vrai pour toute variable mutable

l'exécution des instructions ci-après dans la console conduit alors au résultat suivant :

```
>>> c = 0
>>> d = {'1':False}
>>> gagnante_memo([5,5,5,5],d)
>>> print(c)
379
```

Commenter le résultat obtenu.

15. À l'aide de l'instruction `from time import time`, on peut importer la fonction `time()->float`, sans argument, qui renvoie un flottant correspondant au nombre de secondes écoulées depuis le 1er janvier 1970 à 0:00. Ainsi, une exécution donnera par exemple :

```
>>> from time import time
>>> time()
1736257806.535462
```

Écrire le script d'une fonction `teste_temps(conf:list)->(float,float)` qui prend en argument une liste `conf` décrivant une configuration donnée, et renvoie les flottants t_1 et t_2 donnant respectivement les temps d'exécution des fonctions `gagnante` et `gagnante_memo` appliquées à la configuration `conf`.

16. Commenter le résultat de l'appel suivant :

```
>>> teste_temps([6,6,6,6,6])
(17.385705947875977, 0.015863656997680664)
```

On souhaite maintenant trouver un coup gagnant (s'il existe) dans une configuration donnée. On souhaite donc écrire une fonction `coup_gagnant(conf:list)->list` prenant en argument une liste `conf` décrivant une configuration et qui renvoie

- la liste vide `[]` si la configuration est perdante,
- un coup gagnant sous forme d'une liste de deux entiers si la configuration est gagnante.

Cette fonction devra utiliser la fonction `gagnante_memo`, et dans un souci d'efficacité, on évitera de réévaluer plusieurs fois le caractère gagnant ou perdant d'une configuration.

17. On donne le script incomplet suivant. Compléter le script de cette fonction

```
def coup_gagnant(conf:list)->list:
    d = {'1':False}
    LC = coups_jouables(conf)
    for coup in LC:
        n_conf = joue(####)
        n_conf_str = conf_to_str(n_conf)
        if ######## == False:
            ##########
        else:
            ##########
    return #####
```

On souhaite maintenant visualiser l'ensemble des coups gagnants depuis une configuration de départ en tablette rectangulaire $n \times m$. Pour cela, on souhaite créer une carte des coups gagnants, sous la forme d'une liste des listes `C` telle que `C[i][j]` renseigne sur le caractère gagnant ou perdant du coup `[i,j]`. Plus précisément,

- `C[0][0]` a pour valeur 0,
- Pour tout $(i,j) \in [\![0, n-1]\!] \times [\![0, m-1]\!]$ tel que $(i,j) \neq (0,0)$,
  - ◇ `C[i][j]` vaut 1 si le coup `[i,j]` mène à une configuration gagnante,
  - ◇ `C[i][j]` vaut 2 si le coup `[i,j]` mène à une configuration perdante.

18. Recopier et compléter le script de la fonction `carte_gagnante(n:int,m:int)->list` ci-dessous de manière à ce qu'elle renvoie la liste demandée.

```
def carte_gagnante(n:int,m:int)->list:
    d = {'1':False}
    conf = tablette(n,m)
    C = [] # liste de la carte finale
    for i in range(n):
        L =[] # liste des résultats des coups de la ligne \
        ↪ courante
        for j in range(m):
            if (i,j) == (0,0):
                #########
            else:
                coup = [i,j]
                ############
                ##########
        C.append(L)
    return C
```

19. On donne ci-dessous le résultat d'un appel à la fonction carte gagnante.

```
>>> carte_gagnante(4,4)
[[0, 1, 1, 1], [1, 2, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

Quel résultat retrouve-t-on ici?

Correction
du Devoir Surveillé d'ITC N°1 (le 20/12/2024)
MPSI & PCSI

Solution

1. On peut proposer les deux instructions suivantes :

```
L = L+[[1,2]] # attention au double crochet
# ou bien
L.append([1,2])
```

2. Fonction `enleve_0`

```
def enleve_0(L:list)->list:
    n_L = []
    for e in L:
        if e != 0:
            n_L.append(e)
    return n_L
```

```
>>> enleve_0([1, 2, 0, 0])
[1, 2]
>>> enleve_0([1, 2])
[1, 2]
```

Encore mieux, on peut utiliser un **while** et arrêter quand il le faut.

```
def enleve_0(L:list)->list:
    n_L = []
    n = 0
    while n < len(L) and L[n] != 0:
        n_L.append(L[n])
        n += 1
    return n_L
```

```
>>> enleve_0([1, 2, 0, 0])
[1, 2]
>>> enleve_0([1, 2])
[1, 2]
```

3. Nature des configurations

3.1) **a)** `[5]`

b) `[3,1,1]`

c) `[5,1,1]`

d) `[4,4,4,4]`

3.2) **a)** Si J_1 joue le coup $(0,1)$, alors J_2 est obligé de jouer $(0,0)$. La configuration est donc $\boxed{\textit{gagnante}}$.

b) Si J_1 élimine un ou plusieurs carrés sur une branche, alors J_2 peut faire de même sur la branche symétrique. En répétant l'opération, on aboutit forcément sur une configuration pour J_2 de type $1 \times n$ (ou $n \times 1$) et d'après ce qui précède, J_2 gagne. La configuration est donc $\boxed{\textit{perdante}}$.

c) J_1 peut faire en sorte de modifier la configuration pour que J_2 soit dans le cas du « L » avec deux branches de même longueur. J_2 est alors en configuration perdante (d'après le cas précédent), et donc J_1 gagne. La configuration est $\boxed{\textit{gagnante}}$.

d) J_1 peut jouer le couple $(1,1)$, on a alors J_2 qui se retrouve en configuration « en L » avec deux branches de même longueur, qui est perdante. La configuration en carré est donc $\boxed{\textit{gagnante}}$.

4. Fonction `tablette` :

```
def tablette(n:int,m:int)->list:
    L = []
    for _ in range(n):
       L.append(m)
    return L
```

```
>>> tablette(5, 5)
[5, 5, 5, 5, 5]
```

5. Fonction `affiche`

```
def affiche(conf:list)->None:
    print('o'+'*'*(conf[0]-1)) # première ligne, traitée à part
    for i in range(1,len(conf)): # lignes suivantes
        print('*'*conf[i])
```

```
>>> affiche([5, 3, 1])
o****
***
*
```

6. Fonction `nb_coups`

```
def nb_coups(conf:list)->int:
    nb = conf[0]-1 # on enlève le carré empoisonné de la \
    ↪ première ligne
    for i in range(1,len(conf)):
        nb += conf[i]
    return nb
```

```
>>> nb_coups([5, 3, 1])
```

```
8
```

7. Fonction `coups_jouables`

```
def coups_jouables(conf:list)->list:
    LC = [] # liste des coups jouables
    # première ligne, le coup (0,0) n'est pas jouable
    for j in range(1,conf[0]):
        LC.append([0,j])
    for i in range(1,len(conf)):
        for j in range(conf[i]):
            LC.append([i,j])
    return LC
```

```
>>> coups_jouables([5, 3, 1])
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 0], [1, 1], [1, 2], [2, 0]]
```

8. Fonction `joue`

```
def joue(conf:list,coup:list)->list:
    nl = len(conf) # nombre de lignes total de la configuration
    [ic,jc] = coup # coordonnées du coup joué
    n_conf = [] # nouvelle configuration
    for i in range(ic):
        n_conf.append(conf[i])
    for i in range(ic,nl):
        if conf[i] > jc:
            n_conf.append(jc)
        else:
            n_conf.append(conf[i])
    n_conf = enleve_0(n_conf) # on enlève les zéros éventuels \
    ↪ à la fin (si croque bande de gauche)
    return n_conf
```

```
>>> joue([5, 3, 1], [1, 0])
[5]
```

9. Fonction `gagnante`

```
c = 0
def gagnante(conf:list)->bool:
    global c
    c += 1
    # cas de base
    if conf == [1]:
        return False
    # autres cas
    else:
        LC = coups_jouables(conf)
        for coup in LC:
            n_conf = joue(conf,coup)
            if gagnante(n_conf) == False:
                return True
        return False
```

Testons par exemple notre fonction sur les configurations de début de sujet.

```
>>> gagnante([5])
True
>>> gagnante([3,1,1])
False
>>> gagnante([5,1,1])
True
>>> gagnante([4,4,4,4])
True
```

10. **10.1)** L'instruction `global c` précise que `c` doit être considérée comme une variable globale, qui existe en dehors de la fonction.

10.2) Le contenu de `c` après exécution correspond au nombre total d'appels récursifs mis en jeu lors de l'exécution de la fonction.

11. **11.1)** fonction `nb_4_5`

```
def nb_4_5()->int:
    nb = 0
    for a in range(1,6):
        for b in range(a+1):
            for c in range(b+1):
                for d in range(c+1):
                    nb += 1
    return nb-1
```

11.2)

```
>>> c = 0
>>> gagnante(tablette(4, 5))
True
>>> c
95440
>>> nb_4_5()
124
```

On a 95440 >> 379, ce qui montre que l'on effectue de nombreux appels récursifs pour tester le caractère gagnant ou perdant d'une même configuration.

12. fonction `conf_to_str`

```
def conf_to_str(conf:list)->str:
    s = ''
    for e in conf[:-1]:
        s += str(e) + '-'
    s += str(conf[-1])
    return s
```

13. fonction gagnante_memo

```
def gagnante_memo(conf:list,d:dict)->bool:
    global c
    c += 1
    s = conf_to_str(conf)
    if s in d:
        return d[s]
    else:
        LC = coups_jouables(conf)
        for coup in LC:
            n_conf = joue(conf,coup)
            n_conf_str = conf_to_str(n_conf)
            if gagnante_memo(n_conf,d) == False:
                d[n_conf_str] = False
                return True
            else:
                d[n_conf_str] = True
        return False
```

Instructions pour utiliser gagnante_memo

```
>>> c = 0
>>> d = {'1':False} # initialisation nécessaire du dictionnaire
>>> gagnante_memo([4,4,4,4], d)
True
>>> c
107
```

Autre version possible (mais ne tenant pas compte des trous dans le sujet, qui effectue moins de modifications de dictionnaire) :

```
def gagnante_memo(conf:list,d:dict)->bool:
    global c
    c += 1
    s = conf_to_str(conf)
    if s in d:
        return d[s]
    else:
        LC = coups_jouables(conf)
        for coup in LC:
            n_conf = joue(conf,coup)
            if gagnante_memo(n_conf,d) == False:
                d[s] = True
                return True
        d[s] = False
        return False
```

```
>>> c = 0
>>> d = {'1':False} # initialisation nécessaire du dictionnaire
>>> gagnante_memo([4,4,4,4], d)
True
>>> c
107
```

14. L'emploi du dictionnaire a permis de diminuer le nombre d'appels récursifs par rapport au cas de la fonction gagnante.

15. fonction teste_temps

```
def teste_temps(conf:list)->(float,float):
    tic = time()
    gagnante(conf)
    tac = time()
    t1 = tac-tic

    tic = time()
    d = {'1':False}
    gagnante_memo(conf,d)
    tac = time()
    t2 = tac-tic

    return (t1,t2)
```

```
>>> teste_temps([6,6,6,6,6])
(7.136997222900391, 0.0012629032135009766)
```

16. De manière cohérente avec le nombre d'appels récursifs, le temps d'exécution est beaucoup plus court pour gagnante_memo que pour gagnante.

17. fonction coup_gagnant complétée

```
def coup_gagnant(conf:list)->list:
    d = {'1':False}
    LC = coups_jouables(conf)
    for coup in LC:
```

```
        n_conf = joue(conf,coup)
        n_conf_str = conf_to_str(n_conf)
        if gagnante_memo(n_conf,d) == False:
            return coup
        else:
            d[n_conf_str] = True
    return []
```

18. fonction `carte_gagnante` complétée

```
def carte_gagnante(n:int,m:int)->list:
    d = {'1':False}
    conf = tablette(n,m)
    C = [] # liste de la carte finale
    for i in range(n):
        L =[] # liste des résultats des coups de la ligne courante
        for j in range(m):
            if (i,j) == (0,0):
                L.append(0)
            else:
                coup = [i,j]
                n_conf = joue(conf,coup)
                n_conf_str = conf_to_str(n_conf)
                if gagnante_memo(n_conf,d) == False:
                    d[n_conf_str] = False
                    L.append(2)
                else:
                    d[n_conf_str] = True
                    L.append(1)
        C.append(L)
    return C
```

19. Le coup `[1,1]` mène à une configuration perdante, c'est donc un coup gagnant. On retrouve le fait qu'une tablette carrée correspond à une configuration gagnante, ou bien qu'une configuration « en L » avec branches de même longueur est perdante, comme expliqué en question 3.