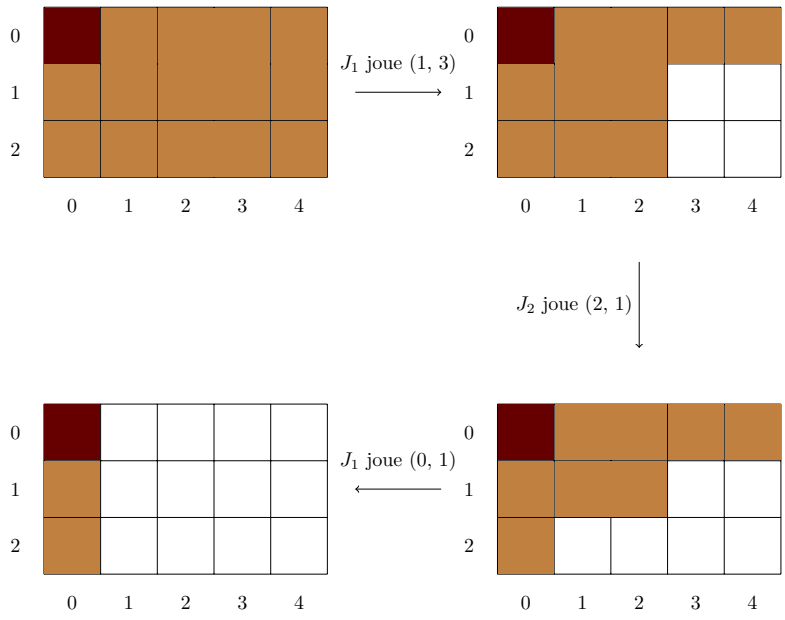
Le chocolat empoisonné

Ce TP est inspiré d'un TP partagé par un collègue, M. Péchaud.
Toutes les fonctions *Python* écrites devront être documentées, et testées.

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

Ce jeu se joue à deux joueurs, que l'on notera dans la suite J_1 et J_2. On dispose d'une tablette de chocolat au lait, et l'on suppose que le carré en haut à gauche est empoisonné.
À tour de rôle, chaque joueur, en commençant par J_1 doit manger l'un des carrés (repéré par ses indices, voir le schéma), ainsi que tout ceux qui se trouvent au dessous et/ou à droite. La personne qui mange le carré empoisonné meurt et a donc perdu. Le dernier vivant a gagné. On ne peut pas passer son tour.
Le schéma ci-dessous montre le début d'une partie qui démarre d'une tablette 3×5.



1. Justifiez qu'une partie se termine toujours, et qu'il y a à la fin un unique vivant. En combien de coup maximum une partie se termine-t-elle ?

On s'intéresse dans ce TP à un algorithme permettant de déterminer si une position donnée est *gagnante* ou *perdante* – et, dans le premier cas, de trouver un *coup gagnant*.

Intuitivement, et sans chercher à définir très formellement les choses,

- On dira qu'une position est *gagnante* si la personne qui joue à partir de cette position peut parvenir à gagner en fin de partie quels que soient les coups de son adversaire. Le premier coup de la séquence permettant d'y parvenir est appelé *coup gagnant*.

- La position est dite *perdante* sinon.

2. À l'aide d'une feuille et d'un crayon (ou de petits carrés découpés), jouez avec votre voisin ou voisine[1] pour vous familiariser avec le jeu. Que pensez-vous du caractère gagnant ou perdant des positions suivantes :
   (a) Ligne $n \times 1$ ou $1 \times n$ (où $n \geq 1$) ? 
   (b) Position en «L» avec les deux branches de même longueur 
   (c) Position en «L» avec les deux branches de longueurs différentes 
   (d) Tablette carrée 

Nous allons commencer à écrire des fonctions permettant de tester si une position est gagnante ou perdante. Nous choisissons pour ce faire de coder une position sous forme d'une liste d'entiers contenant le nombre de carrés de chocolats dans chaque ligne, les lignes étant parcourues de haut en bas. Ainsi, les quatre positions successives données sur la première page correspondent aux listes suivantes :

- `[5, 5, 5]`
- `[5, 3, 3]`
- `[5, 3, 1]`
- `[1, 1, 1]`

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

Dans la suite de l'énoncé, le terme *position* désignera une telle liste.

Par ailleurs, un *coup* sera codé par le couple de coordonnées correspondant au carré choisi.

3. Compléter la fonction `tablette(n, m)` qui prend en argument deux entiers non-nuls, et renvoie la position correspondant à une tablette à m colonnes et n lignes (tablette complète).

> On dispose dans la feuille de script d'une fonction `affiche` qui affiche une position donnée. Rappelons qu'une position est une liste d'entiers

4. Compléter la fonction `coups_jouables(pos)`, qui prend en argument une position et renvoie la liste des coups jouables – i.e. la liste des coordonnées de tous les carrés de la position. Par convention, $(0, 0)$ n'est pas un coup jouable.
   Attention : un coup jouable est donné par (i, j) des indices : i est l'indice de ligne et j l'indice de colonne, dans la tablette.
   Indication : à la main, quels sont les coups jouables pour la position `[5,3,3]` représentée plus haut (position après un coup joué de l'exemple) ?
5. Compléter la fonction `joue(pos, coup)`, qui prend en argument une position et un coup jouable pour cette position, et renvoie une nouvelle position correspondant à la position obtenue en jouant le coup à partir de la position de départ. Il est conseillé, avant de se lancer dans l'écriture du code, d'effectuer quelques essais à la main.
   Quelle est la complexité de votre fonction ?
   Indication : on note (i_0, j_0) le coup à jouer. Quels sont les indices des lignes impactées ? Combien de carrés reste-t-il dans chaque ligne impactée ?

> On appelle la fonction `enleve_zeros` sur la position calculée, et avant de la retourner, pour respecter la convention qu'une position ne se termine jamais par des zéros

On donne maintenant les propriétés suivantes sur les positions gagnantes et perdantes :

1. Sans manger les bouts de papier.

- La position où ne reste que le carré empoisonné est perdante.
- Pour une position $\mathcal{P}$ donnée,
  - s'il existe un coup jouable permettant de passer de $\mathcal{P}$ à une position perdante, alors $\mathcal{P}$ est gagnante,
  - si tout coup jouable à partir de $\mathcal{P}$ amène dans une position gagnante, alors $\mathcal{P}$ est perdante.

On en déduit une stratégie *récursive* permettant de décider si une position est gagnante ou perdante.

6. Compléter la fonction `gagnante(pos)` qui prend en argument une position, et teste si elle est gagnante (elle renverra un booléen). Penser à la tester sur quelques exemples.
   Pour des tests plus poussés, exécuter

   ```
   test_gagnantes()
   ```

   dans la console.
7. (a) Justifier que votre fonction termine.
   (b) Afin de compter le nombre d'appels récursifs effectués par votre fonction, copier le code de `gagnante` dans la fonction `gagnante_compteur` après les lignes

   ```
   global compteur
   compteur += 1
   ```

   Tester alors par exemple avec

   ```
   compteur = 0
   print(gagnante(tablette(4, 5)))
   print(compteur)
   ```

   Combien y a-t-il liste de *position* différentes pour une tablette de 4×5 (en comptant toutes les positions envisageable, pas seulement celles qui sont atteignables) ? Qu'en déduire ?

Afin d'éviter les calculs redondants, on souhaite mémoriser le caractère gagnant ou perdant des positions déjà rencontrées. On parle de *mémoïsation*.

Pour cela, on va utiliser un *dictionnaire*

- dont les clés correspondent aux positions
- dont les valeurs sont des booléens indiquant si la position est gagnante ou perdante.

Il est impossible d'utiliser une liste `pos` comme clé d'un dictionnaire, donc on utilisera à la place un tuple ou une chaîne, que vous pourrez obtenir à l'aide des fonctions `tuple(pos)` ou `str(pos)`.

Ainsi, les entrées du dictionnaire ressembleront à

```
{
'[2]' : True,
'[2,1]' : False,
'[2,2]' : True,
...
}
```

8. Compléter la fonction `gagnante_memo(pos, d)`, qui prend en argument une position et un dictionnaire, en vous inspirant de la fonction `gagnante`. Cependant, la nouvelle fonction devra commencer par tester si la position apparaît déjà dans le dictionnaire, auquel cas il suffira de renvoyer la valeur correspondante. Si ça n'est pas le cas, la position et sa valeur devront être rajoutées au dictionnaire avant de renvoyer le résultat. `d` sera donc modifié par effet de bord [2]
9. Testez votre fonction, et comparez empiriquement (grâce à la fonction `teste_temps`) son temps d'exécution à celui de `gagnante`. Quelle est sa complexité ?

2. une modification d'un objet dans une fonction, qui est visible à l'extérieur de la fonction. Possible seulement sur les objets mutables : listes, dictionnaires

Pour aller plus loin.

10. Tester `gagnante_memo` sur des tablettes rectangulaires de différentes tailles. Que pouvez-vous conjecturer ? Prouver votre conjecture !

On souhaite maintenant trouver un coup gagnant (s'il existe) dans une position donnée.

11. Écrire une fonction `coup_gagnant(pos)`, utilisant `gagnante_memo` qui renvoie
    - `False` si la position est perdante
    - un coup gagnant sous forme de couple si elle est gagnante.

    Dans un but d'efficacité, on évitera ici aussi de réévaluer plusieurs fois le caractère gagnant ou perdant d'une même position.

On souhaite maintenant visualiser l'ensemble des coups gagnants depuis une position de départ en tablette rectangulaire $n \times m$.

12. Écrire une fonction `carte_gagnante(m, n)`, qui prend en argument deux entiers non-nuls m et n et renvoie une matrice[3] $M = (m_{i,j})_{\substack{i \in [\![0,n-1]\!] \\ j \in [\![0,m-1]\!]}} \in \mathcal{M}_{n,m}(\mathbb{R})$ telle que
    - $m_{0,0} = 0$
    - Pour tout $(i,j) \in [\![0,n-1]\!] \times [\![0,m-1]\!]$ tel que $(i,j) \neq (0,0)$
      — $m_{i,j} = 1$ si le coup (i,j) mène à une position gagnante
      — $m_{i,j} = 2$ si le coup (i,j) mène à une position perdante
13. Tester, et afficher les résultats obtenus à l'aide de la fonction fournie. Que pouvez-vous conjecturer ? [4]

À noter que l'on ne connaît pas à l'heure actuelle de formule simple permettant d'obtenir un coup gagnant pour une tablette $m \times n$ (et même $3 \times n$).

La page suivante regroupe notamment un certain nombre de résultats et conjectures sur cette question :
`https://www.win.tue.nl/~aeb/games/chomp.html`

Annexe

Voici les fonctions et méthodes sur les listes et dictionnaires dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste n). Tout autre fonction dont vous auriez besoin doit être implémentée !

Fonctions et méthodes sur les listes

Opération	Exemple	Complexité
Création d'une liste vide	`l=[]`	$O(1)$
Accès direct	`l[0]`	$O(1)$
Longueur	`len(l)`	$O(1)$
Concaténation	`l1+l2`	$O(n1+n2)$
Ajout en fin de liste	`l.append(1)`	$O(1)$
Suppression en fin de liste	`l.pop()`	$O(1)$
Extraction de tranche	`l[1:10]`	$O(n)$, où n est la longueur de la tranche.
Répétition	`[0]*k`	$O(n)$, où n est la longueur de la liste créée.
Création par compréhension	`[k**2 for k in range(n)]`	$O(n)$ si l'expression est évaluée en temps constant

3. Sous forme de liste de listes.
4. Votre conjecture est …fausse ! Tester votre fonction sur une tablette 8×10, ce qui devrait prendre quelques dizaines de secondes selon la puissance de votre machine.

Fonctions et méthodes sur les dictionnaires

Opération	Exemple	Complexité
Création	`d = {cle : valeur}`	$O(1)$
Test d'appartenance d'une clé	`cle in d`	$O(1)$
Ajout d'un couple clé/valeur	`d[cle] = valeur`	$O(1)$
Valeur correspondant à une clé	`d[cle]`	$O(1)$
Itération sur les clés	`for cle in d:`	$O(n)$, le nombre de paires clé/valeur