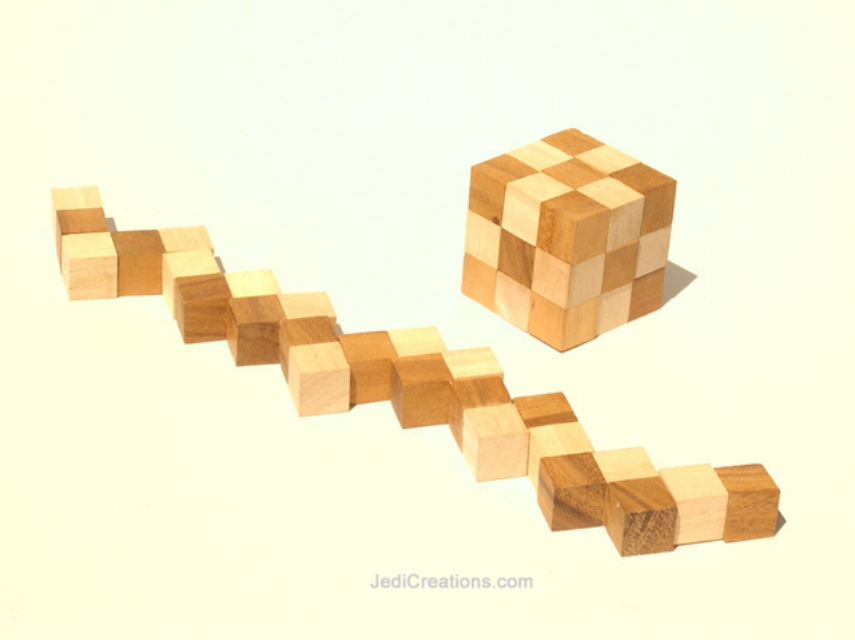
TP RÉCURSIVITÉ : RÉSOLUTION DU PUZZLE SNAKE CUBE PAR BACKTRACKING

1. Enregistrez votre travail dans un dossier `TP_snake`.
Le but du TP de résoudre le puzzle suivant :



L'idée est que l'on a une série de $27 = 3^3$ cubes groupés par deux deux ou trois, chaque groupe pouvant pivoter par rapport au groupe suivant. On souhaite trouver une (ou toutes les...) combinaison·s qui permette·nt de reconstituer un cube.

Il existe plusieurs puzzles c'est-à-dire plusieurs suites de groupes de deux ou trois cubes possibles.

L'idée pour résoudre le problème va être la suivante : on va remplir un cube de taille $3 \times 3 \times 3$ à partir de l'une des extrémités du « serpent » fixée à l'avance.

— On part d'une certaine position dans le cube, dans une certaine direction.

— On avance de 2 places dans le cube car le premier groupe contient trois cubes.

— Puis on repart dans une direction différente avec le groupe suivant.

— Lorsque l'on ne peut plus avancer sans sortir du cube, on est dans une impasse : on essaye une autre direction pour le groupe précédent (backtracking).

Généralités

Chaque groupe de 3 (respectivement 2) cubes demande d'avancer de 2 (respectivement 1) place·s dans le cube. On modélisera donc notre « serpent » par la variable globale :

```
groupes = [2, 1, 1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 2]
```

correspondant au nombre de déplacements pour chaque groupe.

Le cube que l'on va remplir sera modélisé par les 27 points de coordonnées $(x, y, z) \in \{0, 1, 2\}^3$ que l'on nommera positions, à l'aide d'une liste de listes de listes de booléens (Vrai si la case a déjà été remplie, faux sinon).

Les directions dans lesquelles on peut effectuer un déplacement pour remplir le cube seront de la forme $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$ et $(0, 0, \pm 1)$.

2. Écrire une fonction `directions(position)` qui, à partir d'une position, renvoie la liste de toutes les directions dans lesquelles on peut se déplacer en restant dans le cube.

3. Écrire une fonction `avance(position, direction, nb)` qui renvoie la position correspondant à un déplacement de `nb` places dans la `direction` à partir de la `position`.

4. Écrire une fonction `position_valide(position)` qui teste si une position est bien située dans le cube.

5. Écrire une fonction `cube_init(pos_init)` qui renvoie un cube dont toutes les cases sont à `False` sauf la case en position `pos_init`.

6. Écrire une fonction `case(cube, position)` renvoyant la valeur booléenne du `cube` à la `position`.

7. Écrire une fonction `affecte(cube, position, valeur)` donnant la `valeur` dans la `position` du `cube`.

Principe de résolution et fonctions de base

8. Écrire une fonction `direction_valide(position, cube, n, direction)` qui détermine si la `direction` dans le `cube` à partir de la `position` pour le groupe d'indice `n` est valide dans le sens où l'on ne sort pas du cube et où l'on ne rencontre pas de case déjà occupée.

9. Écrire une fonction `liste_directions_valides(position, cube, n)` qui renvoie la liste de toutes les directions valides au sens décrit dans la question précédente.

10. Écrire une fonction `remplir(cube, position, direction, n)` qui remplit effectivement le `cube` à partir de la `position` dans la `direction` avec la pièce d'indice `n`, c'est-à-dire qui va placer `True` dans toutes les cases traversées.

11. Écrire une fonction `retirer(cube, position, direction, n)` qui réalise l'opération réciproque de la précédente.

Résolution

12. Écrire une fonction récursive `resolution(cube, position, pos_init, n=0, L=[])` qui remplit le `cube` à partir de la `position` en y plaçant la pièce d'indice `n`, `L` étant la liste des directions pour les pièces déjà placées résolvant le problème par backtracking : en testant successivement toutes les directions (parcours en profondeur), elle renvoie `True` si la résolution est possible, et `False` s'il n'y a plus de direction possible à partir d'une certaine position (backtracking). À la fin de la résolution, une solution se trouvera dans `L`.

13. Déterminer des solutions à partir de toutes les positions initiales possibles.

14. En déduire un affichage clair de toutes les solutions (point de départ + directions). On pourra utiliser des dictionnaires :

```
correspondance1 = {(-1,0,0) : "à gauche", ...  }

correspondance2 = {(-1,0,0) : "L", ...  }
```

15. Proposer une version permettant d'obtenir toutes les solutions du puzzle.

Aide pour la fonction `resolution` :

- Si `n` ne correspond plus à un indice de pièce, c'est terminé, on a résolu le puzzle : on renvoie `True`, la solution se trouve dans `L`.
- Sinon, déterminer la liste des directions possibles à cette position pour cette pièce. Pour chaque direction,
  - ⋆ Remplir le cube avec la pièce en cours et la direction en cours.
  - ⋆ Rappeler la fonction pour placer la pièce suivante. Si elle renvoie `True`, on renvoie `True`.
  - ⋆ Sinon, retirer la pièce avant de passer à la direction suivante.
- Si toutes les directions sont épuisées, renvoyer `False`. (Backtracking)