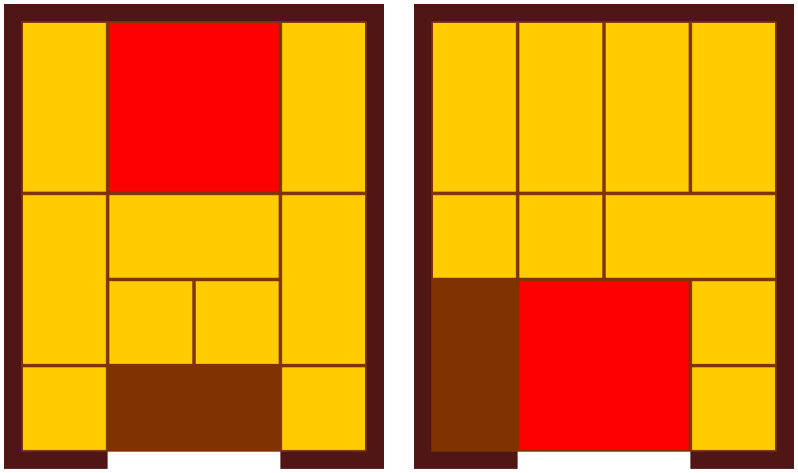
Le problème de l'âne rouge

L'Âne rouge est un puzzle à pièces coulissantes ou casse-tête de déplacements proche du taquin[1]. Il est probablement d'origine thaïlandaise et on le retrouve aujourd'hui sous de nombreux noms (Klotski, etc). Le but est de faire sortir une tuile carrée 2 × 2 appelée *âne rouge* (en blanc sur la photo) par une fente située en bas du jeu sur la photo.

Le jeu démarre comme ci-dessous et on cherche à atteindre une position dans laquelle l'âne rouge est situé au centre et en bas, ce qui permettra de sortir cette tuile du jeu.

[1] d'après Wikipedia



debut une des positions finales

1 Modélisation du jeu

On représente le jeu par une liste de couples $(piece, position)$ où *pièce* est le type de pièce et *position* est un couple (i, j) représentant les coordonnées du bord supérieur gauche de la pièce dans la grille 5×4 du jeu. La ligne $i = 0$ sera la ligne du haut et la colonne $j = 0$ sera la colonne de gauche. On introduit les types suivants :

```
type piece = Ane | Recth | Rectv | Carre;;

type coord = int * int;;

type config = (piece * coord) list;;
```

Puisque l'ordre des pièces n'a pas d'importance dans la liste, deux mêmes configurations peuvent être représentées par deux listes différentes. Cela est embêtant pour réaliser des tests d'égalité c'est pourquoi je vous fourni une fonction `tri_config` qui permet de trier ces listes et d'avoir une unique liste associée à chaque configuration. Vous vous efforcerez à ne conserver que des versions triées des configurations que vous construirez.

Question 1

Écrire un code `OCaml` permettant de construire la valeur initiale `config_init` représentant la configuration initiale (triée) du jeu.

Une fonction `print_config` est fournie pour vérifier vos résultats !

Question 2

Écrire une fonction `config_finale : config → bool` testant si une configuration (triée ou non) est finale.

Nous voulons maintenant tester si une `configuration` est valide. Une configuration valide vérifie les deux points suivants :

1. Les coordonnées de chaque tuile sont valides (les tuiles ne dépassent pas en dehors de la grille 5×4.
2. Les tuiles ne se chevauchent pas.

Nous utiliserons la méthode suivante pour vérifier qu'une configuration est valide :

- On utilise une matrice `m` 5×4 de booléens pour coder si une case est déjà occupée ou non.
- On parcourt chaque pièce et on vérifie ses cooronnées
- De plus, on marque dans la matrice `m` les cases occupées : si une des cases était déjà occupée, on déclenche l'exception `Configuration_invalide`.

Question 3

Compléter la fonction auxiliaire `marquer` qui vérifie si une case est libre et si elle l'est, la marque comme occupée. Si elle n'est pas libre elle déclenche l'exception.

Question 4

En vous inspirant des parties déjà écrites, compléter la fonction `config_valide`.

2 Graphe du jeu

Nous allons maintenant considérer le graphe du jeu ayant les configrations pour sommets, et dans lequel les arcs représentent les déplacements de tuiles possibles.

Question 5

Écrire la fonction `move : coord → direction → coord` calculant les coordonnées d'arrivée lors qu'on se déplace depuis une case de départ dans la direction donnée.

La fonction suivante vous est fournie :

```
val voisins : config -> config list
```

Question 6

1. Déterminer le degré du sommet initial du jeu (avec un programme évidemment)
2. Afficher à l'écran tous les voisins de la position de départ.

3 Résolution par parcours en largeur

Nous allons maintenant résoudre le jeu à l'aide d'un parcours en largeur. Pour cela on adoptera les solutions techniques suivantes :

- On utilisera une `file` de la bibiothèque Caml (module `Queue`)
- L'ensemble des sommets *ouverts* sera codé à l'aide d'une table de hachage où les clefs sont les configurations et les valeurs `()`.
- De même pour les sommets *fermés*.
- L'arborescence du parcours sera aussi codée à l'aide d'une table de hachage `parent` où on associe à chaque configuration (clef) son prédécesseur (valeur) dans l'arborescence du parcours.

Question 7

Écrire la fonction `parcours_bfs : config -> unit`. Cette fonction effectue un parcours en largeur depuis la configuration donnée. Si le parcours rencontre une configuration finale alors on déclenche une exception `Solution` qui contient la configuration finale et l'arborescence du parcours.

Question 8

Écrire une fonction `reconstruction : config -> (config, config) Hashtbl` qui reconstruit le chemin solution étant donné la configuration finale et l'arborescence du parcours depuis la configuration initiale.

Question 9

1. Afficher une solution, au jeu de l'âne rouge.
2. Quel est le nombre minimal de déplacements à effectuer pour résoudre le jeu ?