TP n°16 - Retour sur trace

1. Problème des n reines

On rappelle que le problème des n reines qui, pour un entier $n > 0$ donné, revient à placer n reines sur un échiquier de $n \times n$ cases sans que les reines ne puissent s'attaquer mutuellement, conformément aux règles de déplacement de cette pièce aux échecs.

On représente les configurations par un tableau `config` de taille n. Pour tout i, `config`$[i]$ est la colonne où on place la reine sur la ligne i. Une configuration `config` est solution si :

Pour tous $i, j \in \{0, \dots, n-1\}$,

(1) `i != j` $\Rightarrow$ `config.(i) != config.(j)`,
(2) `i != j` $\Rightarrow$ `config.(i) + j != config.(j) + i`, et
(3) `i != j` $\Rightarrow$ `config.(i) - j != config.(j) - i`.

1. Que testent les trois conditions pour i et j fixés ?

On définit une configuration partielle comme étant une configuration où seule les reines des i premières lignes ont été placées. Dans ce cas, on ignore les valeurs du tableau `config` d'indice supérieur à i. Une solution partielle est une configuration partielle qui vérifie les conditions (1), (2) et (3).

2. Écrire une fonction `conflit : int -> int -> int -> int -> bool` qui teste si deux reines peuvent s'attaquer basé sur leurs lignes et colonnes.
3. Écrire une fonction `est_solution : int array -> bool` qui teste si une configuration est une solution totale.

Pour pouvoir utiliser un algorithme exhaustif, il faut pouvoir énumérer les possibilités. On considère que [| 0 ;0 ;... ;0 ;0 |] est la première configuration, [| 0 ;0 ;... ;0 ;1 |] la deuxième, [| 0 ;0 ;... ;0 ;n-1 |] la n-ième, la n+1-ième est [| 0 ;0 ;... ;1 ;0 |], la n+2-ième [| 0 ;0 ;... ;1 ;1 |], la 2n+1-ième [| 0 ;0 ;... ;2 ;0 |], et [| n-1 ;n-1 ;... ;n-1 ;n-1 |] est la dernière.

4. Écrire une fonction `suivant : int array -> bool` qui prend en entrée une configuration et modifie le tableau pour qu'il contienne la configuration suivante. La fonction renvoie `true` si la configuration en entrée est la dernière et `false` sinon.
5. En énumérant toutes les configurations (recherche exhaustive), écrire une fonction `nb_solutions : int -> int` qui compte le nombre de solutions au problème des n reines.
   On ne dépassera pas $n = 10$ dans les tests pour le bien des ordinateurs.

On va maintenant utiliser le retour sur trace pour trouver plus vite. D'une part on voudrait obtenir une solution, et de l'autre, on veut compter le nombre de solutions, pour comparer la vitesse avec la question précédente.

6. Écrire une fonction `est_valide : int -> int array -> bool` qui dit si la configuration partielle où les i premières reines ont été placées dans le tableau est une solution partielle.
7. En utilisant le principe du retour sur trace et la fonction `est_valide`, écrire une fonction `une_solution : int -> int array` qui renvoie la première solution qu'elle trouve.
   Une aide est disponible pour cette question sous forme d'un code à remplir.
8. Adapter la fonction précédente en une fonction `nb_solutions2 : int -> int` qui compte le nombre de solutions au problème des n reines, toujours en suivant le principe du retour sur trace. Remarque : on a plus besoin d'une exception.
   Vous pouvez cherche pour quelle valeur de n on obtient la réponse en un temps raisonnable (moins d'une minute) et comparer avec la question 5.

2. Permutations

Le but de cette partie est de savoir générer les permutations de $[|0, n-1|]$.

Une permutation d'un ensemble S est une manière d'ordonner les éléments de S. Pour $S = [|0, 2|]$, les permutations sont $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 1, 0)$ et $(2, 0, 1)$. Pour $S = [|0, 5|]$, $(0, 1, 2, 3, 4, 5)$ est une permutation, tout comme $(2, 3, 5, 1, 0, 4)$ ou $(1, 2, 4, 5, 3, 0)$.

On va représenter les permutations en Ocaml par des listes (pour faciliter la création, si on voulait faciliter l'utilisation, on utiliserait des tableaux). Notre algorithme de création sera récursif.

On commence par écrire quelques fonctions utilitaires.

9. Écrire une fonction `remplace : int -> int -> int list -> int list` qui prend en entrée un nombre a, un nombre b et une liste l, et remplace chaque occurence de b par a dans la liste l.

10. Écrire une fonction `remplace_iter : int -> int -> int list list -> int list list` qui prend en entrée, a, b et une liste de listes ll et renvoie une liste de listes où chaque liste de ll a eu ses b remplacés par des a.
11. Écrire une fonction `tete_iter : int -> int list list -> int list list` qui prend en entrée un nombre c et une liste de listes ll et rajoute un c en tête de chaque liste contenue par ll.

Maintenant, on remarque que pour faire une permutation s de $[|0, n-1|]$ on peut choisir un premier élément c dans $[|0, n-1|]$ puis faire une permutation s' de $[|0, n-2|]$ où, si $c \neq n-1$, on a remplacé c par $n-1$ dans s'. s est alors $c :: s'$.

Pour créer toutes les permutations de $[|0, n-1|]$, on doit répéter ce processus avec tous les c et tous les s' possibles. Il est avantageux de ne calculer qu'une seule fois les permutations de $[|0, n-2|]$ (gain exponentiel de temps)

12. Écrire une fonction `permutations : int -> int list list` qui prend en entrée n et renvoie la liste des permutations de $[|0, n-1|]$.