Nonogrammes

1 Préliminaires

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `nonogrammes` contenant une source OCaml :

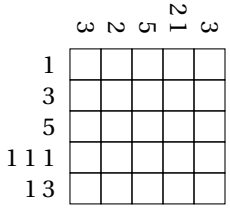
```
curl cdn.sci-phy.org/mp2i/tp18-nonogrammes.tgz | tar xvz
```

Le fichier à compléter est le fichier `nonogrammes.ml`. On y trouvera quelques fonctions utiles qui seront présentées au fur et à mesure de nos besoins, et des problèmes qui permettront de tester les fonctions écrites.

1.2 Présentation du problème

Les *nonogrammes*[1] sont des « puzzles » créés indépendamment par Non Ishida, une graphiste japonaise, et Tetsuya Nishio, un créateur de puzzles japonais. Le but est de noircir un certain nombre de cases dans une grille en respectant certaines contraintes, généralement pour reconstituer une image.

Initialement, le problème se présente sous d'une grille vide, assortie d'indices en regard de chacune des lignes et colonnes, comme ci-dessous :

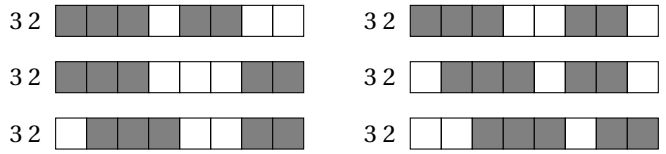


L'objectif du puzzle est de noircir une partie des cases de la grille en respectant certaines règles. Pour chaque ligne (ou colonne) de la grille, les indices indiqués correspondent au nombre de cases noires *consécutives* que l'on trouvera dans la ligne (ou colonne), chaque série de cases noires étant séparée des autres par une ou plusieurs cases blanches. Par exemple, la ligne

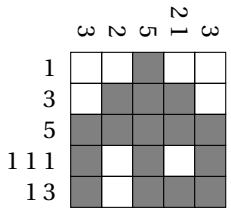
comprend un bloc noir de longueur 3 suivi d'un bloc de longueur 2, et sera annotée « 3 2 ».

1. Ce nom a été proposé par le créateur de puzzles anglais James Dalgety en référence au prénom de Ishida Non, mais il en existe de nombreux, dont certains commerciaux : Panjie, Paint by Numbers, Picross, Griddlers...

Ce même indice « 3 2 », pour une ligne de huit cases, peut en fait correspondre à six situations possibles :



Les contraintes indiquées consuisent, en principe, à une unique solution, et il est aisé de vérifier que c'est le cas pour l'exemple proposé, la solution étant la suivante :



Le but de cette séance de travaux pratiques sera de déterminer la solution d'un problème, et de vérifier si elle est bien unique.

2 Problèmes à une dimension

Dans un premier temps, nous allons nous pencher sur une seule ligne (ou colonne) du problème, et étudier les solutions possibles.

Pour ce faire, on considère deux entiers positifs ou nuls s et n, et on souhaite déterminer l'ensemble des listes de $n \geq 1$ entiers positifs ou nuls dont la somme est s. Par exemple, les listes de trois entiers positifs ou nuls dont la somme est 3 sont `[3; 0; 0]`, `[2; 1; 0]`, `[2; 0; 1]`, `[1; 2; 0]`, `[1; 1; 1]`, `[1; 0; 2]`, `[0; 3; 0]`, `[0; 2; 1]`, `[0; 1; 2]` et `[0; 0; 3]`.

1. écrire une fonction `list_sols` prenant en argument deux entiers `s` et `n` et retournant une liste contenant l'ensemble des listes de `n` entiers positifs dont la somme est `s`.

Trouver l'ensemble des solutions possibles pour le placement des blocs noirs dans les n cases constituant la ligne (ou colonne) revient à se poser la question du placement des cases blanches, dont on connaît le nombre. S'il y a p blocs noirs, alors les cases blanches séparant ces blocs noirs doivent être réparties en $p+1$ groupes.

Le premier groupe peut ne contenir aucune case blanche (si le premier bloc noir se trouve au tout début), le dernier également (si le dernier bloc noir se trouve à la fin) mais

chacun des autres groupes doit contenir au moins une case blanche.

Par exemple, dans le cas d'un groupe de longueur 3 suivi d'un groupe de longueur 2 à placer dans un espace de huit cases, les solutions présentées ci-contre correspondent aux répartitions suivantes des cases blanches : `[0; 1; 2]`, `[0; 2; 1]`, `[0; 3; 0]`, `[1; 1; 1]`, `[1; 2; 0]` et `[2; 1; 0]`.

2. Proposer une fonction `white_blocks` prenant en argument une liste d'entiers positifs représentant les indices, et la taille `n` de la ligne (ou de la colonne), et qui retourne la liste des tailles possibles pour chacun des groupes des cases blanches, comme ci-dessus.

On pourra par exemple appeler la fonction `list_sols` avec des paramètres bien choisi, puis regarder les solutions retournées une par une et construire une seconde liste avec les solutions qui peuvent être retenues. Cependant, générer des solutions qui seront éliminées ensuite est du gâchis. Vous pouvez réfléchir à une meilleure solution.

3. Vérifier que « `white_blocks [3; 2] 8` » fournit bien l'ensemble des six listes précédentes (à l'ordre près).

4. Construire une fonction `build` prenant deux listes d'entiers, représentant respectivement les tailles des groupements blancs et celle des groupements noirs, et retournant un *tableau* de 0 et de 1 correspondant à la succession de groupes blancs et noirs décrits par les deux listes. Par exemple :

```
# build [1; 2; 0] [3; 2];;

- : int array = [|0; 1; 1; 1; 0; 0; 1; 1|]
```

5. En déduire une fonction `solutions` prenant en argument une liste d'indices et la taille `n` de la liste (ou colonne) et retournant la liste des solutions possibles pour la ligne considérée. Ainsi, « `solutions [3; 2] 8` » doit retourner la liste des six solutions possibles présentées tantôt, représentées sous la forme de tableaux de taille huit contenant des 0 et des 1, soit :

```
# solutions [3; 2] 8;;

- : int array list = [[|1; 1; 1; 0; 1; 1; 0; 0|];
[|1; 1; 1; 0; 0; 1; 1; 0|]; [|1; 1; 1; 0; 0; 0; 1; 1|];
[|0; 1; 1; 1; 0; 1; 1; 0|]; [|0; 1; 1; 1; 0; 0; 1; 1|];
[|0; 0; 1; 1; 1; 0; 1; 1|]]
```

On souhaite à présent aller un peu plus loin, et étudier s'il est possible d'avoir des certitudes sur certaines des cases. On notera par la valeur `-1` les cases dont l'état est incertain, et on conservera les valeurs `0` et `1` pour les cases dont on est sûr qu'elles sont respectivement blanches et noires.

Dans l'exemple qui nous occupe, on peut voir que dans les six solutions, la troisième case est toujours noire. En revanche, chacune des sept autres cases peuvent prendre l'une ou l'autre des deux couleurs.

Autrement dit, pour le problème `[3, 2]` dans huit cases, on notera le résultat sous la forme `[|-1; -1; 1; -1; -1; -1; -1; -1|]`.

6. Proposer une fonction `result` prenant en argument les indices et la taille de la ligne (ou colonne) et retournant un tableau à n éléments parmi `0`, `1` et `-1` indiquant ce que l'on a pu déduire de la solution.

Il arrive que l'on ait préalablement des informations sur la solution. Par exemple, on peut savoir que la première case est blanche, et que la première case est noire. On écrira ces indices sous la forme d'un tableau de n éléments parmi `0` (case blanche), `1` (case noire) et `-1` (pas d'information). Par exemple, dans le cas présent, les indices correspondent à `[|0; -1; -1; -1; -1; -1; -1; 1|]`.

7. Proposer une fonction `result_with_data` prenant en argument la liste des indices et un tableau regroupant les informations dont on dispose déjà, et qui retourne un tableau de même taille correspondant à ce que l'on a pu déterminer sur la ligne en question.

Attention, cette dernière fonction ne doit pas faire appel à la fonction `result` ! En effet, il est préférable de faire appel à `solutions`, et de ne conserver que les solutions compatibles avec les informations fournies, avant d'en déduire le résultat.

On devrait obtenir le comportemet suivant :

```
# result_with_data [3; 2] [|0; -1; -1; -1; -1; -1; -1; 1|];;

- : int array = [|0; -1; 1; 1; -1; 0; 1; 1|]
```

La solution se rapproche !

3 Problèmes à deux dimensions

Le problème à une seule dimension est très vite limité. C'est l'intéraction des informations verticales et horizontales qui va permettre de résoudre un nonogramme.

Dans la suite, on représentera l'état courant d'un nonogramme par un tableau à deux dimensions (`int array array`), pouvant contenir les valeurs `0` (case blanche), `1` (case noire) et `-1` (case indéterminée).

Le problème est défini par la donnée d'un couple de deux listes de listes (`int list list * int list list`), chaque liste indiquant, respectivement pour chaque ligne et chaque colonne, la longueur des différents blocs noirs.

Dans le fichier, on dispose de plusieurs problèmes appelés `puzzle1` à `puzzle9`. Les problèmes 1 à 6 sont faciles à résoudre, le problème 7 est plus difficile et ne peut pas être résolu avec l'approche proposée dans cette section, mais pourra l'être avec la section

suivante. Le problème 8 est un exemple de problème qui possède plusieurs solutions. Le problème 9, enfin, est plus difficile à résoudre.

Pour afficher le résultat à l'écran, on peut utiliser la fonction `affiche` qui prend en argument un tableau d'entiers et en dessine le contenu.

Il vous est fourni quatre fonctions, `get_line` et `get_col` de signature `int -> int array array -> int array` prenant en argument un index et un tableau à deux dimensions et retournant un tableau contenant une copie de la ligne ou de la colonne demandée, et deux fonctions `set_line` et `set_col` de signature `int -> int array -> int array array -> unit` prenant en argument un index, un tableau à une dimension représentant une ligne ou une colonne et un tableau à deux dimensions et copiant le contenu du premier tableau dans le second à l'endroit demandé.

8. Proposer une fonction `solve_lines` prenant en argument un puzzle et un tableau, et essayant de résoudre chacune des lignes du tableau avec les données contenues dans le tableau et les indices, et rangeant le résultat obtenu dans le tableau.

9. Faire de même avec une fonction `solve_cols` opérant cette fois sur les colonnes de la grille.

10. Proposer une fonction `count` prenant en argument une grille (soit un tableau à deux dimensions) et retournant le nombre de cases indéterminées restant dans la grille fournie en argument.

11. En déduire une fonction `solve_1` de signature `int list list * int list list -> int array array -> unit` qui prend en argument un puzzle et une grille et ne retourne rien, mais tente de résoudre le puzzle en appelant alternativement `solve_lines` et `solve_cols` tant que le nombre de cases inconnues diminue. Ce faisant, elle modifie la grille fournie en second paramètre.

Pour tester la fonction précédente, on dispose d'une fonction `init_grid` prenant en argument un puzzle et retourant une grille de la bonne taille, où toutes les cases correspondent à une case indéterminée. On pourra donc écrire, pour un puzzle donné :

```
let grid = init_grid puzzle1;; (* création de la grille *)

solve_1 puzzle1 grid;;       (* tentative de résolution *)

draw grid;;             (* affichage du résultat obtenu *)
```

Pour de nombreuses grilles, cette approche suffit à obtenir la solution. On la testera avec les premiers problèmes.

4 Grilles difficiles

Pour certains problèmes plus difficiles, la démarche précédente peut ne plus être suffisante (le nombre de cases inconnues ne diminue plus même s'il en reste). Il faut alors utiliser une approche de type retour sur trace.

12. Proposer une fonction `find_unknown` prenant en argument une grille et retournant un couple de coordonnées correspondant à une case encore inconnue.

Dans le fichier fourni, on a défini une exception `Impossible` avec

```
exception Impossible;;
```

13. Modifier la fonction `result_with_data` pour qu'elle lève l'exception `Impossible` s'il n'existe aucune solution compatible avec les informations fournies.

14. Modifier la fonction `solve_1` pour qu'elle retourne un booléen indiquant si aucune solution n'est possible (`false`) ou si elle n'a pas trouvé de conflit (`true`). Un appel à `solve_1` ne doit pas laisser d'exception non traitée.

Note : un résultat `true` retournée par la fonction précédente ne signifie pas nécessairement que la grille peut être résolue

15. Proposer une fonction `solve` de signature `int list list * int list list -> int array array` prenant en argument un problème, génère une grille vide, et utilise le principe du retour sur trace pour résoudre un problème que `solve_1` ne parvient pas à résoudre directement. On fournit pour ce faire une fonction `copy_grid` permettant d'obtenir une copie (profonde) d'un tableau à deux dimensions (pourquoi est-ce utile ici ?).

Pour des grilles vraiment difficiles, l'approche précédente peut ne pas être suffisante. Envisager des améliorations possibles pour la résolution.