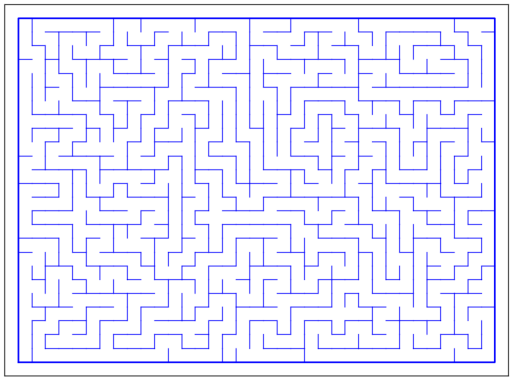


Un **labyrinthe parfait** est un un labyrinthe ayant la propriété suivante : pour toutes positions s et t dans le labyrinthe, il existe un unique chemin allant de s à t.

L'objectif de ce TP est d'écrire un algorithme générant des labyrinthes parfaits de taille $n \times n$, pour $n \in \mathbb{N}^\star$. Pour cela, nous pouvons considérer le graphe où les sommets sont les positions du labyrinthe, et où chaque position $(i, j) \in [\![0, n-1]\!]^2$ est reliée aux positions adjacentes $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$ si elles existent.

Un fichier `TP11_eleve.ml` est disponible sur la page web du cours, et contient une implémentation d'une structure Union-Find utile pour la fin du TP ainsi qu'une fonction d'affichage de labyrinthe.

1 Étude préliminaire

Supposons qu'au début de l'algorithme, il y a des murs entre toutes les cases du labyrinthe.

1. Considérons un **arbre couvrant** de ce graphe.
   Comment supprimer des murs (en utilisant cet arbre couvrant) afin d'obtenir un labyrinthe parfait ?
2. En déduire un algorithme permettant de générer un labyrinthe parfait (le décrire en quelques lignes).

2 Mélange équitable

On peut donc implémenter une version simplifiée de l'algorithme de Kruskal pour calculer un arbre couvrant : on aura toujours besoin d'une structure Union-Find, mais pas besoin de file de priorité car ici le graphe considéré n'est pas pondéré.

Selon l'ordre dans lequel on va considérer les arêtes, on obtiendra un arbre couvrant différent (et donc un labyrinthe parfait différent). On s'intéresse alors à trouver un algorithme générant aléatoirement un ordre de considération des arêtes. Plus précisément, on voudrait un algorithme de **mélange équitable**.

Définition 1 (mélange équitable)
Un ***mélange équitable*** *est une fonction* `melange : 'a array -> unit` *telle que :*

- `melange` *est un algorithme probabiliste ;*
- `melange` *termine toujours avec la même complexité ;*
- `melange tab` *modifie le tableau* `tab` *pris en argument par effets de bords en effectuant des échanges entre ses cases (à la fin de l'algorithme,* `tab` *contient les mêmes éléments qu'au départ mais dans un ordre différent) ;*
- *une fois l'appel à* `melange tab` *terminé, la probabilité que l'élément initialement dans la case* `tab`$.(i)$ *se retrouve au final dans la case* `tab`$.(\sigma(i))$ *est* $\frac{1}{n!}$ *(où n est la taille de* `tab`*), pour toute permutation* $\sigma \in \mathfrak{S}_{[\![0,n-1]\!]}$.

3. Une telle fonction `melange` est-elle un algorithme de type Monte Carlo ou Las Vegas ?

On s'intéresse au mélange suivant, appelé **mélange de Knuth** :

Algorithme 1 : mélange de Knuth

Données : un tableau T
Résultat : une permutation des éléments de T (T est modifié par effets de bords)
$n \leftarrow$ longueur(T) ;
pour i *allant de* 1 *à* $n-1$ **faire**
  $j \leftarrow$ un élément de $[\![0, i]\!]$ choisi aléatoirement (avec probabilité uniforme) ;
  échanger $T[i]$ et $T[j]$;

On souhaite montrer que ce mélange est équitable. On introduit les notations suivantes :

- pour $\ell \in [\![0, n-1]\!]$, x_ℓ est la valeur initiale de $T[\ell]$;
- pour $i \in [\![0, n-1]\!]$, pour $k \in [\![0, i]\!]$, y_k^i est la valeur contenue dans $T[k]$ à la fin de l'étape i ;
- pour $i \in [\![0, n-1]\!]$, pour $\sigma \in \mathfrak{S}_{[\![0,i]\!]}$, $\mathbb{P}(i, \sigma)$ est la probabilité que $y_k^i = x_{\sigma(k)}$ pour $k \in [\![0, i]\!]$.

4. Montrer que la propriété suivante par récurrence sur i : $\forall \sigma \in \mathfrak{S}_{[\![0,i]\!]},\ \mathbb{P}(i, \sigma) = \frac{1}{(i+1)!}$.
5. En déduire que le mélange de Knuth est un mélange équitable.
6. Écrire une fonction `melange_knuth : 'a array -> unit` implémentant cet algorithme.

Exemple

```
# let tab = Array.init 10 (fun i -> i) in
melange_knuth tab ;
tab ;; (* le résultat obtenu change à chaque exécution *)
- : int array = [|4; 3; 0; 5; 8; 7; 9; 1; 6; 2|]
```

3 Générateur de labyrinthes parfaits

Soit $n \in \mathbb{N}^\star$. On considère une grille de taille $n \times n$, avec initialement des murs partout. Pour une case d'une telle grille, on définit deux choses :

- les **coordonnées** $(i, j) \in [\![0, n-1]\!]^2$ de la case : i est le numéro de la ligne (en partant du haut), et j est le numéro de la colonne (en partant de la gauche) (la case $(0, 0)$ est en haut à gauche) ;
- le **numéro** k de la case : on numérote les cases de 0 à $n^2 - 1$, ligne par ligne, de haut en bas (si (i, j) sont les coordonnées de la case, on a donc $k = i \times n + j$).

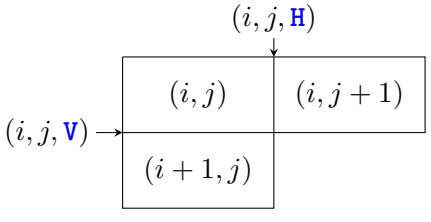
On souhaite représenter les arêtes du graphe associé à la grille (qui pour rappel correspond aux murs internes potentiels du labyrinthe). Pour cela, on introduit les types suivants :

types direction et arete

```
type direction = H | V ;; (* horizontal ou vertical *)

type arete = int * int * direction ;;
```

- si $i \in [\![0, n-1]\!]$ et $j \in [\![0, n-2]\!]$, $(i, j, \texttt{H})$ représente l'arête horizontale entre (i, j) et $(i, j+1)$;
- si $i \in [\![0, n-2]\!]$ et $j \in [\![0, n-1]\!]$, $(i, j, \texttt{V})$ représente l'arête verticale entre (i, j) et $(i+1, j)$.



7. Combien y a t-il de murs internes potentiels dans une grille de taille $n \times n$?

8. Écrire une fonction `generer_aretes : int -> arete array` prenant en argument un entier n et renvoyant un tableau contenant toutes les arêtes du graphe considéré pour une grille de taille $n \times n$.

Exemple

```
generer_aretes 3 ;;
- : arete array =
[|(0, 0, V); (0, 0, H); (0, 1, V); (0, 1, H); (0, 2, V); (1, 0, V); (1, 0, H);
  (1, 1, V); (1, 1, H); (1, 2, V); (2, 0, H); (2, 1, H)|]
```

Pour finir, on utilise une structure Union-Find pour implémenter une version simplifiée de l'algorithme de Kruskal et générer des labyrinthes parfaits : lors de cet algorithme, conserver une arête dans l'arbre couvrant revient à supprimer le mur correspondant dans le labyrinthe. Pour cela, on introduit deux matrices `horiz` et `vert` de taille $n \times (n-1)$ et $(n-1) \times n$ respectivement telles que :

- si $i \in [\![0, n-1]\!]$ et $j \in [\![0, n-2]\!]$, `horiz.(i).(j)` contient `true` si l'arête $(i, j, \texttt{H})$ est bloquée par un mur (et `false` sinon) ;
- si $i \in [\![0, n-2]\!]$ et $j \in [\![0, n-1]\!]$, `vert.(i).(j)` contient `true` si l'arête $(i, j, \texttt{V})$ est bloquée par un mur (et `false` sinon).

Au départ, toutes les cases de ces matrices sont initialisées à `true`, et on génère un mélange des arêtes du graphe associé. Puis on parcourt toutes les arêtes dans l'ordre obtenu et on applique l'algorithme de Kruskal (lors duquel on va casser les murs correspondant aux arêtes de l'arbre couvrant qu'on est en train de calculer).

9. Écrire une fonction `labyrinthe_parfait : bool array array * bool array array` prenant en argument un entier n et générant un labyrinthe parfait selon le principe décrit ci-dessus ; on renverra les deux tableaux `horiz` et `vert`.

Exemple

```
# labyrinthe_parfait 3 ;; (* le résultat obtenu change à chaque exécution *)
- : bool array array * bool array array =
([|[|true; true|]; [|true; false|]; [|false; true|]|],
 [|[|false; false; false|]; [|false; false; false|]|])
```

10. Tester votre fonction à l'aide de la fonction `afficher` fournie.

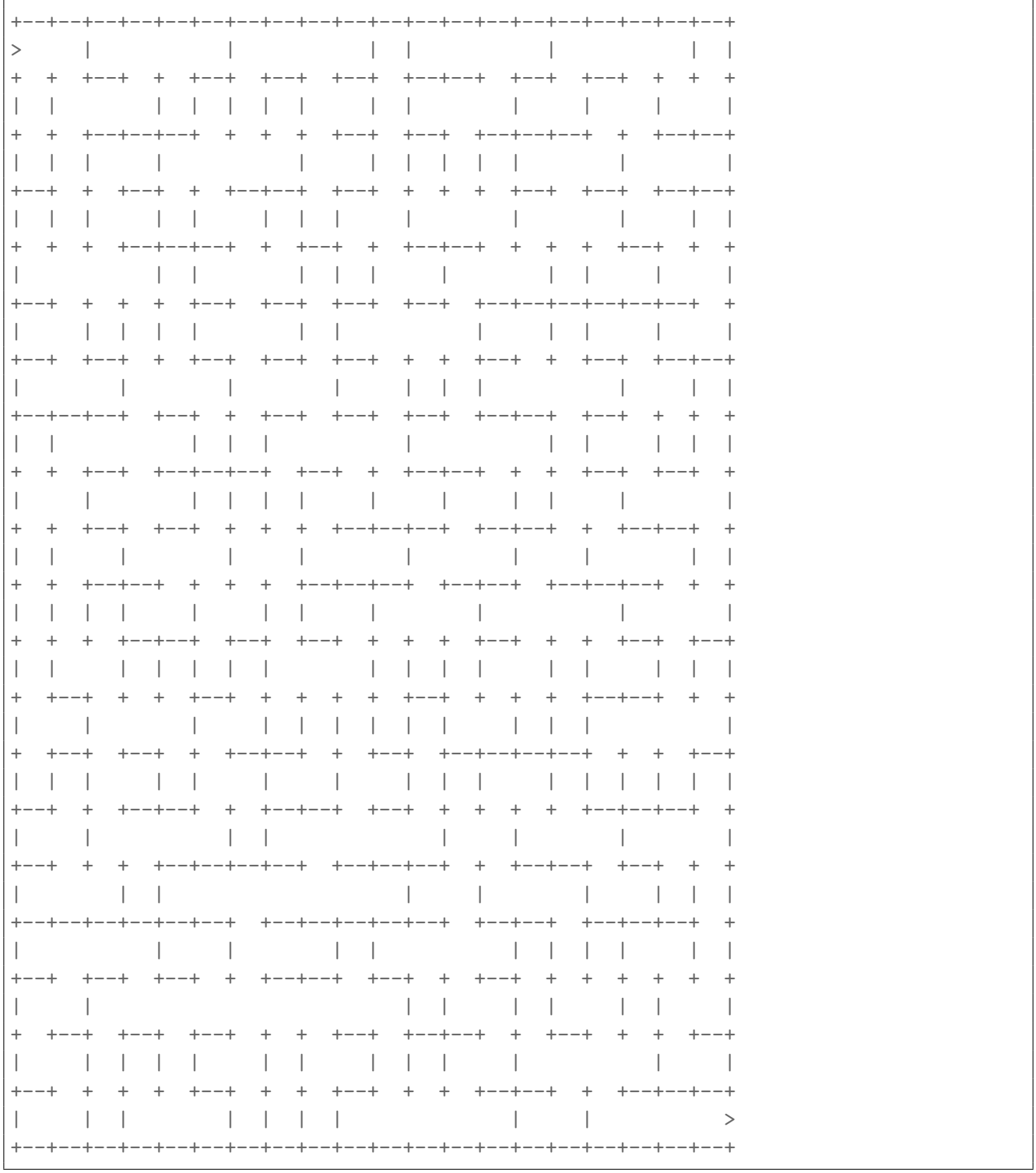
Exemple : taille 10

```
# let (horiz, vert) = labyrinthe_parfait 10 in afficher horiz vert ;;

+--+--+--+--+--+--+--+--+--+--+
>        |     |     |  |  |  |
+--+  +--+  +--+  +  +  +  +  +
|  |     |        |           |
+  +  +  +  +--+--+  +--+  +--+
|     |        |        |     |
+  +  +--+--+  +--+--+--+--+--+
|  |  |  |              |  |  |
+--+--+  +  +--+--+--+--+  +  +
|  |  |                    |  |
+  +  +--+--+--+--+  +--+--+  +
|     |  |                    |
+--+  +  +  +--+--+--+--+--+  +
|  |     |  |                 |
+  +--+  +--+  +  +  +--+  +  +
|              |  |     |  |  |
+--+  +  +--+  +  +--+  +  +--+
|  |  |     |  |  |     |  |  |
+  +  +  +--+  +--+--+--+  +  +
|     |     |        |        >
+--+--+--+--+--+--+--+--+--+--+
```

Exemple : taille 20

```
# let (horiz, vert) = labyrinthe_parfait 20 in afficher horiz vert ;;
```



11. Résoudre les deux labyrinthes ci-dessus à la main, à l'aide d'un parcours en profondeur ; l'entrée et la sortie sont représentées par un >, en haut à gauche et en bas à droite respectivement.