Initiation au développement

TD6bis : Tableaux - Labyrinthe

Tableau de tableaux

On peut représenter des objets comme des matrices ou des tableaux à deux dimensions (d'un point de vue programmation et informatique) par des tableaux de tableaux. Dans les faits, il s'agit d'un tableau dont les éléments sont eux-mêmes des tableaux. Ces derniers tableaux contiennent les éléments que l'on veut stocker. Prenons l'exemple de la matrice d'entiers ci-dessous :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

elle peut être représenter de la manière suivante :

```
let matrice = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];
```

On peut accéder à chacune des lignes par son indice :

```
console.log("Ligne 0 =", matrice[0]);
console.log("Ligne 0 =", matrice[1]);
console.log("Ligne 0 =", matrice[2]);
```

```
Ligne 0 = [ 1, 2, 3 ]
Ligne 0 = [ 4, 5, 6 ]
Ligne 0 = [ 7, 8, 9 ]
```

Une fois que l'on a la ligne on peut accéder aux éléments sur les colonnes par leurs indices dans le tableau représentant la ligne :

```
console.log("Ligne 0 Colonne 0 =", matrice[0][0]);
console.log("Ligne 1 Colonne 2 =", matrice[1][2]);
```

```

```

```
Ligne 0 Colonne 0 = 1
Ligne 1 Colonne 2 = 6
```

Ci-dessous voici le code que l'on peut utiliser pour afficher sur le terminal les éléments de la matrice en faisant apparaître la structure de la matrice :

```
for (let ligne of matrice) {
  for (let valeur of ligne) process.stdout.write(valeur + " ");
  console.log();
}
```

```
1 2 3
4 5 6
7 8 9
```

ou de manière équivalent si on a besoin de connaître les indices :

```
for (let i in matrice) {
  for (let j in matrice[i]) process.stdout.write(matrice[i][j] + " ");
  console.log();
}
```

Rappel : dans ce cas, `i` et `j` sont des `string` que l'on peut si besoin transformer en nombre : `Number(i)` et `Number(j)`.

Enfin, on pourrait utiliser une `for` standard :

```
for (let i = 0; i < matrice.length; i++) {
  for (let j = 0; j < matrice[i].length; j++) process.stdout.write(matrice[i][j] + "
");
  console.log();
}
```

On peut construire un tableau de tableaux avec des boucles et la méthode `push`. Par exemple, on peut construire la matrice ci-dessus de la manière suivante :

```
let matrice = new Array<Array<number>>(); // On commence par une matrice vide

// On construit les lignes une à une
for (let i = 0; i < 3; i++) {
  // On commence avec une ligne vide
  let ligne = new Array<number>();
```

```
    // on construit la ligne
    for (let j = 0; j < 3; j++) {
      ligne.push(i * 3 + j + 1);
    }

    // On ajoute la ligne à la matrice
    matrice.push(ligne);
  }
```

Représentation d'un labyrinthe

Un labyrinthe est défini par une grille rectangulaire dont les cases peuvent être pleines (mur) ou vide (couloir). La case de coordonnée (0, 0) représente le coin supérieur gauche du labyrinthe. On peut représenter un labyrinthe par un tableau à deux dimensions de booléens. Si une case vaut `true`, il s'agit d'un couloir, s'il la case vaut `false`, il s'agit d'un mur.

Par exemple, le labyrinthe de 4 lignes et 5 colonnes ci-dessous

```
#####
  # #
#
#####
```

est représenté par la liste de listes de booléens :

```
[
  [false, false, false, false, false],
  [true, true, false, true, false],
  [false, true, true, true, true],
  [false, false, false, false, false],
];
```

Initialisation

Écrivez une fonction `init(nbreLignes: number, nbreColonnes: number): Array<Array<boolean>>` qui prend en paramètre le nombre de lignes et le nombre de colonnes et qui retourne une référence sur un tableau de tableaux représentant un labyrinthe initialisé de telle sorte qu'il soit vide sauf sur les bords qui seront des murs.

Test d'une case

Écrivez une fonction `estVide(labyrinthe: Array<Array<boolean>>, i: number, j: number): boolean` qui retourne `true` si et seulement si la case (i, j) du labyrinthe est vide. Écrire aussi une fonction

`estMur(labyrinthe: Array<Array<boolean>>, i: number, j: number): boolean`.

Affichage

Écrivez une fonction `affichage(labyrinthe: Array<Array<boolean>>): void` qui affiche le labyrinthe en mode texte en représentant les cases vides par des espaces (`' '`) et les murs par des `#`. Par exemple, un appel à la fonction pour un labyrinthe de 4 lignes et 6 colonnes construits par la fonction `ìnit` affichera le labyrinthe de la manière suivante :

```
######
#    #
#    #
######
```

Indication : vous pouvez utilisez `process.stdout.write` si vous souhaitez afficher quelque chose sans un retour à la ligne systématique. Vous pouvez aussi construire une chaîne de caractères avant de l'afficher avec `console.log`.

Génération d'un labyrinthe

Les fonctions ci-dessous vont permettre de générer un labyrinthe à partir d'un labyrinthe vide généré par la fonction `ìnit`.

Placement d'îlots

Écrivez une fonction `placeIlots(labyrinthe: Array<Array<boolean>>, nbreIlots: number): void` qui place `nbreIlots` îlots aléatoirement dans le labyrinthe. Un îlot est un mur. Le principe est de choisir aléatoirement les coordonnées des îlots en dehors des "murs extérieurs". Lorsque l'on a choisi une coordonnée, on ne vérifiera s'il y a déjà un mur ou non. Il se peut donc que l'on ait moins que `nbreIlots` îlots placés à la fin.

Remarque : la fonction modifie le labyrinthe passé en paramètre.

Indication : pour choisir les coordonnées, vous pouvez reprendre la fonction `randint` que l'on a déjà utilisée :

```
function randint(inf: number, sup: number): number {
  return Math.floor(Math.random() * (sup - inf + 1) + inf);
}
```

Cases "constructibles"

Pour générer un labyrinthe, on essaie d'étendre les murs déjà existants. On identifie pour cela des cases dites "constructibles". Un case X est constructible si et seulement si elle n'est pas un mur et elle se trouve dans une des situations suivantes.

- Cas 1 :

vide	vide	?
vide	X	mur
vide	vide	?

- Cas 2 :

?	mur	?
vide	X	vide
vide	vide	vide

- Cas 3 :

?	vide	vide
mur	X	vide
?	vide	vide

- Cas 4 :

vide	vide	vide
vide	X	vide
?	mur	?

Question

Écrivez une fonction `estConstructible(labyrinthe: Array<Array<boolean>>, i: number, j: number): boolean` qui retourne `true` si la case (i, j) est constructible, `false` sinon.

Question

Écrivez une fonction `casesConstructibles(labyrinthe: Array<Array<boolean>>): Array<[number, number]>` qui retourne un tableau de tuples où les tuples représentent les cases constructibles du labyrinthe.

Définition : un **tuple** est une structure linéaire comme les tableaux mais dont les éléments peuvent être de types différents. les éléments d'un tuple sont donc indicés comme pour les tableaux et les chaînes de caractères. Les tuples servent à réunir facilement des informations qui vont ensembles comme ici des coordonnées. Dans le cas des coordonnées les deux types sont donc des nombres. Voici un exemple d'utilisation :

```
let tab = new Array<[number, number]>();

tab.push([1, 2]);
tab.push([2, 4]);
tab.push([5, 6]);
```

```
console.log("tab =", tab);
console.log("tab[0] =", tab[0]);
console.log("tab[1] =", tab[1]);
console.log("tab[2] =", tab[2]);
console.log("tab[0][0] =", tab[0][0]);
console.log("tab[0][1] =", tab[0][1]);
console.log("tab[1][0] =", tab[1][0]);
console.log("tab[1][1] =", tab[1][1]);
console.log("tab[2][0] =", tab[2][0]);
console.log("tab[2][1] =", tab[2][1]);
```

```
tab = [ [ 1, 2 ], [ 2, 4 ], [ 5, 6 ] ]
tab[0] = [ 1, 2 ]
tab[1] = [ 2, 4 ]
tab[2] = [ 5, 6 ]
tab[0][0] = 1
tab[0][1] = 2
tab[1][0] = 2
tab[1][1] = 4
tab[2][0] = 5
tab[2][1] = 6
```

Génération

Écrivez une fonction `générerLabyrinthe(nbreLignes: number, nbreColonnes: number, nbreIlots: number): Array<Array<boolean>>` qui génère un labyrinthe de taille `nbreLignes` ×`nbreColonnes` initialisé avec `nbreIlots`. L'algorithme de génération d'un labyrinthe est :

1. Génération d'un labyrinthe vide
2. Placement des îlots
3. Tant qu'il existe des cases constructible, on en choisit une au hasard et on place un mur.