ITERATION 2

Jeu du labyrinthe

Modalités

- Le travail doit être réalisé **seul** sur les étapes de rédaction du code
- Pour les étapes de réflexion et d'écriture du pseudo-code vous pouvez le faire en groupe
- 5 Jours

Objectif Global

Vous êtes nouvellement arrivé pour un premier stage en entreprise. Cette dernière est spécialisée dans la réalisation de versions web de jeux anciens, et souhaite rajouter à sa bibliothèque un jeu du labyrinthe.

Un précédent stagiaire, vous a laissé un export JSON contenant la description de plusieurs labyrinthes générés aléatoirement. Ce qu'il vous reste à faire est de créer une procédure permettant de résoudre le labyrinthe automatiquement.

Conseil général

Pensez à régulièrement lâcher la souris et le clavier et à travailler **ordinateur éteint** avec papier et stylo (ou tableau blanc et marqueur, ça fonctionne aussi !). Il est toujours très important de prendre du recul par rapport à son code. **Une étape fondamentale est la réflexion autour de la manière de structurer vos données**.

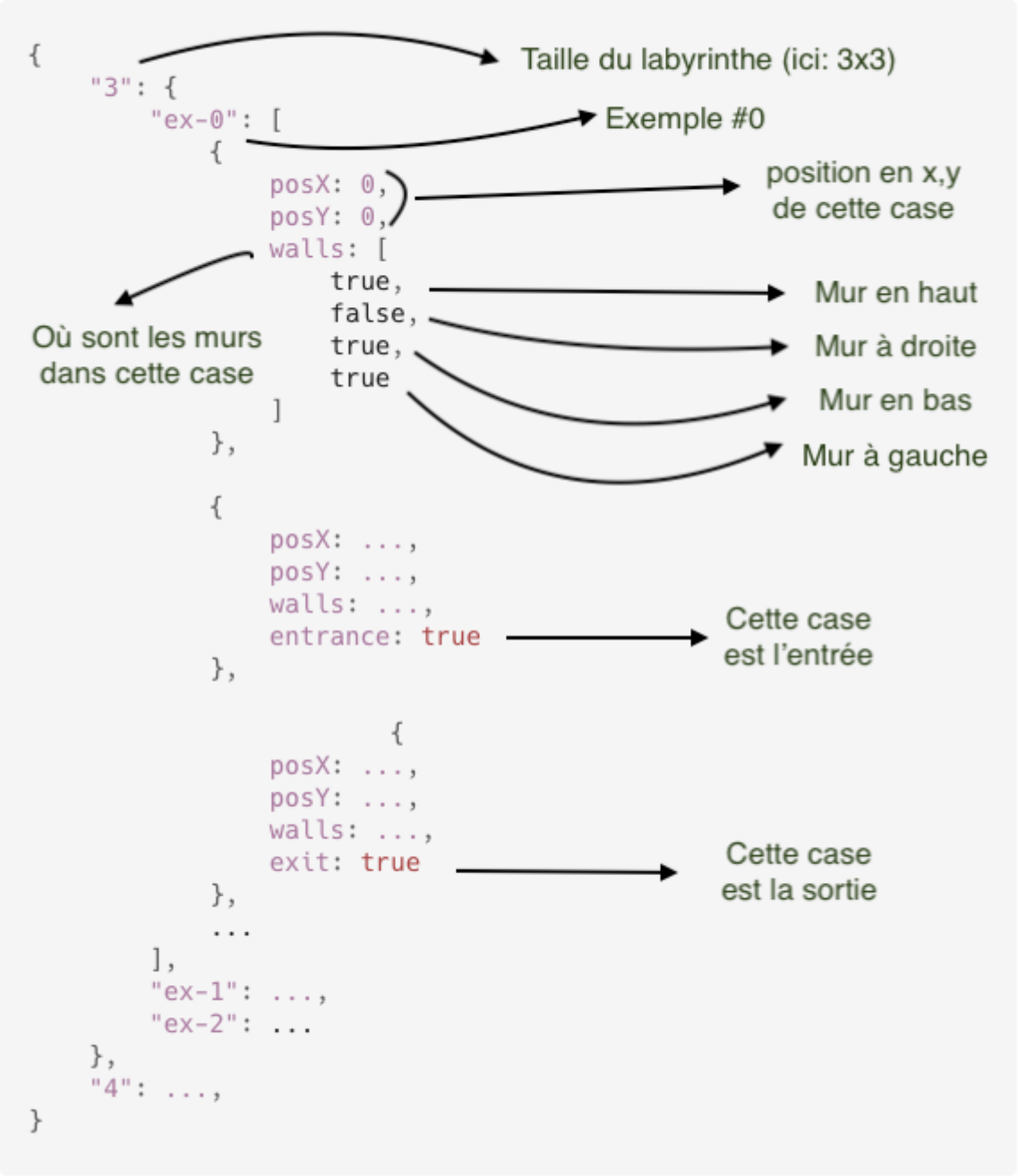
Livrables

➔ Au fil des étapes, les rendus doivent être “commits” sur un repository github structuré (ex un dossier par étape). Les rendus sont le code source, les réponses aux questions, les pseudo-code, etc.

1 — Construction du labyrinthe

1J — Présentiel

La première étape nécessaire à ce projet est la construction du labyrinthe en lui-même. Un fichier JSON contenant les informations nécessaires à la création du labyrinthe vous est fourni. Il est structuré de la manière qui suit :



Vous avez à votre disposition 66 labyrinthes dans ce fichier JSON : trois exemples de labyrinthe pour différentes tailles, allant de 3 x 3 cases à 25 x 25 cases. Commencez bien par les plus petits, ce qui vous permettra de débugger votre code.

Le troisième exemple de chaque taille (nommé ex-2) est souvent plus compliqué, car il a été modifié à la main pour tester les limites de vos algos.

OBJECTIF

1. Servez vous de ce fichier JSON pour créer le fameux labyrinthe. Le rendu final doit ressembler à la figure 1.
2. Passez du temps **hors ordinateur** pour réfléchir à la structure des données la plus adaptée au projet (j'ai déjà dit ça ?). La "qualité" de votre représentation des données déterminera la difficulté des étapes suivantes !

Conseil de construction du labyrinthe

- ➔ **Accès aux données du labyrinthe** :
  - une astuce très simple pour accéder aux données est la suivante → mettez les données du JSON dans un fichier `.js` (à part) et stockez ces données dans une variable. Appelez le script (dans la page HTML) contenant ces données avant le script JS qui vous servira à construire et résoudre le labyrinthe.
- ➔ Simplifiez vous la vie au maximum pour cette étape de construction du labyrinthe. Vous pouvez tout faire avec du CSS (flexbox) et un peu de JS.
- ➔ Commencez d'abord par créer une grille carrée avec des cases (sans tenir compte de la présence ou non de murs dans un premier temps)
  - **Astuce 1** → une case du labyrinthe = une div
  - **Astuce 2** → utilisez l'option *`box-sizing : border-box`*
  - **Astuce 3** → pensez à donner un ID à vos div (lors de la construction) pour pouvoir ensuite les sélectionner facilement.
  - **Astuce 4** → pour dessiner les murs : taille des *`borders`* > 0px

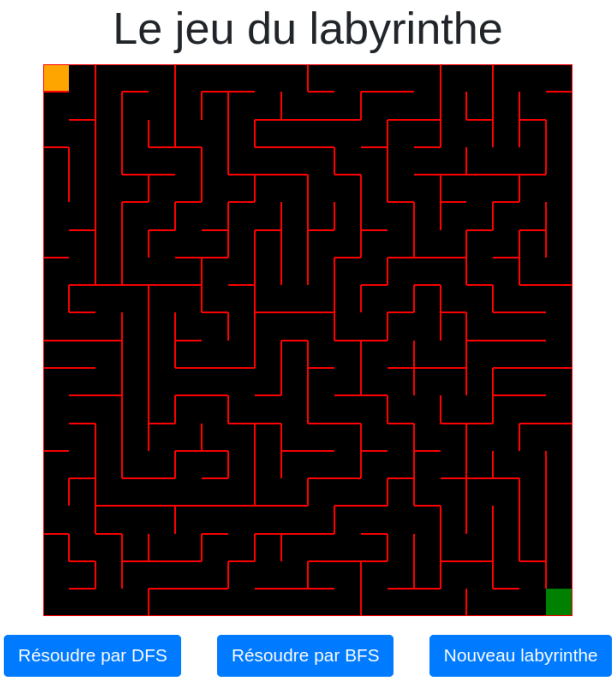


Figure 1. Exemple de labyrinthe généré.
L'entrée est en orange, la sortie en vert, les murs en rouge.

0.5J— Présentiel/Distanciel

2 —Résolution du labyrinthe - Intuition

OBJECTIF

Établir une première méthode de résolution "à votre sauce" **SUR PAPIER**

ETAPES

- Chacun de votre côté, pendant 15 minutes, imaginez vous dans le labyrinthe, et décrivez comment vous pourriez trouver la sortie (en vert) depuis l'entrée (en orange).
- Ecrivez votre algorithme sous forme de pseudo-code (pas de formalisme particulier attendu : Pseudo-code — Wikipédia)
- Expliquez votre algorithme à votre îlot. Comparez-les ensuite à votre algorithme. Êtes-vous arrivés indépendamment au même algorithme de recherche ?
- Il n'est pas demandé d'optimisation. C'est normal que votre algorithme se perde en chemin, et que ce ne soit pas "le plus court". Assurez-vous simplement que quel que soit le labyrinthe fourni, vous puissiez trouver à coup sûr la sortie du labyrinthe.

PISTES DE RÉFLEXION

- Afin de faciliter la réflexion, imaginons que le labyrinthe soit de taille réduite, par exemple : 6 x 6 cases. Une fois la logique établie sur un petit labyrinthe, elle devrait se généraliser sur un labyrinthe de taille quelconque.

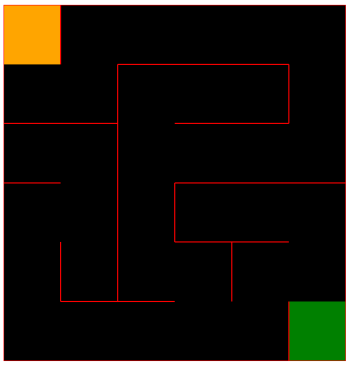


Figure 2 : Exemple de labyrinthe de taille 6 x 6

- Vous pouvez emporter un sac de riz avec vous dans le labyrinthe et jouer au petit poucet.

Figure 3 : Un indice de taille !

1.5J— Présentiel

3 —Résolution du labyrinthe - Parcours DFS

OBJECTIF

Implémenter une méthode de recherche de chemin.

ETAPES

1. Comparez votre algo “intuition” avec le pseudo-code du parcours dit **Depth First Search (DFS)** (parcours en profondeur d’abord). Avez-vous eu une idée similaire?

Version itérative de DFS

```
DFS_iterative (G, e):
  → let S be a stack
  → insert e in the stack

  while ( S is not empty ) :
    → Pop an element v from stack to visit next
    if v was not visited :
      → mark v as visited
      if v is the exit :
          → return path from e to v
      for all neighbours w of v in graph G:
        if w was not visited :
          → Tag v as the parent of w
          → insert w in the stack
  → return Undefined

------------------------------------------------------------
Explication des différentes variables :
   • G → votre labyrinthe
   • e → entrée du labyrinthe
   • v → le vertex courant (la position actuelle)
   • Stack → voir ressource
     https://www.geeksforgeeks.org/stack-data-structure-introduction-program/

“Tag v as the parent of w” est utile pour retourner le chemin en sortie de fonction, en revenant sur ses pas (en remontant les parents successifs depuis la sortie).
```

Version récursive de DFS

```
  DFS_recursive (G, v):
    if v was not visited :
      → mark v as visited
        if v is the exit :
          → return v
        for all neighbours w of v in graph G :
          → let path be the result of calling DFS_recursive (G, w)
          if path is valid :
            → return the concatenation of v and path
    → return Undefined

------------------------------------------------------------
Explication des différentes variables :
   • G → votre labyrinthe
   • v → le vertex courant (la position actuelle)

Retourner le chemin revient à remonter la pile d'appels.
```

Si vous avez peur de la récursivité, c'est normal. Allez voir cette ressource :

- Récursivité : How Recursion Works — Explained with Flowcharts and a Video
- Call stack (pile d'appel) : Recursion: The Call Stack

2. Jouer ces deux versions de l'algorithme en utilisant des post-its (vous pouvez ignorer la partie en marron pour cette étape).
3. Quelques ressources pour comprendre :
   - Depth First Search Tutorials & Notes | Algorithms
   - Depth-First Search (DFS)
   - Uninformed Search 1 - Depth-First Search
4. Il est plusieurs fois question de graphes dans ces explications. Quel est le rapport avec votre projet ? Êtes-vous capable de représenter le labyrinthe sous forme de graphe ?

Livrables

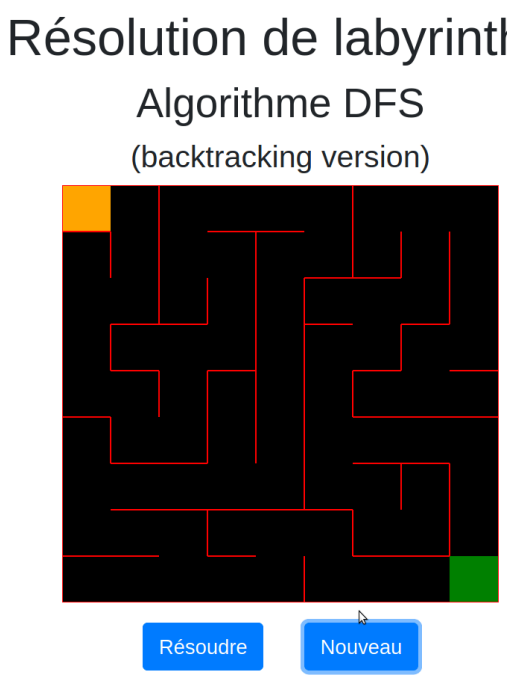
➔ Comprendre l'algorithme de parcours DFS.
➔ Dessiner un exemple de labyrinthe et sa représentation sous forme de graphe.

5. Codez cet algorithme de résolution DFS
    - Langage à utiliser -> JS
    - Affichez à chaque étape les coordonnées de la case courante.
    - Lorsque vous trouvez la sortie, arrêtez le programme et affichez le chemin entre l'entrée et la sortie.
    - Affichez le nombre de cases qui ont été visitées et la taille du chemin trouvé.

6. Faites les deux versions, itérative et récursive.

CONSEILS

- Utilisez **votre débugger** comme si votre vie en dépendait.
- Assurez vous de la validité de votre implémentation en testant plusieurs labyrinthes



GIF#1 : Exemple d'animation possible sur DFS.

En noir les zones pas encore explorées. En bleu : la case courante évaluée. En Violet les zones explorées. En gris, les impasses identifiées. En vert, la sortie. En rouge, les murs du labyrinthe.

4 —Résolution du labyrinthe - Parcours BFS

1J— Présentiel

OBJECTIF

Implémenter un autre algorithme de recherche : **Breadth First Search** (BFS)
→ parcours en largeur d'abord.

ETAPES

1. Il existe de nombreuses façons de parcourir des graphes. Une autre manière adaptée, dans le cas du labyrinthe est la recherche BFS
   Ressource : Uninformed Search 2 - Breadth-First Search
2. Essayez de comprendre seuls et en îlot le pseudo code suivant :

```
BFS (G, e)
  → let Q be a queue.
  → push e to Q.
  while ( Q is not empty ) :
    → let v be the next element waiting in the queue
    if v was not visited :
      → mark v as visited.
      if v is the exit :
        → return path from e to v
      for all neighbours w of v in graph G
        if w is not visited :
          → Tag v as the parent of w
          → add w to the queue
  → return Undefined


Ressource : Queue →
https://www.geeksforgeeks.org/queue-data-structure/
```

3. Implémentez l'algorithme BFS, en suivant les mêmes étapes que pour le DFS.
4. Essayez de modifier votre programme de manière à faire tomber quelques murs aléatoires dans le labyrinthe. Assurez-vous que vos algos de résolution fonctionnent encore.
5. Comparer les résultats du DFS et du BFS dans ce cas.
   a. DFS et BFS trouvent-ils toujours le même chemin ?
   b. Quelles sont les différences de résultat ? Pourquoi ?

6. Si vous ne l'avez pas encore remarqué, la seule différence entre le DFS et le BFS est la structure de données pour stocker les nœuds à visiter. Vous pouvez même passer de l'un à l'autre en changeant une ligne dans le code. Saurez-vous trouver laquelle ?

5 —Pour aller plus loin

OBJECTIF

Aller creuser les concepts développés jusqu'à maintenant.

OPTION #1

Fabriquer un générateur aléatoire de labyrinthe

- Les données que vous avez reçu pour générer vos labyrinthes ont été obtenues par implémentation d'un **"Recursive randomized depth-first search"**. Vous pouvez choisir votre méthode.
- Analyse : quels sont les cas qui ne sont pas couverts par cette méthode ? Est-ce réellement un labyrinthe le plus général possible. Vous pouvez-vous aider d'une représentation en graphe pour aboutir à la bonne solution.
- Ressource : Maze generation algorithm - Wikipedia

OPTION #2

Ajouter une difficulté à passer sur certaines cases (par exemple, elles sont remplies de sable).
Tenez en compte pour trouver le chemin le plus rapide (qui n'est pas forcément le plus court).
Vous pouvez pourrez pour cela regarder les algorithmes de Dijkstra et A* (A-Star)

OPTION #3

Le parcours de graphe permet de solutionner tout un ensemble de problèmes/jeux.
Vous pouvez utiliser le BFS ou le DFS (avec une profondeur max) pour programmer un algo qui joue à un jeu de plateau morpion, dames, échecs, taquin …
Chaque coup possible correspondant à noeud du graphe, la sortie étant la victoire ou, pour les jeux qui ont une multitudes de coups possibles et pour lesquels il n'est pas possible de tout tester, le chemin vers le nœud qui vous procure le meilleure score.