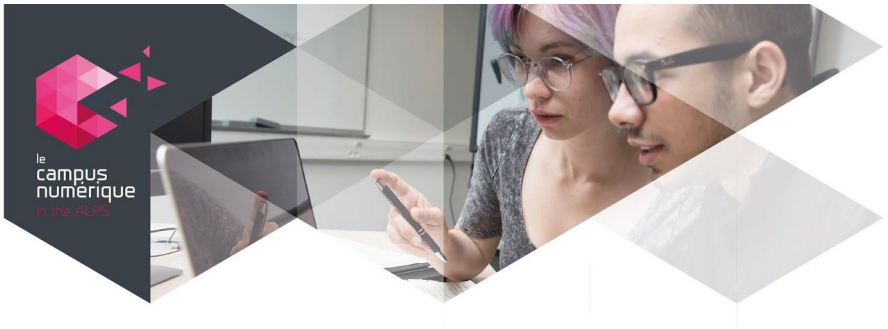





# Algorithmique

## Introduction

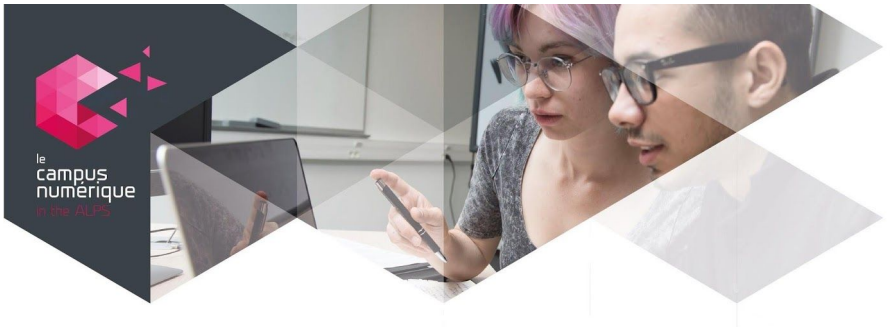
Un algorithme consiste à décrire de manière formelle, précise, les étapes pour résoudre un problème. Par exemple, pour calculer une moyenne de plusieurs valeurs :

- démarrer avec un accumulateur *a* à 0 et un compteur *n* à 0 ;
- pour chaque valeur l'ajouter à *a* et incrémenter *n* ;
- quand il n'y a plus de valeurs, la moyenne est *a/n*.

Les algorithmes existent avant et sans les ordinateurs et l'informatique (voir par exemple sur https://fr.wikipedia.org/wiki/Algorithmique), mais c'est la forme "naturelle" pour exprimer un traitement informatique. L'algorithme est "indiqué" à l'ordinateur par l'intermédiaire d'un langage de programmation.

Si différents langages (vous avez déjà rencontré Javascript, PHP, Java, CSS) apportent différentes briques de bases pour exprimer des algorithmes (fonctions, boucles, variables, constantes, …), il y a un invariant : Un algorithme consiste à effectuer des opérations sur des données. La structure de ces données (tableau, liste, dictionnaire, …) fait partie, de même que les structure de contrôle (types de boucles, appels de fonction, …) d'un algorithme.

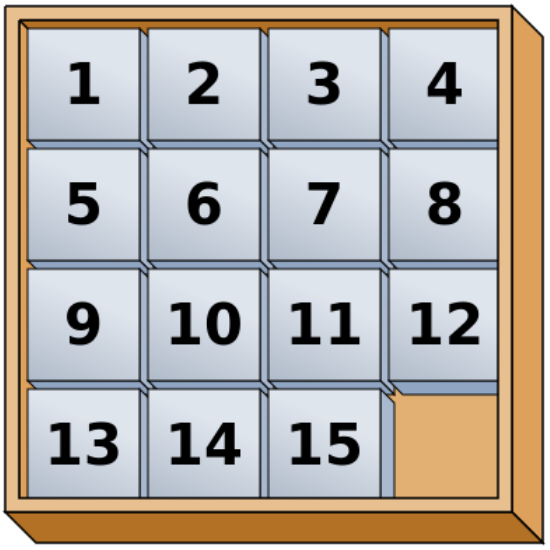
L'algorithmique est l'étude des algorithmes, de leurs propriétés. Par exemple, est-ce qu'ils finissent ? Combien de temps prennent-ils ? Quelle place mémoire utilisent-ils ? Il ne s'agit pas ici - comme d'habitude - de faire un cours, mais par le biais de la réalisation d'un projet, jeu du taquin, de se concentrer sur des compétences qui relèvent plus de l'algorithmique en tant que telle que des techniques du Web.






# Jeu du taquin

Le jeu du taquin est un casse-tête célèbre : Il s'agit d'un carré de 4x4 cases, qui ont un ordre précis, dont une case est vide :




La première étape du jeu consiste à le mélanger en faisant glisser les cases et la seconde étape consiste à revenir à l'état initial !

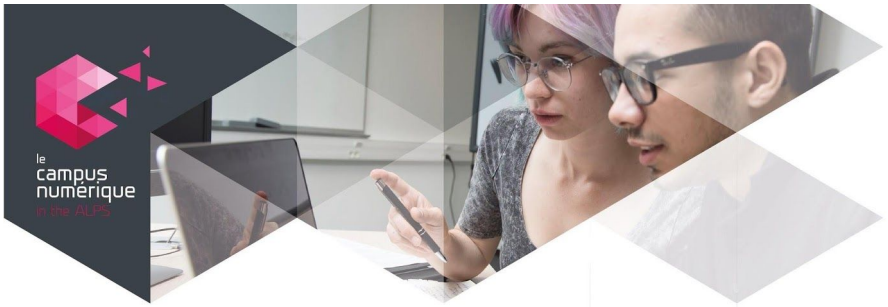
## Objectifs

Nous allons travailler sur un taquin en **3x3** afin de faciliter les résolutions manuelles. Les algorithmes seront les mêmes que pour des tailles plus grandes.

Dans ce projet, vous allez :

- faire une page Web reproduisant le jeu "physique" (*vous utiliserez Javascript et JQuery, chargé dans une page Web. À nouveau, l'objectif n'est pas la programmation Web utilisant un framework.*)

- ajouter des fonctionnalités automatique :
  - mélange du jeu ;
  - vérification de l'état ("est-ce gagné ?") ;
  - résolution automatique (en bonus)





## Activité offline

Nous allons apprivoiser le jeu du Taquin en le recréant physiquement avec des feuilles de papier.

### Exercice 1

Positionner le jeu dans l'état initial et faire aléatoirement des mouvements (en respectant les contraintes physiques) puis résoudre.

### Exercice 2

Mélanger les feuilles et les positionner aléatoirement puis résoudre.

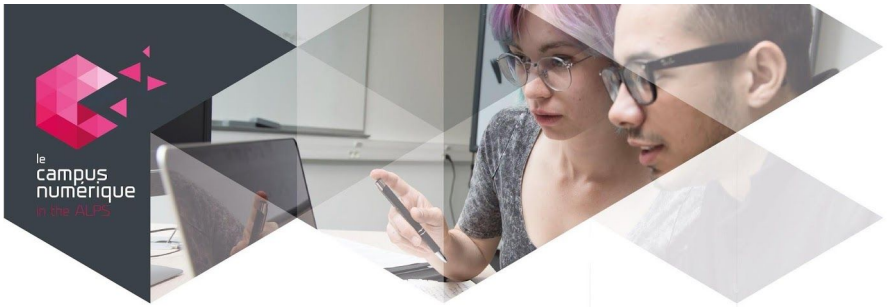
Vous avez remarqué que si l'on mélange les pièces sans tenir compte des contraintes physiques d'un vrai jeu, il se peut que le jeu n'ai pas de solution, on se retrouve bloqué. Il existe un test qui permet de vérifier si, à partir d'une position donnée, il existe une solution.

Cet algorithme est expliqué dans cette vidéo: https://www.youtube.com/watch?v=-3IsCOJieCc

En fait - et nous nous limiterons à l'intuition - le puzzle a une solution si la parité du nombre de permutations pour mettre les tuiles à leur place (en "trichant", c'est-à-dire en démontant le jeu) est la parité du nombre de mouvement pour amener la case vide à sa place. Pas de panique ! Lisez d'abord
https://fr.wikipedia.org/wiki/Taquin#Configurations_solubles_et_insolubles

### Exercice 3

Utiliser l'algorithme expliqué dans la vidéo pour tester si il existe une solution pour votre jeu.






# Activité online

**Indices** : Ne nous précipitons pas ! Comme écrit dans l'introduction, une première étape est de réfléchir aux structures de données qui vont permettre de représenter l'état du jeu et de permettre facilement la réalisation des opérations, c'est-à-dire les changements d'états. Comme toujours, la lecture au moins rapide de la suite peut être intéressante.

### Exercice 1: Afficher le plateau de jeu

Cette étape consiste à écrire la page Web affichant le plateau de jeu. Lors du chargement de la page, on doit être dans l'état initial (tel que l'image ci-dessus).

### Exercice 2: Mélanger

Implémenter un algorithme permettant de mélanger les pièces (sans respecter les contraintes du jeu, comme dans l'exercice 2 en offline) et afficher le résultat.

### Exercice 3: Gagné ?

Implémenter l'algorithme permettant de savoir si il existe une solution pour la position générée aléatoirement.

- **Étape 1** : faire une fonction qui détermine la parité de la case vide. **Indice** : faites un dessin avec des couleurs différentes pour les cases paires et impaires et relier ceci aux numéros des lignes et colonnes.

- **Étape 2** : faire le calcul des permutations pour obtenir la parité de la solution. **Indices** : lisez bien l'exemple 3 du paragraphe de Wikipedia "Configurations solubles et insolubles". Si vous remplacez la case vide par "9", il est clair qu'il s'agit en fait de l'algorithme de tri par sélection (https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection) que vous avez déjà vu. Le taquin peut être représenté par un tableau à 2 dimensions 3x3, mais on peut aussi mettre les lignes bout à bout et considérer un tableau de 9 éléments si cela est plus simples pour certains raisonnements : on peut adapter les données ou le raisonnement.

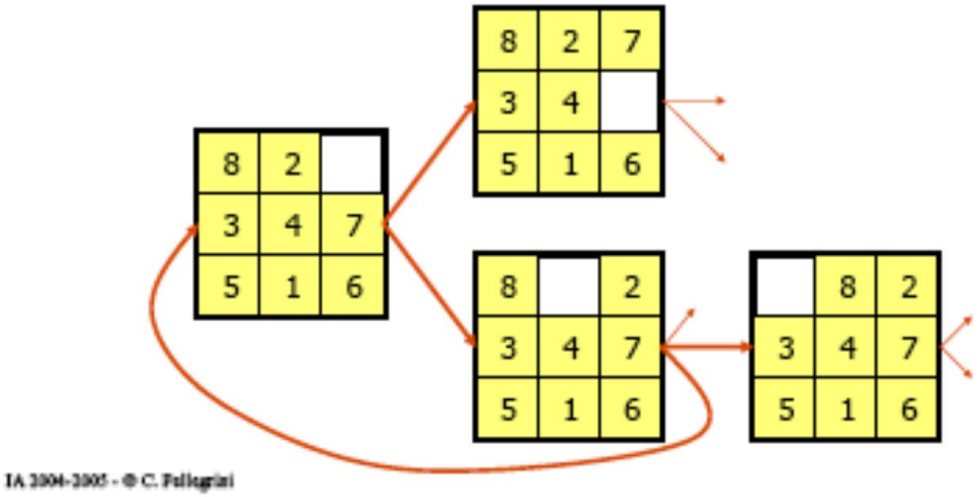
### Exercice 4: Jeu gagnable

Utiliser les deux exercices précédents pour faire une fonction qui donne une position de départ gagnable.

## Bonus: Résolutions

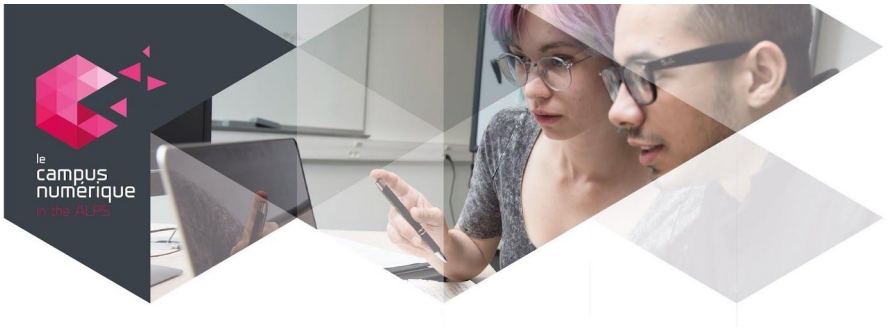
Nous allons maintenant écrire des algorithmes pour trouver un solution à un état de départ donné.

L'objectif est d'atteindre l'état gagnant à partir de l'état de départ. Il s'agit d'un exploration dans l'ensemble des états atteignables depuis l'état de départ. Par exemple, pour une situation arbitraire et pour le taquin 3x3 :




Pour le raisonnement algorithmique il est plus simple de considérer le mouvement de la case vide (depuis l'état gagnant, la case vide peut monter ou aller sur la gauche). Par exemple gauche-gauche-haut-haut-droit-bas (ou G-G-H-H-D-B, ou L-L-U-U-R-D en anglais pour coller à diverses ressources que l'on peut trouver sur Internet pour la résolution du taquin).

Les états sont reliés par des flèches et on doit parcourir ces états en respectant le sens des flèches. C'est ce que l'on appelle un graphe (des points [= états] relié) orienté (les flèches qui imposent le sens de parcours).
Il existe plusieurs stratégies pour construire et parcourir le graphe de possibilités pour trouver un chemin vers la solution.
**Note** : ces chemins peuvent être dénoté par les mouvements de la case vide. On a pour l'illustration précédente le chemin "D" (down) pour l'état du haut et le chemin "L-L" pour l'état en bas à droite.
**Note 2** : les points du graphe sont généralement appelés sommets ou nœuds.






Il existe de multiples algorithme d'exploration de graphe, mais nous allons tout d'abord nous concentrer sur le parcours en dits "en profondeur". Nous évoquerons aussi les parcours "en largeur", puis A* ou IDA*.

Pour les divers algorithmes qui suivent, dans un premier temps déclencher leur exécution en ajoutant un bouton qui va les appeler à partir de l'état courant de votre taquin. Affichez les résultats dans un "div" dans le bas de la page, qui sera votre "debug bar". Dans un second temps, chaque algorithme donnera un chemin qui pourra être "joué" dans la page du jeu - par exemple un coup toutes les 500 ms.

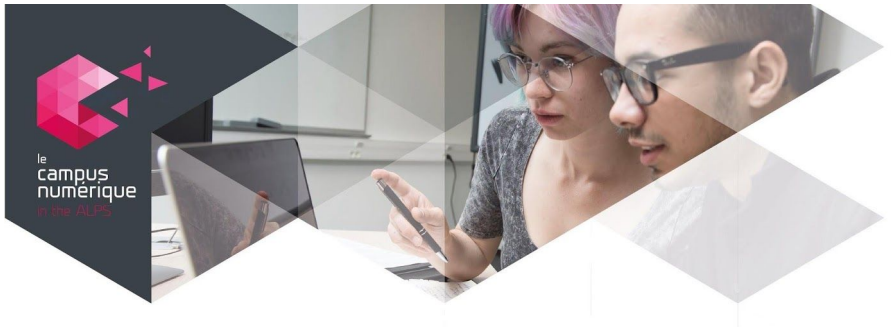
**Résolution 1 : "en profondeur d'abord"**

Consultez la vidéo https://www.youtube.com/watch?v=pZgPD2QOb_0

Notes sur la vidéo :
**Note 1** : nous avons pu considérer que la case vide était 9 au lieu de 0, pour faciliter l'algorithme de tri ci-dessus, mais dans cette vidéo, les valeurs ne sont que des symboles.
**Note 2** : dans la vidéo, un coup est noté par la nouvelle position de la case vide. On peut aussi le représenter par une des valeur U(p), D(own), L(eft) et R(ight).
**Note 3** : dans un premier temps, on peut ignorer la prise en compte du mouvement précédent. C'est ici une optimisation car le nombre de mouvements étant limité (par MAX_DEPTH), on ne pourra pas tourner dans une boucle du graphe à l'infini ! De plus, le raisonnement peut être poussé plus loin et pas seulement des cycles entre 2 états...
**Note 4** : on peut également ignorer le problème de la recherche optimale (nombre minimum de mouvements) : c'est dans la vidéo à partir de la minute 47 à propos de *best_moves* et *best_depth*. On peut d'abord chercher une solution sans chercher le nombre minimum de coups.

Élements de solution:

- Ecrire une fonction *isWinPosition* qui indique si une position est gagnante
- écrire une fonction *possibleMoves* qui renvoie les mouvements possibles en fonction de la position de la case vide et du mouvement précédent pour éviter de revenir en arrière dans l'exploration et de tourner en rond.
- un pseudo-code de DFS ne cherchant pas peut être le suivant :

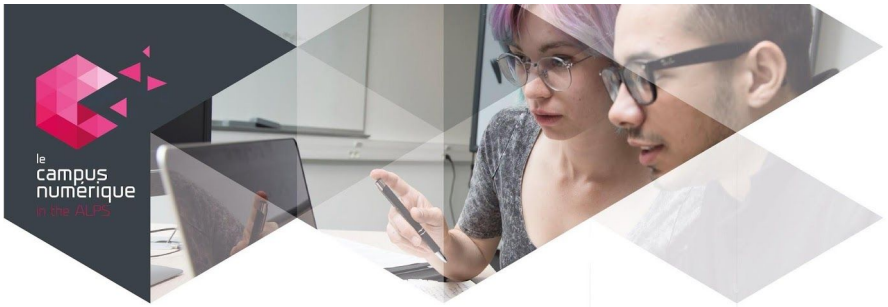





```
DFS(m, e, p) // m = maximum, e = état, p = profondeur
    si p > m alors // trop de coups
        renvoyer FAUX
    si isWinPosition(e) alors
        renvoyer VRAI
    pour chaque x dans possibleMoves(e)
        ee = x(e) // on applique le mouvement x sur e => ee
        si DFS(m, ee, p+1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

**Bonus 1** :
L'algorithme ci-dessus est récursif et peut poser des problèmes d'exécution (que vous pouvez rencontrer avec MAX_DEPTH suffisamment grand) : dépassement de la profondeur maximale de la pile d'appel. On peut alors considérer le pseudo-code suivant qui au passage donne un indice pour optimiser en gardant les états déjà rencontrés (mais ne limite pas la profondeur !) :

```
DFS(e)
    frontière = Stack(e) // états depuis lesquels explorer
    connus = Set() // états déjà rencontrés
    tant que non frontière.empty()
        état = frontière.pop()
        connus.add(état)
        si est_gagnant(état) alors
            renvoyer VRAI
        pour chaque x dans possibleMoves(état)
            si x not in (frontière U connus) alors
               frontière.push(x)
    renvoyer FAUX
```






**Résolution 2 : “en largeur d’abord”**

Cet algorithme est introduit mais non développé dans la vidéo précédente (en particulier à partir de 9’53). En voici le pseudo-code :

```
BFS(e)
    frontière = File(e) // états depuis lesquels explorer
    connus = Set() // états déjà rencontrés
    tant que non frontière.empty()
        suivants = File()
        pour chaque x dans frontière
            si est_gagnant(x) alors
                renvoyer VRAI
            suivants.enfiler(possibleMoves(x))
        frontière = suivants
    renvoyer FAUX
```

Indice : à nouveau le pseudo-code indique l’”ossature” du raisonnement. Il faut ajouter la construction d’un chemin et le renvoyer quand on a trouvé une solution.

**Résolution 3 : “en profondeur avec horizon” (IDA*)**

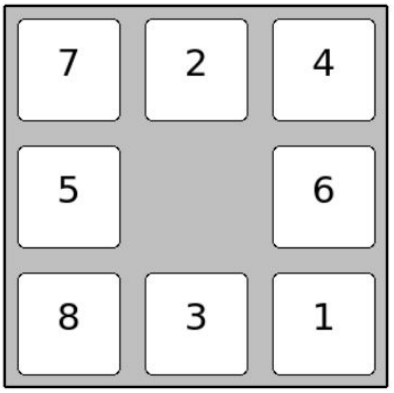
https://en.m.wikipedia.org/wiki/Iterative_deepening_A*

On procède alors comme pour la recherche itérée en profondeur, mais avec une heuristique (approximation) qui nous permet d’estimer la distance (le nombre de coups) qu’il reste à parcourir pour arriver à une solution. Si on note cette heuristique pour un état *e* par *h(e)* (c’est-à-dire l’estimation du nombre de coups pour atteindre une solution depuis *e*), pour chaque état *e* considéré à la profondeur *p* on s’interrompt dès que *p + h(e)* dépasse la profondeur maximale *m* (au lieu de s’arrêter simplement lorsque *p* > *m*).

Intuitivement, on va favoriser les directions les plus prometteuses.

Pour les heuristiques, il y a des conditions mathématiques que nous allons ignorer ici. On va considérer deux possibilité, le nombre de pièces mal placées (*h1*) et la somme des “distances de Manhattan” (*h2*). La distance de Manhattan se définit par le nombre de déplacements (horizontaux ou verticaux) à faire pour ramener la pièce au bon endroit, en ignorant toutes les autres pièces. Dans l’exemple ci-dessous, *h1* = 6 car seul “2” et “6”

sont à leur place et *h2* = 4+0+3+3+5+1+0+2+1 = 19 car il faut 4 déplacements pour mettre “1” à sa place, aucun pour “2”, etc.

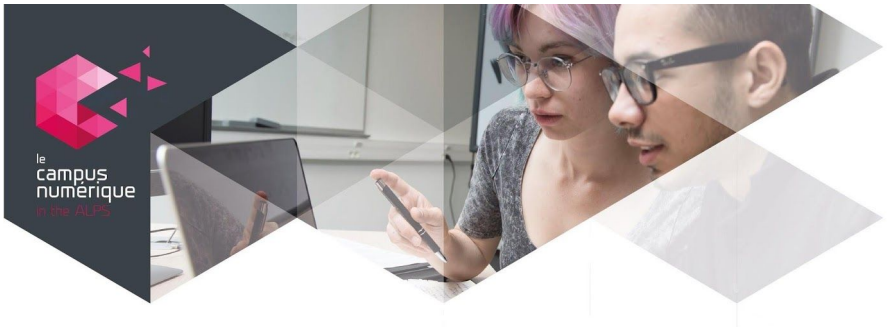



L’idée est la suivante :

- on prend une valeur *MAX_DEPTH*, comme limite de la profondeur de recherche (quoi qu’il arrive, on ne cherchera pas plus profond) ;
- on part d’un état initial *e0* et on calcule une approximation du nombre de coups maximal avec *h(e0)* que l’on considère comme un premier objectif pour DFS.
- on fait une recherche DFS avec ce maximum de profondeur, en gardant la plus petite valeur (variable globale) qui dépasse le maximum en cas d’échec comme nouvel objectif (si plus petit que *MAX_DEPTH*).

```
IDA*(e0)
   m = h(e0)
   tant que m <= MAX_DEPTH
      min = MAX_DEPTH
      si DFS*(m, e0, 0) = VRAI alors
         renvoyer VRAI
      m = min
      renvoyer FAUX

DFS*(m, e, p)
   c = p+h(e)
   si c > m alors
      min = c // on dépasse, il faut aller au moins à la profondeur c (gardée dans min)
   renvoyer FAUX
   si est_gagnant(e) alors
      renvoyer VRAI
   pour chaque x dans possibleMoves(e)
      si DFS*(m, x, p+1) = VRAI alors
```





```
        renvoyer VRAI
    renvoyer FAUX
```

**Résolution 4 : “Best First, Glouton, A*”**

À vous de jouer ! : https://fr.wikipedia.org/wiki/Algorithme_de_recherche_best-first et http://www.math-info.univ-paris5.fr/~bouzy/Doc/IAL3/03_IA_heuristique_BB.pdf