

# 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  $a/n$ .

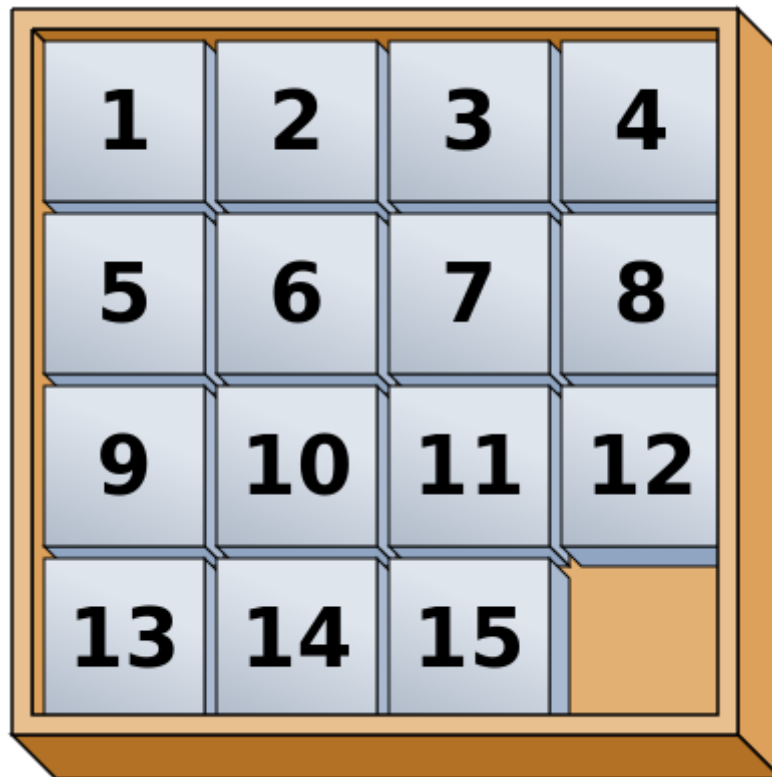
Les algorithmes existent avant et sans les ordinateurs et l'informatique (voir par exemple sur <https://fr.wikipedia.org/wiki/Algorithmique>), mais c'est la forme "naturelle" pour exprimer un traitement informatique. L'algorithme est "indiqué" à l'ordinateur par l'intermédiaire d'un langage de programmation.

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 la 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 !

## Réalisation

Dans ce projet, vous allez :

- faire une page Web reproduisant le jeu "physique" ;
- ajouter des fonctionnalités automatique :
  - vérification de l'état ("est-ce gagné ?") ;
  - mélange du jeu ;
  - résolution automatique.

Vous utiliserez Javascript et JQuery, chargé dans une page Web. À nouveau, l'objectif n'est pas la programmation Web utilisant un framework.

### 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). Il faudra implémenter les fonctionnalités suivantes :

- bouton pour revenir à l'état initial ;
- possibilité d'indiquer un changement (par exemple pour l'état initial, on peut pousser le "12" vers le bas ou le "15" vers la droite).

**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.

**Bonus** : au lieu d'afficher des nombres de 1 à 15, utilisez une image découpée.

### Gagné ?

Ajouter un affichage qui indique si le taquin est dans l'état gagnant.

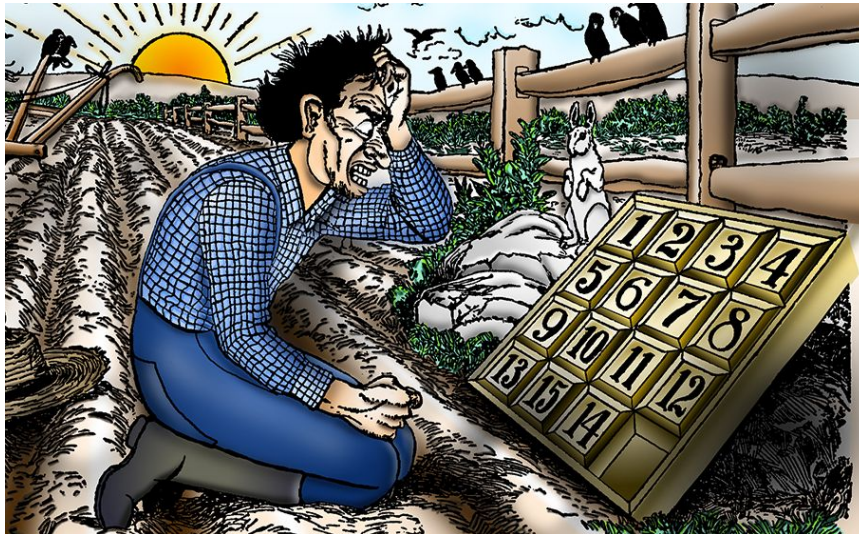
### Mélange

Il s'agit de reproduire le mélange qui serait effectué sur un jeu physique : on déplace les tuiles jusqu'à obtenir un défi "satisfaisant". Dans notre projet, nous allons simplement déplacer les tuiles avec un nombre de coups décidés à l'avance.

**Indices** : précédemment, on indiquait la tuile que l'on déplaçait et son mouvement. Il est en fait plus simple pour le raisonnement algorithmique de considérer le mouvement de la case vide (depuis l'état gagnant, la case vide peut monter ou aller sur la gauche). Un mélange sera donc une succession de mouvements **valides** de la case vide. 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 15 puzzle - ou 8 puzzle pour la version 3x3).

## Gagnable ?

Le jeu de taquin, sous son nom américain "15 puzzle" a été popularisé par Sam Loyd ([https://fr.wikipedia.org/wiki/Sam\\_Loyd](https://fr.wikipedia.org/wiki/Sam_Loyd)) avec un concours qui n'avait en fait pas de solution (d'où le nom du jeu !) :



Si vous observez le mélange vous voyez que les 2 tuiles "14" et "15" ont été inversées par rapport à l'état gagnant.

En fait - et nous nous limiterons à l'intuiter - 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](https://fr.wikipedia.org/wiki/Taquin#Configurations_solubles_et_insolubles)

**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.

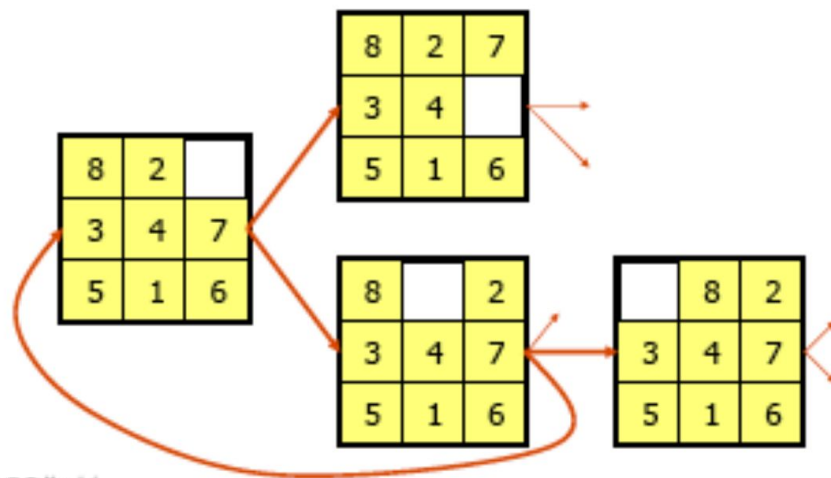
**Étape 2** : faire le calcul des permutations pour obtenir la parité de la solution. **Indices** : lisez bien l'exemple 3 du paragraphe de Wikipedia "Configurations solubles et insolubles". Si vous remplacez la case vide par "16", il est clair qu'il s'agit en fait d'un algorithme de tri. Ce tri s'appelle le tri par sélection ([https://fr.wikipedia.org/wiki/Tri\\_par\\_s%C3%A9lection](https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection)). Le taquin peut être représenté par un tableau à 2 dimensions 4x4, mais on peut aussi mettre les lignes bout à bout et considérer un tableau de 16 éléments si cela est plus simples pour certains raisonnements : on peut adapter les données ou le raisonnement.



## Résolutions

Nous allons maintenant écrire des algorithmes pour trouver une 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'une exploration dans l'ensemble des états atteignables depuis l'état de départ. Par exemple, pour une situation arbitraire et pour le taquin 3x3 :



IA 2004-2005 - © C. Pellegrini

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 notés 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 algorithmes d'exploration de graphe, mais nous allons tout d'abord nous concentrer sur les parcours dits "en largeur" et "en profondeur", 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. **Bonus général** : quand un chemin est joué, laissé la possibilité à l'utilisateur d'interrompre en continuant en mode manuel.

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

Consultez la vidéo [https://www.youtube.com/watch?v=pZgPD2QOb\\_0](https://www.youtube.com/watch?v=pZgPD2QOb_0)

Notes sur la vidéo :

**Note 1** : nous avons pu considérer que la case vide était 16 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(ef) 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.

Éléments de solution:

**Éléments 1** : 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.

**Éléments 2** : 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 est_gagnant(e) alors
    renvoyer VRAI
  pour chaque x dans possibleMoves(e)
    si DFS(m, x, p+1) = VRAI alors
      renvoyer VRAI
  renvoyer FAUX
```

**Éléments 3** : ne pas oublier le cas où la solution n'est pas trouvée...

## 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
```

**Bonus 2** : on peut adapter DFS pour chercher d'abord à une distance de 1, puis 2, puis 3, ... jusqu'à MAX\_DEPTH. C'est ce que l'on appelle "en profondeur itératif".

**Remarque 1** : pour un taquin à 16 cases, le nombre d'états différents est de l'ordre de 100 000 milliards, il est donc important d'avoir une stratégie !

## 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\\*](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 possibilités, 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.



7	2	4
5		6
8	3	1

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](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](http://www.math-info.univ-paris5.fr/~bouzy/Doc/IAL3/03_IA_heuristique_BB.pdf)