Complexité des algorithmes, deux exemples ludiques

Michel Vasquez

Séminaire DHM
19 novembre 2020

michel.vasquez@mines-ales.fr

Complexité des Algorithmes

Nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme.

Elle s'exprime en fonction de la taille n des données.

Notée $O(f(n))$.

Typiquement f est une combinaison de polynômes, logarithmes ou exponentielles. Cela veut dire que le nombre d'opérations effectuées est borné par $c \times f(n)$, où c est une constante, lorsque n tend vers l'infini.

Classes de complexité

1. $O(log_2(n))$: recherche dichotomique d'un élément dans un ensemble ordonné fini,
2. $O(n)$: recherche d'une clé dans une table de n éléments,
3. $O(n \times log_2(n))$: algorithmes optimaux de tri,
4. $O(n^2)$ et $O(n^3)$: multiplication de matrices, algorithmes de parcours dans les graphes,
5. $O(2^n)$ supérieure à tout polynôme en n : énumération exhaustive, recherche arborescente.

Les algorithmes appartenant aux quatre premières classes sont de **complexité polynomiale**.

Ceux relevant de la dernière sont de **complexité exponentielle**.

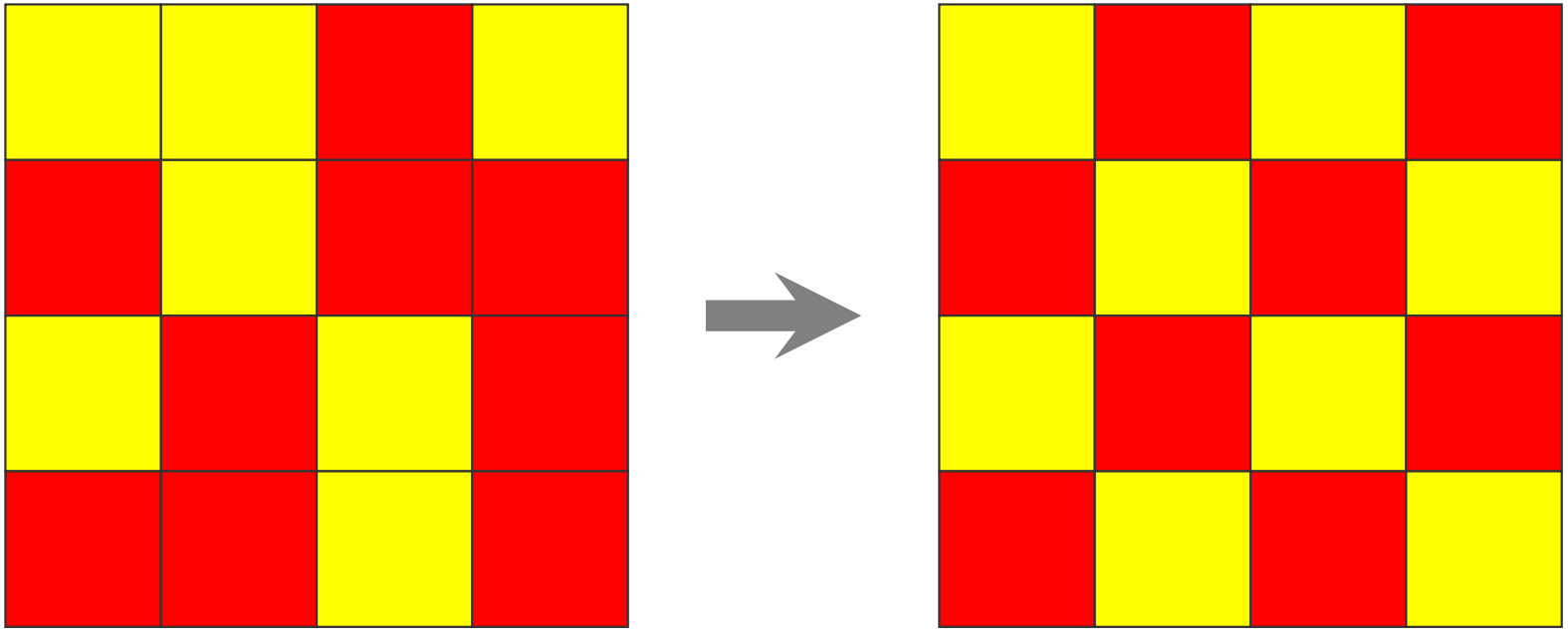
Temps d'exécution

En supposant que la machine effectue une instruction en 1 millionième de seconde, nous obtenons les temps de calcul suivants (n est le nombre d'instructions) :

n	O(log(n))		$O(n^3)$		$O(2^n)$	
10	3,32	micro sec.	1,00	milli sec.	1,02	milli sec.
20	4,32	micro sec.	8,00	milli sec.	1,05	seconde
40	5,32	micro sec.	64,00	milli sec.	305,42	heures
80	6,32	micro sec.	512,00	milli sec.	383,35	millions de siècles

Exemple 1 : Puzzle Merlin

Comment passer de la configuration de gauche à celle de droite en un **minimum** de **clicks** ?

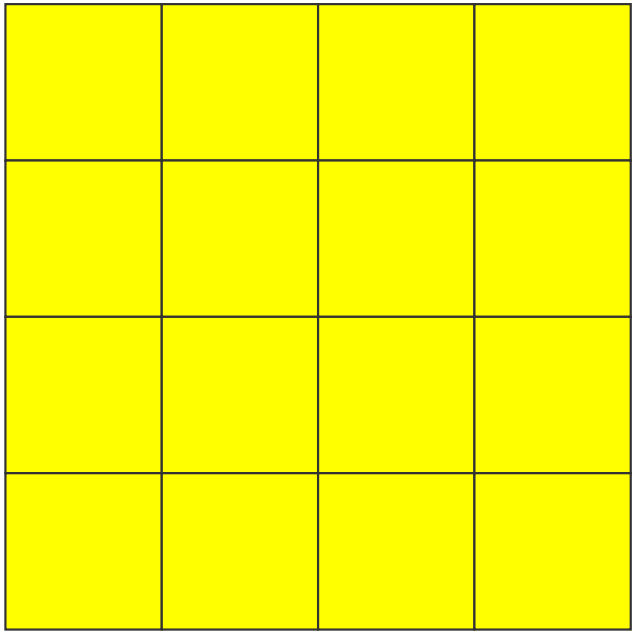


$(0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, ...) \rightarrow (0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0...)$

- case rouge $\rightarrow 1$
- case jaune $\rightarrow 0$
- configuration du puzzle $\leftrightarrow$ vecteur $\{0, 1\}^{16}$

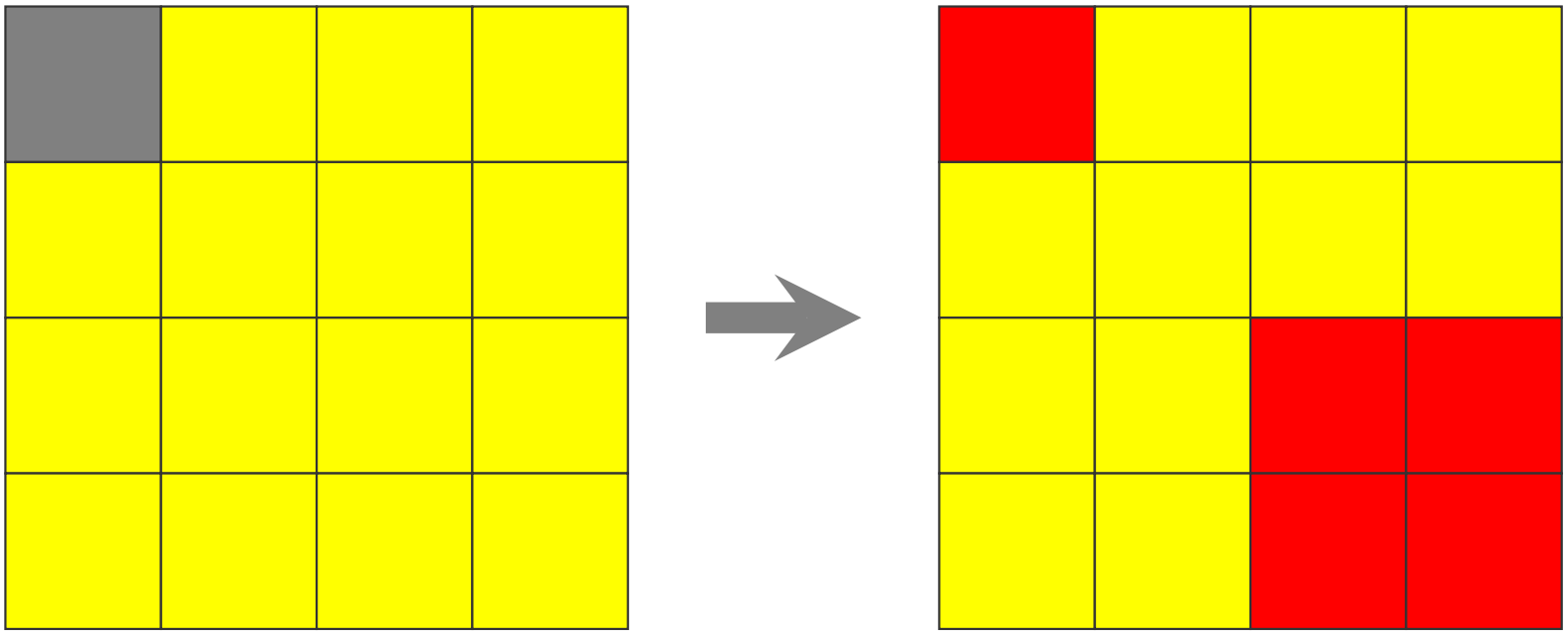
Puzzle Merlin, règles du jeu et modélisation

Chacun des 16 clicks inverse la valeur d'un sous ensemble de cases.



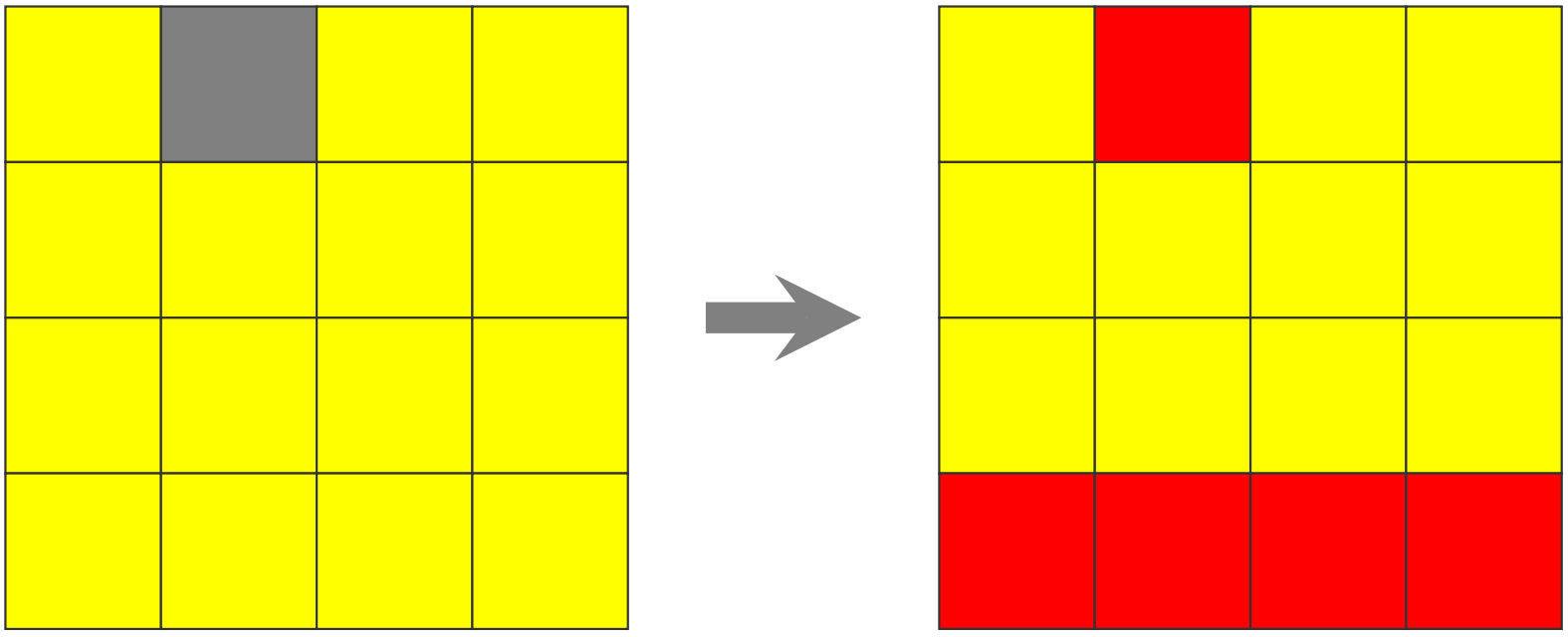
Vecteur de configuration : $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

Puzzle Merlin, modélisation : "click(1,1)"



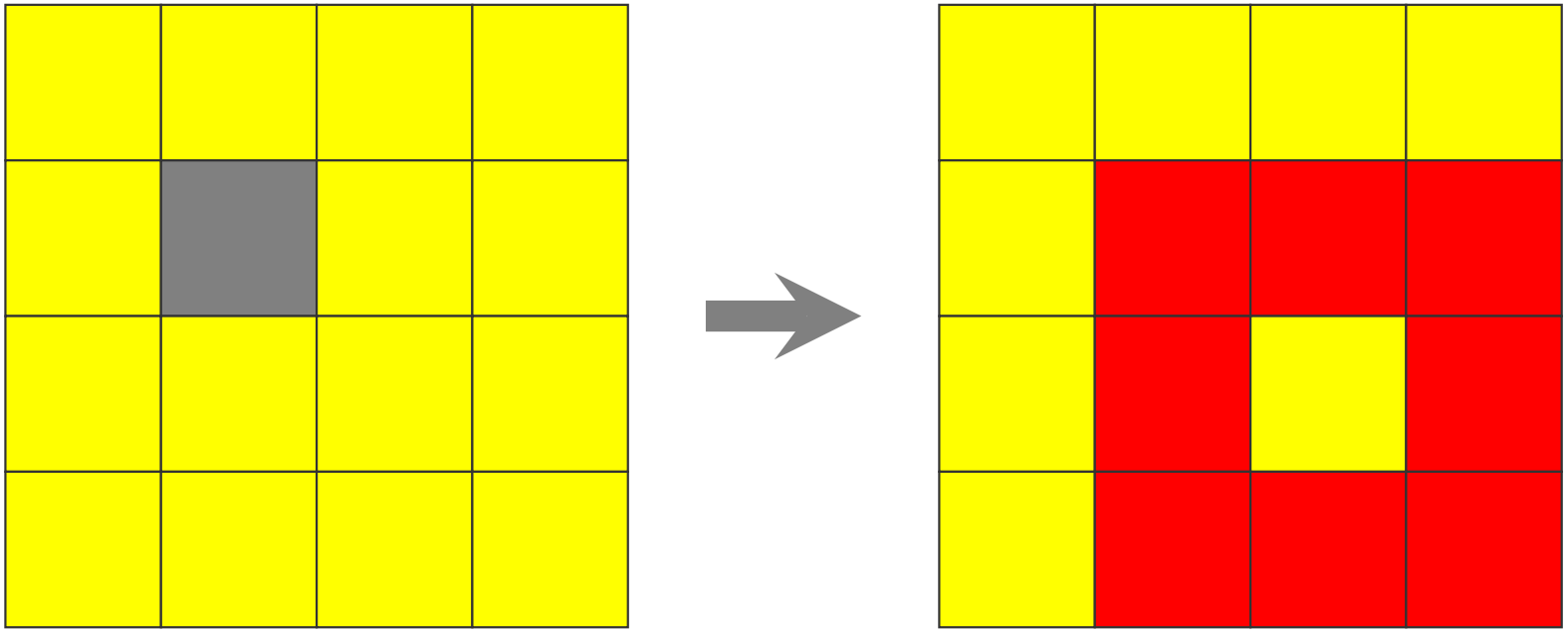
Vecteur de configuration : $(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1)$

Puzzle Merlin, modélisation : "click(1,2)"



Vecteur de configuration : $(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1)$

Puzzle Merlin, modélisation : "click(2,2)"



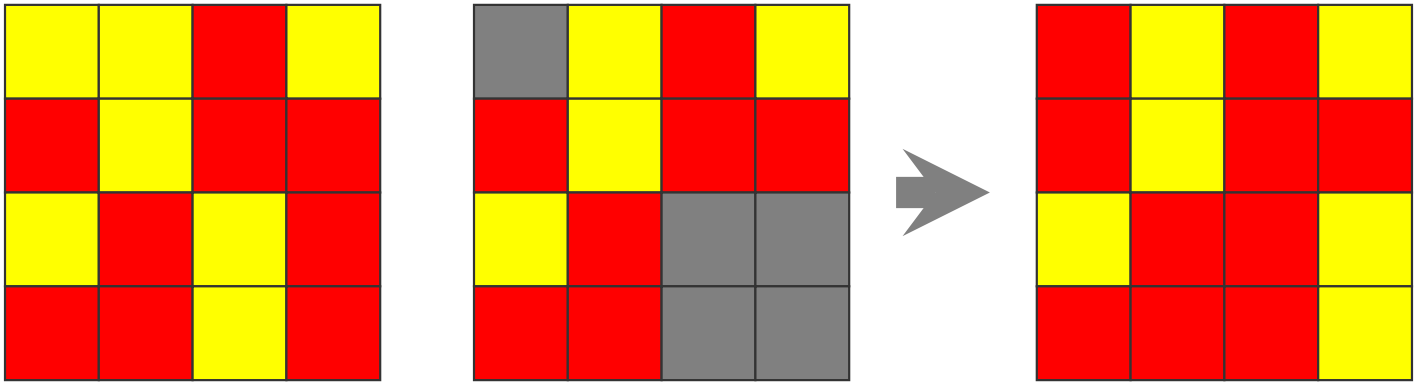
Vecteur de configuration : $(0,0,0,0,0,1,1,1,0,1,1,1,0,1,1,1)$

Les autres clicks se déduisent par rotation de $\frac{\pi}{2}$ et symétrie.

Puzzle Merlin, modélisation

"Cliquer" sur une des 16 cases du puzzle revient à additionner deux vecteurs binaires dans l'espace vectoriel $E = \{\{0,1\}^{16}, +, \times\}$

- $0 + 0 = 0$
- $0 + 1 = 1 + 0 = 1$
- $1 + 1 = 0$

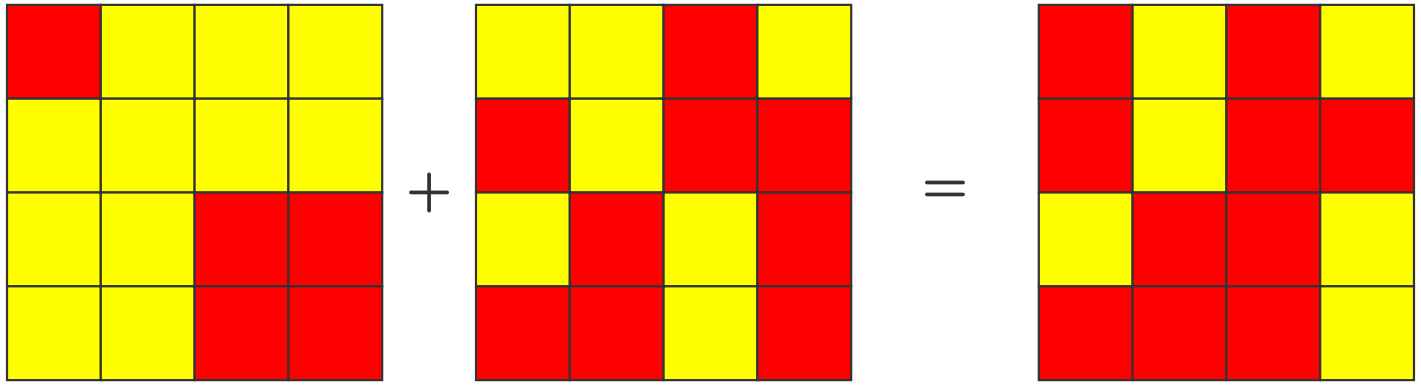


v_1 **click 1** v_2

Puzzle Merlin, modélisation

"Cliquer" sur une des 16 cases du puzzle revient à additionner deux vecteurs binaires dans l'espace vectoriel $E = \{\{0,1\}^{16}, +, \times\}$

- $0 + 0 = 0$
- $0 + 1 = 1 + 0 = 1$
- $1 + 1 = 0$



Puzzle Merlin : algorithme de résolution

Chacun des 16 possibles clicks correspond à une colonne dans la matrice binaire :

$$B = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

- cette matrice est inversible,
- ses colonnes constituent une base de $\{0,1\}^{16}$,
- elles engendrent toutes les configurations possibles du puzzle.

Puzzle Merlin : algorithme de résolution

Trouver la séquences minimale de clicks pour passer d'un vecteur v_0, à un vecteur cible v_1 revient à résoudre le système d'équations linéaires :

$$B \times s = v_1 - v_0 \Rightarrow s = B^{-1} \times (v_1 - v_0).$$

Les composantes non nulles de s déterminent la séquence de clicks.

Cet algorithme appliqué au vecteur $v_0 - v_1$ avec

v_0 =(0,0,1,0,1,0,1,1,0,1,0,1,1,0,1) et v_1 =(0,1,0,1,1,0,1,0,0,1,0,1,1,0,1) donne le vecteur (1,0,0,1,0,1,1,0,0,1,1,1,1,0,0,1).

La réponse à la question posée slide "**E**xemple 1 : Puzzle Merlin" est donc la séquences de 8 clicks : (1,1), (1,4), (2,2), (2,3), (3,2), (3,3), (3,4), (4,1), (4,4).

Puzzle Merlin : algorithme de Gauss-Jordan

```
// 1) RENDRE LA MATRICE M TRIANGULAIRE INFERIEURE : DIM=16 ET mat=B|v1-v0
for (i=0;i<DIM;i++) {
    if (mat[i][i]==0) {          // RECHERCHE D'UN ELEMENT NON NUL (PIVOT)
        for (j=i+1;j<DIM;j++)
            if (mat[j][i]!=0) break;
        if (j>=DIM)                  // matrice irrégulière
            exit(0);
        for (k=0;k<=DIM;k++) {   // ECHANGE DES LIGNES i et j
            val=mat[i][k];
            mat[i][k]=mat[j][k];
            mat[j][k]=val;
        }
    }
    for (j=i+1;j<DIM;j++)
        if (mat[j][i]!=0)
            for (k=i;k<=DIM;k++)
                mat[j][k]=mat[i][k]!=mat[j][k]?1:0;
}
// 2) B DE mat DEVIENT TRIANGULAIRE SUPERIEURE
for (i=DIM-1;i>0;i--) {
    for (j=0;j<i;j++)
        if (mat[j][i]!=0)
            for (k=i;k<=DIM;k++)
                mat[j][k]=mat[i][k]!=mat[j][k]?1:0;
} // ET LA DERNIERE COLONNE DE mat EST LE VECTEUR DE DECISION s RECHERCHE
```

Puzzle Merlin : comlexité de l'algorithme

Les trois boucles imbriquées du code précédent :

1. i de 0 à $DIM - 1$, (ligne 1)
2. j de $i + 1$ à $DIM - 1$, (ligne 13)
3. k de i à DIM, (ligne 15)

font que, dans le pire des cas, nous effectuerons de l'ordre de DIM^3 opérations sur la variable $mat[][]$.

En posant $n = DIM$ la taille des données d'entrée, cet algorithme est donc en $O(n^3)$. Il est de complexité polynomiale (cubique).

Le puzzle Merlin est donc considéré comme **facile** et appartient à la classe **P** des problèmes pour lesquels il existe un algorithme de résolution polynomial.

Problèmes Difficiles

Problèmes pour lesquels il n'existe pas d'algorithme polynomial de résolution.

Les problèmes difficiles de **décision** (existe-t-il une configuration satisfaisant un ensemble de contraintes ?), pour lesquels on peut vérifier polynomialement la solution produite, appartiennent à **NP**.

Exemple : soit un graphe $\mathcal{G} = \{\mathcal{S}, \mathcal{A}\}$, existe-t-il une affectation de valeurs entières à ses sommets telle que :

1. $\forall x \in \mathcal{S} : 1 \leq Col(x) \leq k$,
2. $\forall (x, y) \in \mathcal{A} \Rightarrow Col(x) \neq Col(y)$

Pour $k \geq 3$, le problème de k-**colorabilité** appartient à **NP**.

Exemple 2 : jeu de SUDOKU

Peut-on compléter la grille suivante ?

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

Jeu de SUDOKU : modélisation

Les 81 cases (i, j) sont les sommets d'un graphe.

Il existe une arête entre (i, j) et (i', j') si :

1. $i = i'$: une seule couleur par ligne,
2. $j = j'$: une seule couleur par colonne,
3. $i \equiv_3 i'$ et $j \equiv_3 j'$: une seule couleur par carré disjoint 3×3.

C'est une instance du problème de k-**colorabilité** avec $k = 9$.

Jeu de SUDOKU : énumération exhaustive

Les instances du problème de SUDOKU correspondent à des graphes de petite taille (maximum 81 sommets et de fortes contraintes sur les sommets du graphe à colorer).

A défaut d'un algorithme polynomial il est possible, dans ce cas, d'envisager une énumération exhaustive par recherche arborescente en profondeur d'abord.

Cet algorithme essaiera systématiquement toutes les valeurs possibles pour les sommets (cases de la grille de SUDOKU) libres du graphe.

Jeu de SUDOKU : codage des contraintes

La fonction récursive *coloreCase*(), qui implémente cette recherche exhaustive, utilise les structures de données suivantes :

- LIGNE[10][9] : LIGNE[k][i] $= k$ si la couleur k est présente sur la ligne i, 0 sinon,
- COLONNE[10][9] : COLONNE[k][j] $= k$ si la couleur k est présente sur la colonne j, 0 sinon,
- CARRE[10][3][3] : CARRE[k][$i/3$][$j/3$] $= k$ si la couleur k est présente dans le carré 3×3 de coin supérieur-gauche $(i/3, j/3)$, 0 sinon.

Ces structures permettent de gérer les contraintes avec une **complexité minimale**.

Jeu de SUDOKU : recherche exhaustive

```
void coloreCase(void)
    {
    char i, j, k;
    for (i = 0; i < 9; i++)
        for (j = 0; j < 9; j++)
            if (GRILLE[i][j] == 0)
                { // la case (i,j) est non colorée :
                for (k = 1; k <= 9; k++) // essayer les 9 couleurs
                    if (LIGNE[k][i] == 0 &&
                        COLONNE[k][j] == 0 &&
                        CARRE[k][i / 3][j / 3] == 0)
                        {
                        GRILLE[i][j] = CARRE[k][i / 3][j / 3] =
                             LIGNE[k][i] = COLONNE[k][j] = k;
                         coloreCase();
                        GRILLE[i][j] = CARRE[k][i / 3][j / 3] =
                             LIGNE[k][i] = COLONNE[k][j] = 0;
                        }
                return;
                }
    // Plus de cases non colorées : SOLUTION TROUVEE
    printDamier();
    }
```

Jeu de SUDOKU : complexité de *coloreCase*()

Soit n ce nombre ($n = 58$ pour la grille proposée en exemple), dans le pire des cas la fonction *coloreCase*() pourrait exécuter 9^n appels à elle-même pour explorer toute la combinatoire.

Cet algorithme est d'une complexité exponentielle.

En pratique, la prise en compte des contraintes ainsi que des variables fixées dans l'instance du slide "**E**xemple 2 : jeu de SUDOKU", réduisent significativement cette complexité.

Jeu de SUDOKU : solution de l'Exemple 2

8911 appels récursifs pour prouver la 9-**colorabilité** de cette instance de graphe (**bien spécifique**) de 58 sommets :

```
LAC_Nov2020>solve grille.txt

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

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

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

     8911 bktk 0.0 sec.
```

Conclusion

Du point de vue opérationnel, ce sont les instances des problèmes qu'il faut résoudre :

- problème facile ($\in \mathbf{P}$) ne veut pas dire algorithme simple,
- problème difficile ($\in \mathbf{NP}$) n'implique pas forcément **instance** intraitable.

Remarque : seule la complexité temporelle a été évoquée dans cette présentation.

michel.vasquez@mines-ales.fr