

Deuxième solution au problème

Peut-on faire mieux ? Voici ce que nous pouvons remarquer :

1. Notre algorithme analyse un très grand nombre de cas inutilement puisque le problème stipule qu'il faut utiliser des nombres différents dans chaque case de la matrice, ce qu'on ne valide pas. On procède donc au test de validité **EstUneRéussite** sur plein de matrice qui sont, à l'évidence, mauvaises. Le programme travaille beaucoup pour rien.

Nous allons donc modifier le programme de la manière suivante :

- Nous allons implanter un tableau de 9 cases booléennes qui vont indiquer les nombres utilisés dans la grille solution du carré magique. Ainsi, avant d'affecter à notre grille une valeur, nous pourrions vérifier que cette valeur n'est pas encore utilisée ailleurs dans la grille et ainsi diminuer d'un facteur considérable le nombre de cas à considérer lors de la recherche de la solution

Notre classe CarreMagique est modifiée de la manière suivante (les caractères gras indiquent les ajouts à la classe) :

```
1  #pragma once
2  #include "Matrice.h"
3  #include <chrono>
4  #include <iostream>
5  using namespace std;
6  using namespace std::chrono;
7
8  class CarreMagique
9  {
10     static const int iORDRE = 3;
11     int reponseMagique_;
12     Matrice <int> leCarreMagique;
13     bool tableauChiffresDisponibles[iORDRE * iORDRE];
14     ostream & rOut;
15     system_clock::time_point debut;
16
17 public:
18     CarreMagique(ostream &);
19     void RechercherSolution();
20
21 private:
22     void Afficher() const;
23     bool Normaliser(int & x, int & y) const;
24     void PlacerChiffre(int i, int j);
25
26     bool EstUneReussite() const;
27     // bool SontTousLesChiffresDifférents() Cette méthode est
28     // maintenant devenue inutile
```

```
29     bool SontSommesHorizontalesCorrectes() const;
30     bool SontSommesVerticalesCorrectes() const;
31     bool EstSommeDiagonaleGDCorrecte() const;
32     bool EstSommeDiagonaleDGCorrecte() const;
33
34     // ajout version 2 du Carré magique
35     void BloquerChiffre(int x);
36     void DebloquerChiffre(int x);
37     bool EstUtilisable(int x) const;
38 };
39
40
41 #include "CarreMagique.h"
42 #include <string>
43 using namespace std;
44 using namespace std::chrono;
45
46 CarreMagique::CarreMagique(ostream & out)
47     : rOut(out), leCarreMagique(iORDRE, iORDRE)
48 {
49     reponseMagique_ = (iORDRE)*(iORDRE*iORDRE+ 1) / 2;
50
51     // Initialisation de la matrice
52     for (int i = 0; i < leCarreMagique.GetNbLignes(); ++i)
53         for (int j = 0; j < leCarreMagique.GetNbColonnes(); ++j)
54             {
55                 leCarreMagique[i][j] = 0;
56             }
57
58     // Initialisation du tableau des chiffres disponibles
59     for (int i = 0; i < iORDRE * iORDRE; ++i)
60     {
61         tableauChiffresDisponibles[i] = true;
62     }
63 }
64
65 void CarreMagique::Afficher () const
66 {
67     auto fin = system_clock::now();
68     auto tempsEcoule = fin - debut;
69
70     rOut << "Contenu de la matrice : " << endl;
71     for (int i = 0; i < leCarreMagique.GetNbLignes(); i++)
72     {
73         for (int j = 0; j < leCarreMagique.GetNbColonnes(); j++)
74             {
75                 rOut << leCarreMagique[i][j] << " ";
76             }
77         rOut << endl;
78     }
79     rOut << endl;
80     rOut << "Solution en "
81         << duration_cast<milliseconds>(tempsEcoule).count() << " ms." << endl;
82     rOut << string(69, '-') << endl;
83 }
```

```
84
85 bool CarreMagique::SontSommesHorizontalesCorrectes() const
86 {
87     int somme, i, j;
88
89     // Vérifions les horizontales
90     for (i = 0; i < iORDRE; i++)
91     {
92         somme = 0;
93         for (j = 0; j < iORDRE; j++)
94             somme = somme + leCarreMagique[i][j];
95         if (somme != reponseMagique_)
96             return false;
97     }
98     return true;
99 }
100
101 bool CarreMagique::SontSommesVerticalesCorrectes() const
102 {
103     int somme, i, j;
104
105     // Vérifions les verticales
106     for (i = 0; i < iORDRE; i++)
107     {
108         somme = 0;
109         for (j = 0; j < iORDRE; j++)
110             somme = somme + leCarreMagique[j][i];
111         if (somme != reponseMagique_)
112             return false;
113     }
114     return true;
115 }
116
117 bool CarreMagique::EstSommeDiagonaleGDCorrecte() const
118 {
119     int somme, i;
120
121     // Vérifions la diagonale g - d
122     somme = 0;
123     for (i = 0; i < iORDRE; i++)
124         somme = somme + leCarreMagique[i][i];
125
126     return somme == reponseMagique_;
127 }
128
129 bool CarreMagique::EstSommeDiagonaleDGCorrecte() const
130 {
131     int somme, i;
132
133     // Vérifions la diagonale d - g
134     somme = 0;
135     for (i = iORDRE - 1; i >= 0; i--)
136         somme = somme + leCarreMagique[i][iORDRE - 1 - i];
137
138     return somme == reponseMagique_;
139 }
```

```
140 bool CarreMagique::EstUneReussite() const
141 {
142     return SontSommesHorizontalesCorrectes() && SontSommesVerticalesCorrectes() &&
143         EstSommeDiagonaleGDCorrecte() && EstSommeDiagonaleDGCorrecte();
144 }
145
146 void CarreMagique::BloquerChiffre(int x)
147 {
148     tableauChiffresDisponibles[x - 1] = false;
149 }
150
151 void CarreMagique::DebloquerChiffre(int x)
152 {
153     tableauChiffresDisponibles[x - 1] = true;
154 }
155
156 bool CarreMagique::EstUtilisable(int x) const
157 {
158     return tableauChiffresDisponibles[x - 1];
159 }
160
161 void CarreMagique::PlacerChiffre(int x, int y)
162 {
163     // Quand Normaliser retourne 'faux', toutes les cases sont
164     // comblées et on doit vérifier si la solution est valide.
165     if ( Normaliser(x,y))
166     {
167         for (int z=1; z <= iORDRE*iORDRE; ++z)
168         {
169             if (EstUtilisable(z))
170             {
171                 // Rendre non disponible ce chiffre
172                 BloquerChiffre(z);
173
174                 // Affecter à la position en cours
175                 leCarreMagique[x][y] = z;
176
177                 // Comblé le reste des cases
178                 PlacerChiffre(x, y + 1);
179
180                 // Rendre à nouveau ce chiffre disponible
181                 DebloquerChiffre(z);
182             }
183         }
184         // réinitialiser la case en cours de traitement
185         leCarreMagique[x][y] = 0;
186     }
187     else
188     {
189         if (EstUneReussite())
190         {
191             Afficher();
192         }
193     }
194 }
195
```

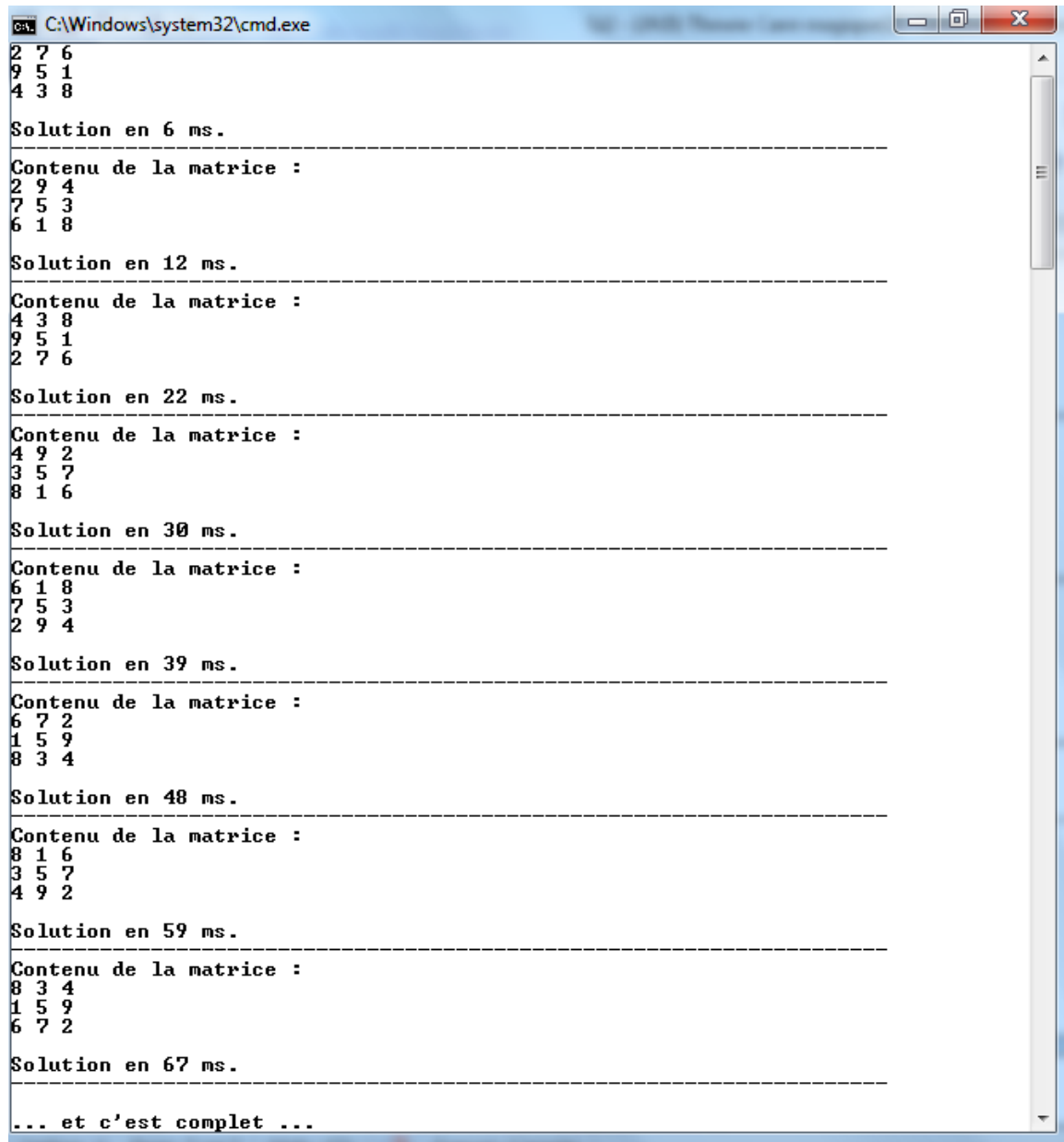
```
196 bool CarreMagique::Normaliser(int & x, int & y) const
197 {
198     if ( x < iORDRE && y < iORDRE) return true;
199     if ( y >= iORDRE)
200     {
201         ++x;
202         y = 0;
203     }
204     if ( x < iORDRE) return true;
205     return false;
206 }
207
208 void CarreMagique::RechercherSolution()
209 {
210     debut = system_clock::now();
211     PlacerChiffre(0,0);
212 }
```

Notre programme principal qui fait appel à la méthode **RechercherSolution** n'a pas changé.

```
230 #include "CarreMagique.h"
231 #include <iostream>
232 using namespace std;
233
234 int main()
235 {
236     CarreMagique oCarreMagique(cout);
237
238     system("Color F0");
239     oCarreMagique.RechercherSolution();
240 }
241
```

KEE – Le problème du carré magique (2)

À l'usage, on se rend compte que cet algorithme **est un très net progrès** puisqu'il trouve la première solution **en 6 millisecondes** et les huit solutions possibles sont trouvées en **67 millisecondes** environ.



```
C:\Windows\system32\cmd.exe
2 7 6
9 5 1
4 3 8
Solution en 6 ms.
-----
Contenu de la matrice :
2 9 4
7 5 3
6 1 8
Solution en 12 ms.
-----
Contenu de la matrice :
4 3 8
9 5 1
2 7 6
Solution en 22 ms.
-----
Contenu de la matrice :
4 9 2
3 5 7
8 1 6
Solution en 30 ms.
-----
Contenu de la matrice :
6 1 8
7 5 3
2 9 4
Solution en 39 ms.
-----
Contenu de la matrice :
6 7 2
1 5 9
8 3 4
Solution en 48 ms.
-----
Contenu de la matrice :
8 1 6
3 5 7
4 9 2
Solution en 59 ms.
-----
Contenu de la matrice :
8 3 4
1 5 9
6 7 2
Solution en 67 ms.
-----
... et c'est complet ...
```

	Programme # 1 un peu bêta	Programme # 2 amélioré
Première solution	1,15 secondes	0,006 secondes
Dernière solution correcte	4,6 secondes	0,067 secondes

Peut-on faire encore mieux ? Oui. Comment ?