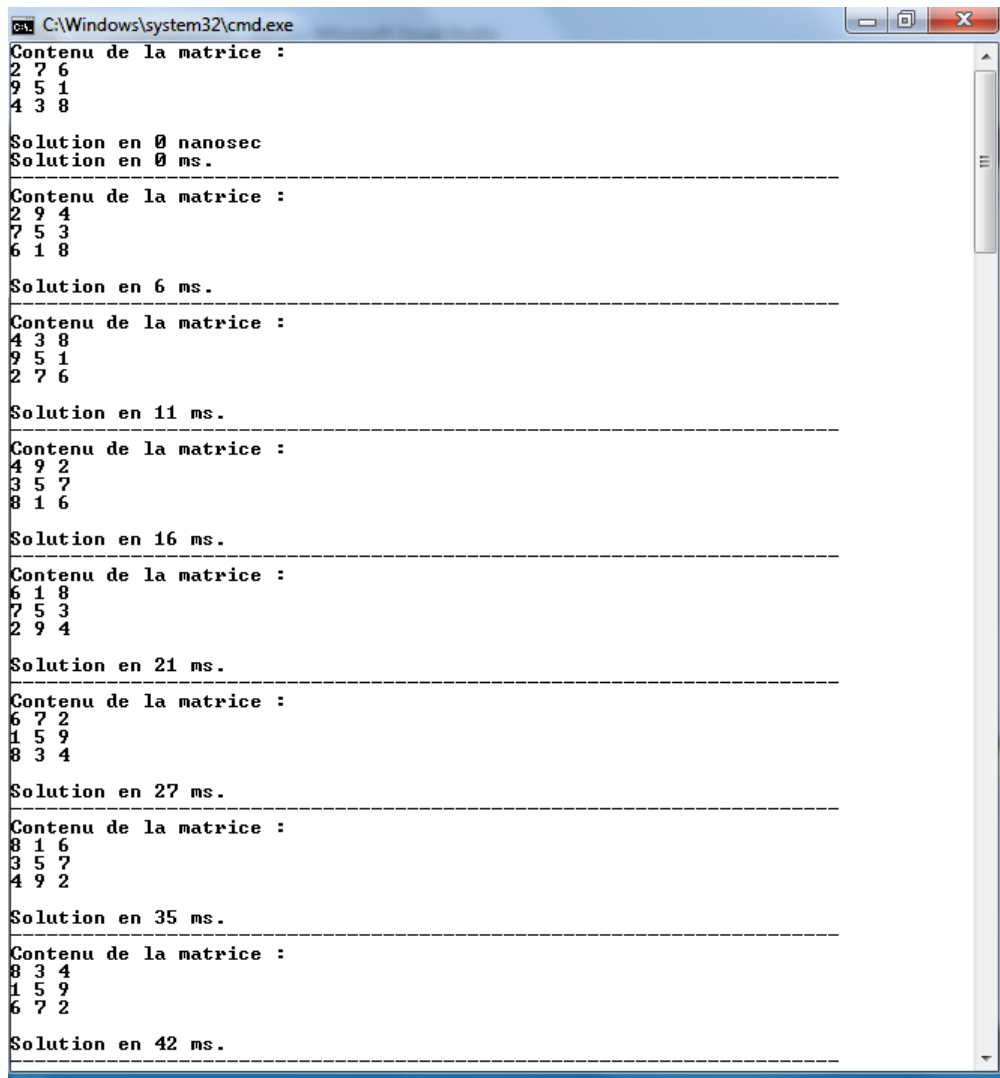


Troisième solution au problème

On se rappelle du tableau suivant :

| | 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 |

Avec quelques légères améliorations à notre algorithme, on aura maintenant :



```
C:\Windows\system32\cmd.exe
Contenu de la matrice :
2 7 6
9 5 1
4 3 8
Solution en 0 nanosec
Solution en 0 ms.
-----
Contenu de la matrice :
2 9 4
7 5 3
6 1 8
Solution en 6 ms.
-----
Contenu de la matrice :
4 3 8
9 5 1
2 7 6
Solution en 11 ms.
-----
Contenu de la matrice :
4 9 2
3 5 7
8 1 6
Solution en 16 ms.
-----
Contenu de la matrice :
6 1 8
7 5 3
2 9 4
Solution en 21 ms.
-----
Contenu de la matrice :
6 7 2
1 5 9
8 3 4
Solution en 27 ms.
-----
Contenu de la matrice :
8 1 6
3 5 7
4 9 2
Solution en 35 ms.
-----
Contenu de la matrice :
8 3 4
1 5 9
6 7 2
Solution en 42 ms.
```

Donc une première solution en moins de **1 nanoseconde** ($< 0,000000001$ seconde) et toutes les solutions trouvées en **42 millisecondes environ**, toujours sur le même ordinateur dans les mêmes conditions qu'auparavant.

Donc

| | Programme # 1 un peu bêta | Programme # 2 amélioré | Programme #3 optimal (?) |
|-------------------|---------------------------|---|---|
| Première solution | 1,15 secondes | 0,006 secondes 192 fois plus rapide que #1 | < 0,000000001 seconde 6 millions de fois plus rapide que #2 et 1150 millions de fois plus rapide que #1 |
| Dernière solution | 4,6 secondes | 0,067 secondes 69 fois plus rapide que #1 | 0,044 secondes 1,5 fois plus rapide que #2 et 105 fois plus rapide que #1 |

Qu'avons-nous changé ? Un tout petit truc, vraiment. Quand une ligne que nous venons de compléter n'a pas pour somme la réponse magique recherchée, nous procédions à remplir toutes les autres cases de la grille pour rien.

C'est une perte de temps importante puisque pour la première ligne, nous savons dès la 4^e case de la matrice si le reste de la recherche est inutile. Pour la deuxième ligne, nous pouvons valider dès la 7^e case si le reste de la recherche a du sens.

On peut donc vérifier la somme à chaque fin de ligne pour s'assurer que nous ne sommes pas en train de travailler en vain.

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     static const int iORDRE_CARRE = iORDRE * iORDRE;
12     int reponseMagique_;
13     Matrice <int> leCarreMagique;
14     bool tableauChiffresDisponibles[iORDRE_CARRE];
15     ostream & rOut;
16     system_clock::time_point debut;
17
18 public:
19     CarreMagique(ostream &);
20     void RechercherSolution();
21
22 private:
23     void Afficher() const;
24     bool Normaliser(int & x, int & y) const;
25     void PlacerChiffre(int i, int j);
26
27     bool EstUneReussite() const;
28     bool SontSommesHorizontalesCorrectes() const;
29     bool SontSommesVerticalesCorrectes() const;
30     bool EstSommeDiagonaleGDCorrecte() const;
31     bool EstSommeDiagonaleDGCorrecte() const;
32
33     // ajout version 2 du Carré magique
34     void BloquerChiffre(int x);
35     void DebloquerChiffre(int x);
36     bool EstUtilisable(int x) const;
37
38     // ajouter version 3 du Carré magique
39     bool SuccesToujoursPossible(int x) const;
40 };
```

Définition de la classe CarreMagique

```
41 #include "CarreMagique.h"
42 #include <string>
43 using namespace std::chrono;
44
45 CarreMagique::CarreMagique(ostream & out)
46     : rOut(out), leCarreMagique(iORDRE, iORDRE)
47 {
48     reponseMagique_ = (iORDRE)*(iORDRE_CARRE + 1) / 2;
49
50     // Initialisation de la matrice
51     for (int i = 0; i < leCarreMagique.GetNbLignes(); ++i)
52         for (int j = 0; j < leCarreMagique.GetNbColonnes(); ++j)
53             {
54                 leCarreMagique[i][j] = 0;
55             }
56
57     // Initialisation du tableau des chiffres disponibles
58     for (int i = 0; i < iORDRE * iORDRE; ++i)
59     {
60         tableauChiffresDisponibles[i] = true;
61     }
62 }
63
64 void CarreMagique::Afficher () const
65 {
66     auto fin = system_clock::now();
67     auto tempsEcoule = fin - debut;
68
69     rOut << "Contenu de la matrice : " << endl;
70     for (int i = 0; i < leCarreMagique.GetNbLignes(); i++)
71     {
72         for (int j = 0; j < leCarreMagique.GetNbColonnes(); j++)
73             {
74                 rOut << leCarreMagique[i][j] << " ";
75             }
76         rOut << endl;
77     }
78     rOut << endl;
79     if (duration_cast<milliseconds>(tempsEcoule).count() < 1)
80         rOut << "Solution en " <<
81             duration_cast<nanoseconds>(tempsEcoule).count() << " nanosec" << endl;
82     rOut << "Solution en " <<
83         duration_cast<milliseconds>(tempsEcoule).count() << " ms." << endl;
84     rOut << std::string(69, '-') << endl;
85 }
86
87
```

```
88 bool CarreMagique::SontSommesHorizontalesCorrectes() const
89 {
90     int somme, i, j;
91
92     // Vérifions les horizontales
93     for (i = 0; i < iORDRE; i++)
94     {
95         somme = 0;
96         for (j = 0; j < iORDRE; j++)
97             somme = somme + leCarreMagique[i][j];
98         if (somme != reponseMagique_)
99             return false;
100     }
101     return true;
102 }
103
104 bool CarreMagique::SontSommesVerticalesCorrectes() const
105 {
106     int somme, i, j;
107
108     // Vérifions les verticales
109     for (i = 0; i < iORDRE; i++)
110     {
111         somme = 0;
112         for (j = 0; j < iORDRE; j++)
113             somme = somme + leCarreMagique[j][i];
114         if (somme != reponseMagique_)
115             return false;
116     }
117     return true;
118 }
119
120 bool CarreMagique::EstSommeDiagonaleGDCorrecte() const
121 {
122     int somme, i;
123
124     // Vérifions la diagonale g - d
125     somme = 0;
126     for (i = 0; i < iORDRE; i++)
127         somme = somme + leCarreMagique[i][i];
128
129     return somme == reponseMagique_;
130 }
131
132 bool CarreMagique::EstSommeDiagonaleDGCorrecte() const
133 {
134     int somme, i;
135
136     // Vérifions la diagonale d - g
137     somme = 0;
138     for (i = iORDRE - 1; i >= 0; i--)
139         somme = somme + leCarreMagique[i][iORDRE - 1 - i];
140
141     return somme == reponseMagique_;
142 }
143
```

```
144 bool CarreMagique::EstUneReussite() const
145 {
146     return SontSommesHorizontalesCorrectes() && SontSommesVerticalesCorrectes() &&
147         EstSommeDiagonaleGDCorrecte() && EstSommeDiagonaleDGCorrecte();
148 }
149
150 void CarreMagique::BloquerChiffre(int x)
151 {
152     tableauChiffresDisponibles[x - 1] = false;
153 }
154
155 void CarreMagique::DebloquerChiffre(int x)
156 {
157     tableauChiffresDisponibles[x - 1] = true;
158 }
159
160 bool CarreMagique::EstUtilisable(int x) const
161 {
162     return tableauChiffresDisponibles[x - 1];
163 }
164
165 bool CarreMagique::SuccesToujoursPossible(int x) const
166 {
167     int somme = 0;
168     if (x > 0) // on vérifie la somme de la ligne précédente pour valider
169     {          // qu'on ne continue pas pour rien
170         for (int y = 0; y < iORDRE; y++)
171             somme = somme + leCarreMagique[x - 1][y];
172         if (somme != reponseMagique_) return false;
173     }
174     return true;
175 }
176
177 void CarreMagique::PlacerChiffre(int x, int y)
178 {
179     // Quand Normaliser retourne 'faux', toutes les cases sont
180     // comblées et on doit vérifier si la solution est valide.
181     if (Normaliser(x, y) && SuccesToujoursPossible(x))
182     {
183         for (int z = 1; z <= iORDRE_CARRE; ++z)
184         {
185             if (EstUtilisable(z))
186             {
187                 // Rendre non disponible ce chiffre
188                 BloquerChiffre(z);
189
190                 // Affecter à la position en cours
191                 leCarreMagique[x][y] = z;
192
193                 // Comblé le reste des cases
194                 PlacerChiffre(x, y + 1);
195
196                 // Rendre à nouveau ce chiffre disponible
197                 DebloquerChiffre(z);
198             }
199         }
```

```
200         // réinitialiser la case en cours de traitement
201         leCarreMagique[x][y] = 0;
202     }
203     else
204     {
205         if (EstUneReussite())
206         {
207             Afficher();
208         }
209     }
210 }
211
212 bool CarreMagique::Normaliser(int & x, int & y) const
213 {
214     if ( x < iORDRE && y < iORDRE) return true;
215     if ( y >= iORDRE)
216     {
217         ++x;
218         y = 0;
219     }
220     if ( x < iORDRE) return true;
221     return false;
222 }
223
224 void CarreMagique::RechercherSolution()
225 {
226     debut = system_clock::now();
227     PlacerChiffre(0,0);
228 }
```

Notez qu'on a eu un gain de temps **non pas** par le retrait d'un certain nombre de choses mais bien **par l'ajout d'instructions** à l'endroit et au moment propice pour s'éviter du traitement inutile.

Peut-on faire encore mieux ? Je pense que oui. Avez-vous des idées ?