Quelle est la nature du problème

C'est un problème classique. Il s'agit de placer, dans une matrice de 3 lignes par 3 colonnes, les nombres de 1 à 9 une seule fois chacun afin que la somme soit identique dans chaque ligne, chaque colonne et chaque diagonale.

Voici un exemple de carré magique :

2	7	6
9	5	1
4	3	8

On observe que la somme de chaque ligne donne 15, même chose pour les colonnes et même chose pour les deux diagonales.

Première solution au problème

Première idée que nous pouvons développer : utiliser **la force brute** de l'appareil. Pour chaque case de la grille, nous allons essayer tous les nombres de 1 à 9. Ainsi, nous essaierons d'abord les combinaisons :

1	1	1
1	1	1
1	1	1

1	1	1
1	1	1
1	1	2

1	1	1
1	1	1
1	1	3

Et ainsi de suite...

Nous allons donc créer la classe CarreMagique qui permettra de créer un objet représentant le carré magique. Cet objet devra bien sûr être en mesure de s'afficher. Nous utiliserons dans cet objet une composition avec la classe Matrice que nous avons vue en début de session. La déclaration prendra la forme suivante :

```
#pragma once
    #include "Matrice.h"
   #include <chrono>
   #include <iostream>
    using namespace std;
    using namespace std::chrono;
7
    class CarreMagique
9
10
          static const int iORDRE;
          static const int iREPONSE MAGIQUE;
11
12
          Matrice <int> leCarreMagique;
13
          ostream & rOut;
          system clock::time point debut;
14
15
16
    public:
          CarreMagique(ostream &);
17
18
          void RechercherSolution();
19
20
    private:
21
          void Afficher() const;
          bool Normaliser(int & x, int & y) const;
22
          void PlacerChiffre(int i, int j);
23
24
25
          bool EstUneReussite() const;
          bool SontTousLesChiffresDifferents() const;
26
27
          bool SontSommesHorizontalesCorrectes() const;
          bool SontSommesVerticalesCorrectes() const;
28
29
          bool EstSommeDiagonaleGDCorrecte() const;
30
          bool EstSommeDiagonaleDGCorrecte() const;
31
    } ;
```

Essentiellement, on a une matrice d'entier qui sera instanciée à la construction de l'objet CarreMagique et cette matrice sera de 3 lignes par 3 colonnes. La classe dévoile un constructeur ainsi qu'une méthode RechercherSolution (). Voici le code de la classe :

Définition de la classe CarreMagique

```
#include "CarreMagique.h"
32
33
    #include <string>
34
    // initialisation des constantes de classe
35
    const int CarreMagigue::iORDRE = 3;
36
37
    const int CarreMagique::iREPONSE MAGIQUE = 15;
38
    CarreMagique::CarreMagique(ostream & out)
39
40
          : rOut(out), leCarreMagique(iORDRE, iORDRE)
41
          for (int i = 0; i < leCarreMagique.GetNbLignes(); ++i)</pre>
42
                for (int j = 0; j < leCarreMagigue.GetNbColonnes(); ++j)</pre>
43
44
                leCarreMagique[i][j] = 0;
45
46
47
48
49
```

```
50
    void CarreMagique::Afficher () const
51
          auto fin = system clock::now();
          auto tempsEcoule = fin - debut;
52
53
54
          rOut << "Contenu de la matrice : " << endl;
          for (int i = 0; i < leCarreMagigue.GetNbLignes(); i++)
55
                for (int j = 0; j < leCarreMagique.GetNbColonnes(); j++)</pre>
56
57
                     rOut << leCarreMagigue[i][j] << " ";</pre>
58
59
60
                rOut << endl;
61
62
          rOut << endl;
          rOut << "Solution en " << duration cast<milliseconds>(tempsEcoule).count() << " ms." << endl;</pre>
63
          rOut << std::string(69, '-') << endl;
64
65
66
67
    bool CarreMagique::SontTousLesChiffresDifferents() const
68
69
          int i, j;
70
71
          // Vérifions qu'on a des chiffres différents dans toutes les cases de la matrice
72
          bool bTab[] = {false, false, false, false, false, false, false, false, false};
73
          for (i = 0; i < iORDRE; ++i)
               for (j = 0; j < iORDRE; ++j)
74
75
                     if (!bTab[leCarreMagique[i][j] - 1])
76
                           bTab[leCarreMagique[i][j] - 1] = true;
77
78
                     else
79
                           return false;
80
81
82
          return true;
83
```

```
84
85
     bool CarreMagique::SontSommesHorizontalesCorrectes() const
86
           int somme, i, j;
87
88
89
           // Vérifions les horizontales
           for (i = 0; i < iORDRE; i++)
90
91
                 somme = 0;
92
                for (j = 0; j < iORDRE; j++)
93
                      somme = somme + leCarreMagique[i][j];
94
                if (somme != iREPONSE MAGIQUE)
95
                      return false;
96
97
98
           return true;
99
100
     bool CarreMagique::SontSommesVerticalesCorrectes() const
101
102
103
           int somme, i, j;
104
           // Vérifions les verticales
105
           for (i = 0; i < iORDRE; i++)
106
107
                 somme = 0;
108
                for (j = 0; j < iORDRE; j++)
109
                      somme = somme + leCarreMagique[j][i];
110
                if (somme != iREPONSE MAGIQUE)
111
                      return false;
112
113
114
           return true;
115
116
117
```

```
bool CarreMagique::EstSommeDiagonaleGDCorrecte() const
118
119
120
           int somme, i;
121
           // Vérifions la diagonale q - d
122
123
           somme = 0;
           for (i = 0; i < iORDRE; i++)
124
                somme = somme + leCarreMagique[i][i];
125
126
           return somme == iREPONSE MAGIQUE;
127
128
129
     bool CarreMagique::EstSommeDiagonaleDGCorrecte() const
130
131
132
           int somme, i;
133
           // Vérifions la diagonale d - q
134
           somme = 0;
135
           for (i = iORDRE - 1; i >= 0; i--)
136
                 somme = somme + leCarreMagique[i][iORDRE - 1 - i];
137
138
           return somme == iREPONSE MAGIQUE;
139
140
141
     bool CarreMagique::EstUneReussite() const
142
143
144
                      SontTousLesChiffresDifferents() && SontSommesHorizontalesCorrectes() &&
           return
                      SontSommesVerticalesCorrectes() && EstSommeDiagonaleGDCorrecte() &&
145
                      EstSommeDiagonaleDGCorrecte();
146
147
148
```

```
void CarreMagique::PlacerChiffre(int x, int y)
149
          //Si on veut une trace de l'exécution, on peut mettre les deux instructions suivantes
150
151
           //mais le chrono sera naturellement inexact.
           //system("cls");
152
          //Afficher();
153
154
155
           // Quand Normaliser retourne 'faux', toutes les cases sont comblées et on doit vérifier
156
           // si la solution est valide.
           if ( Normaliser(x,y))
157
158
159
                for (int z=1; z \le iORDRE*iORDRE; ++z)
                     // Placer un nombre de 1 à 9 à tour de rôle dans la case en cours de traitement
160
                      leCarreMagique[x][y] = z;
161
162
163
                      // Combler le reste des cases
                      PlacerChiffre(x, y+1);
164
165
                // réinitialiser la case en cours de traitement
166
                leCarreMagique[x][y] = 0;
167
168
169
           else
170
                if (EstUneReussite())
171
172
173
                      Afficher();
174
175
176
177
```

```
bool CarreMagique::Normaliser(int & x, int & y) const
178
179
180
           if (x < iORDRE && y < iORDRE) return true;</pre>
181
           if (y >= iORDRE)
182
183
                 ++x;
                 y = 0;
184
185
186
           if (x < iORDRE) return true;</pre>
           return false;
187
188
189
     void CarreMagique::RechercherSolution()
190
191
192
           debut = system clock::now();
           PlacerChiffre (0,0);
193
194
```

195

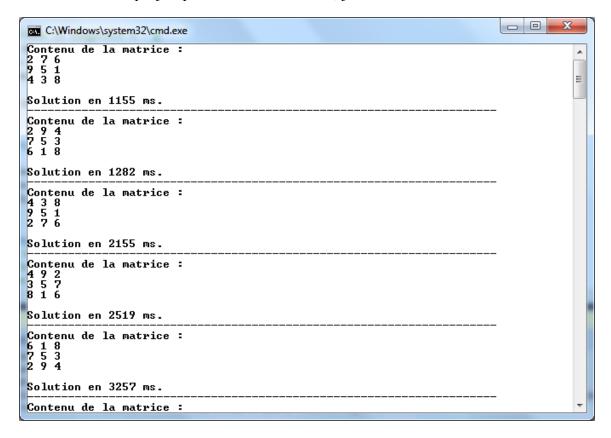
Notre programme main prend la forme suivante :

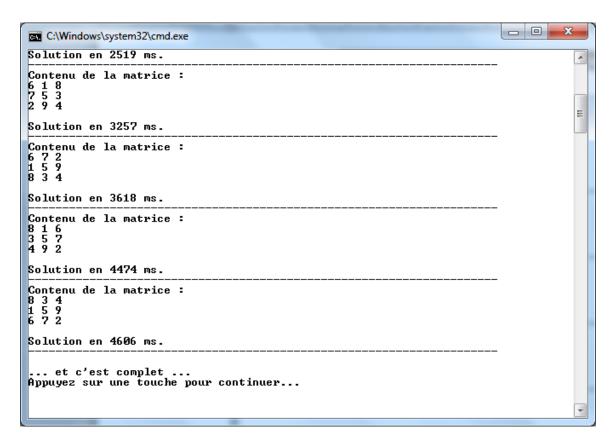
```
#include "CarreMagique.h"
195
196
     #include <iostream>
     using namespace std;
197
198
199
     int main()
200
201
          CarreMagique oCarreMagique(cout);
202
203
          system("Color F0");
204
          oCarreMagique.RechercherSolution();
205
          cout << endl << "... et c'est complet ..." << endl;</pre>
206
207
208
```

Donc, le programme principal appelle une méthode RechercherSolution qui prend en note la valeur de l'horloge et lance PlacerChiffre qui s'appelle récursivement pour rechercher une solution. La méthode EstUneReussite() vérifie si on a bel et bien un carré magique et si oui, on calcule le temps écoulé et on affiche la solution.

À l'usage, on se rend compte que cet algorithme n'est pas efficace puisque

- Avec la machine que j'ai présentement au bureau, j'obtiens les résultats suivants :





- Donc j'obtiens un premier résultat en 1,15 secondes et le dernier en 4,6 secondes en mode release
- Notez que les machines s'améliorent : il me fallait avec l'ordinateur précédent que j'avais au bureau, **10,76** secondes pour la première solution et **46,1** secondes pour la dernière solution. Deux ordinateurs plus tôt, un Pentium IV de 2,8 gigahertz, j'obtenais la première solution en **1672 secondes** environ, c'est-à-dire **27,9** minutes...

Donc, dans l'état actuel des choses, il nous faut quand même **5 secondes** environ pour trouver toutes les solutions au problème. Ce n'est quand même pas mal compte tenu de l'algorithme de force brute. *Peut-on faire mieux*?