

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 :

```

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;
11     static const int iREPONSE_MAGIQUE;
12     Matrice <int> leCarreMagique;
13     ostream & rOut;
14     system_clock::time_point debut;
15
16 public:
17     CarreMagique(ostream &);
18     void RechercherSolution();
19
20 private:
21     void Afficher() const;
22     bool Normaliser(int & x, int & y) const;
23     void PlacerChiffre(int i, int j);
24
25     bool EstUneReussite() const;
26     bool SontTousLesChiffresDifferentes() const;
27     bool SontSommesHorizontalesCorrectes() const;
28     bool SontSommesVerticalesCorrectes() const;
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

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

```

50 void CarreMagique::Afficher () const
51 {     auto fin = system_clock::now();
52     auto tempsEcoule = fin - debut;
53
54     rOut << "Contenu de la matrice : " << endl;
55     for (int i = 0; i < leCarreMagique.GetNbLignes(); i++)
56     {     for (int j = 0; j < leCarreMagique.GetNbColonnes(); j++)
57         {
58             rOut << leCarreMagique[i][j] << " ";
59         }
60         rOut << endl;
61     }
62     rOut << endl;
63     rOut << "Solution en " << duration_cast<milliseconds>(tempsEcoule).count() << " ms." << endl;
64     rOut << std::string(69, '-') << endl;
65 }
66
67 bool CarreMagique::SontTousLesChiffresDiffereents() 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, false};
73     for (i = 0; i < iORDRE; ++i)
74     {     for (j = 0; j < iORDRE; ++j)
75         {
76             if (!bTab[leCarreMagique[i][j] - 1])
77                 bTab[leCarreMagique[i][j] - 1] = true;
78             else
79                 return false;
80         }
81     }
82     return true;
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 != iREPONSE_MAGIQUE)
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 != iREPONSE_MAGIQUE)
112             return false;
113     }
114     return true;
115 }
116
117
```

```

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

```

```

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

```

```

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

```


195

Notre programme main prend la forme suivante :

```

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

- Si on fait une approximation grossière du nombre d'appels récursifs pour tester tous les cas possibles, on peut dire qu'on aurait environ $(9 * 9 * 9 * 9 * 9 * 9 * 9 * 9 * 9)$ appels avant d'avoir testé tous les cas possibles. C'est 387 420 489 appels récursifs.
- Avec la machine que j'ai présentement au bureau, j'obtiens les résultats suivants :

```
C:\Windows\system32\cmd.exe
Contenu de la matrice :
2 7 6
9 5 1
4 3 8
Solution en 1155 ms.
-----
Contenu de la matrice :
2 9 4
7 5 3
6 1 8
Solution en 1282 ms.
-----
Contenu de la matrice :
4 3 8
9 5 1
2 7 6
Solution en 2155 ms.
-----
Contenu de la matrice :
4 9 2
3 5 7
8 1 6
Solution en 2519 ms.
-----
Contenu de la matrice :
6 1 8
7 5 3
2 9 4
Solution en 3257 ms.
-----
Contenu de la matrice :
```

```
C:\Windows\system32\cmd.exe
Solution en 2519 ms.
-----
Contenu de la matrice :
6 1 8
7 5 3
2 9 4

Solution en 3257 ms.
-----
Contenu de la matrice :
6 7 2
1 5 9
8 3 4

Solution en 3618 ms.
-----
Contenu de la matrice :
8 1 6
3 5 7
4 9 2

Solution en 4474 ms.
-----
Contenu de la matrice :
8 3 4
1 5 9
6 7 2

Solution en 4606 ms.
-----
... et c'est complet ...
Appuyez sur une touche pour continuer...
```

- 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 ?*