Matrices de Hadamard

In [1]:
```
import matplotlib.pyplot as plt
import random
import residus
%matplotlib inline
```

In [2]:
```
def show_mat(A):
    plt.matshow(A, cmap=plt.cm.Oranges)
    plt.show()
```

Une matrice A de taille n est dite de Hadamard lorsque ses coefficients sont égaux à ± 1 et ses colonnes (ou lignes) sont orthogonales. Cela revient à dire que $B = \frac{1}{\sqrt{n}} A \in \mathcal{O}_n(\mathbb{R})$ et a tous ses coefficients égaux en valeur absolue.

On a le résultat suivant : pour tout $n \geq 3$, s'il existe une matrice de Hadamard de taille n alors n est un multiple de 4. Que dire de la réciproque ?

Conjecture (Hadamard) : pour tout entier n multiple de 4 il existe une matrice de Hadamard de taille n.

Preuve : personne ne l'a prouvé pour l'instant.

On sait tout de même construire des matrices de Hadamard de taille n pour certaines valeurs de n. Nous allons passer en revue quelques cas intéressants.

1 Une matrice est-elle de Hadamard ?

La fonction `check_hadamard` teste en temps $O(n^3)$ si la matrice A de taille n est bien une matrice de Hadamard.

In [3]:
```
def prod_scal(u, v):
    s = 0
    n = len(u)
    for k in range(n):
        s = s + u[k] * v[k]
    return s
```

In [4]:
```
def check_hadamard(A):
    b = True
    n = len(A)
    for i in range(n):
        for j in range(n):
            b = b and (A[i][j] ** 2 == 1)
    for j in range(n):
        for k in range(n):
            if j != k:
                b = b and prod_scal(A[j], A[k]) == 0
    return b
```

In [5]:
```
check_hadamard([[1, 1],[1, -1]])
```

Out[5]:
```
True
```

In [6]:
```
check_hadamard([[1, 1], [1, 1]])
```

Out[6]:
```
False
```

2 Produit tensoriel de deux matrices - Matrices de Hadamard de taille 2^n

Soit A une matrice de taille m. Soit B une matrice de taille n. Le produit tensoriel de A et B est la matrice $A \otimes B$ de taille mn définie par blocs dont la $i$ème ligne de blocs, $i = 1, \dots, m$, est $(a_{i1}B \dots a_{im}B)$

In [7]:
```
def tensor_product(A, B):
    m = len(A)
    n = len(B)
    p = m * n
    C = [[0 for i in range(p)] for j in range(p)]
    for i in range(m):
        for j in range(m):
            for k in range(n):
                for l in range(n):
                    C[n * i + k][n * j + l] = A[i][j] * B[k][l]
    return C
```

In [8]:
```
tensor_product([[1, 2], [3, 4]], [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

Out[8]:
```
[[1, 2, 3, 2, 4, 6],
 [4, 5, 6, 8, 10, 12],
 [7, 8, 9, 14, 16, 18],
 [3, 6, 9, 4, 8, 12],
 [12, 15, 18, 16, 20, 24],
 [21, 24, 27, 28, 32, 36]]
```

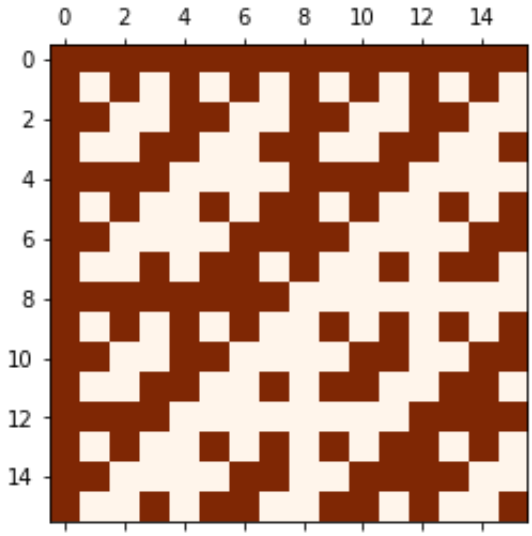
Si A et B sont deux matrices de Hadamard, alors $A \otimes B$ est encore une matrice de Hadamard. Comme il est évident de fabriquer des matrices de Hadamard de tailles 1 et 2 on en déduit un algorithme pour créer des matrices de Hadamard de taille 2^n , ceci pour tout entier naturel n. Les matrices obtenues s'appellent les matrices de Sylvester-Hadamard.

In [9]:
```
def sylvester(n):
    S = [[1, 1],[1, -1]]
    A = [[1]]
    for k in range(n):
        A = tensor_product(S, A)
    return A
```

In [10]:
```
sylvester(1)
```

Out[10]:
```
[[1, 1], [1, -1]]
```

In [12]:
```
M = sylvester(4)
show_mat(M)
check_hadamard(M)
```



Out[12]:
```
True
```

3 Mélanges

Si A est une matrice de Hadamard, toute matrice obtenue en échangeant des lignes ou des colonnes de A est encore une matrice de Hadamard.

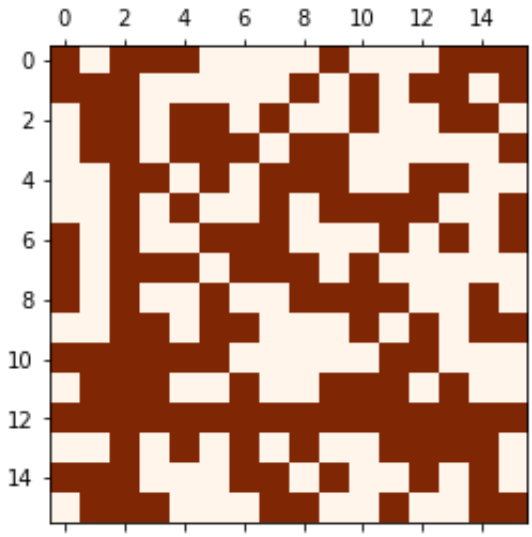
In [13]:
```
def transpose(A):
    m = len(A)
    n = len(A[0])
    B = [[0 for i in range(n)] for j in range(m)]
    for i in range(n):
        for j in range(m):
            B[i][j] = A[j][i]
    return B
```

In [14]:
```
transpose([[1, 2], [3, 4]])
```

Out[14]:
```
[[1, 3], [2, 4]]
```

In [15]:
```
def shuffle(A):
    random.shuffle(A)
    A = transpose(A)
    random.shuffle(A)
    A = transpose(A)
    return A
```

In [16]:
```
M = shuffle(sylvester(4))
show_mat(M)
check_hadamard(M)
```



Out[16]:
```
True
```

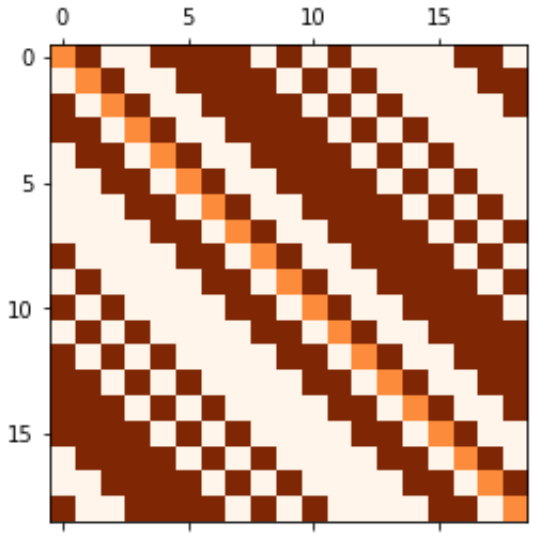
4 Matrices de Paley-Hadamard

Soit q un nombre premier impair. On note χ l'application qui à x non multiple de q associe 1 si x est un carré dans $(\mathbb{Z}/q\mathbb{Z})^*$ et -1 sinon. On prolonge χ en posant $\chi(x) = 0$ si x est multiple de q. On note Q la matrice dont les coefficients sont les $\chi(i-j)$ pour $0 \leq i,j < q$.

Le module `residus` (non montré ici) contient quelques fonctions arithmétiques, et en particulier la fonction χ.

In [17]:
```
def mat_q(q):
    return [[residus.chi(q, (i - j) % q) for i in range(q)] for j in r
```

In [19]:
```
show_mat(mat_q(19))
```

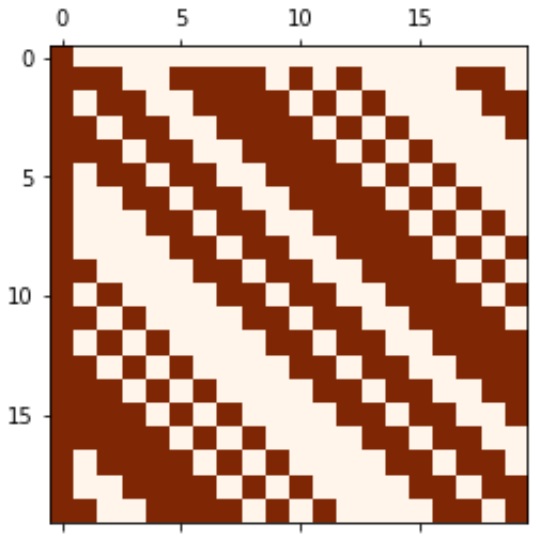


3.1 Nombre premier congru à 3 modulo 4

Si q est congru à 3 modulo 4 on peut construire une matrice de Hadamard de taille $q+1$:

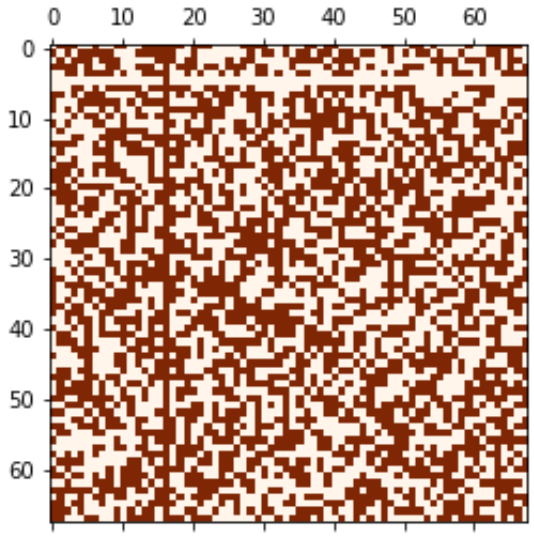
In [20]:
```
def paley1(q):
    Q = mat_q(q)
    P = [[0 for i in range(q + 1)] for j in range(q + 1)]
    P[0][0] = 1
    for j in range(1, q + 1): P[0][j] = -1
    for i in range(1, q + 1): P[i][0] = 1
    for i in range(1, q + 1):
        for j in range(1, q + 1):
            if i == j: P[i][j] = 1
            else: P[i][j] = Q[i - 1][j - 1]
    return P
```

In [21]:
```
M = paley1(19)
show_mat(M)
check_hadamard(M)
```



Out[21]: True

In [22]:
```
M = shuffle(paley1(67))
show_mat(M)
check_hadamard(M)
```



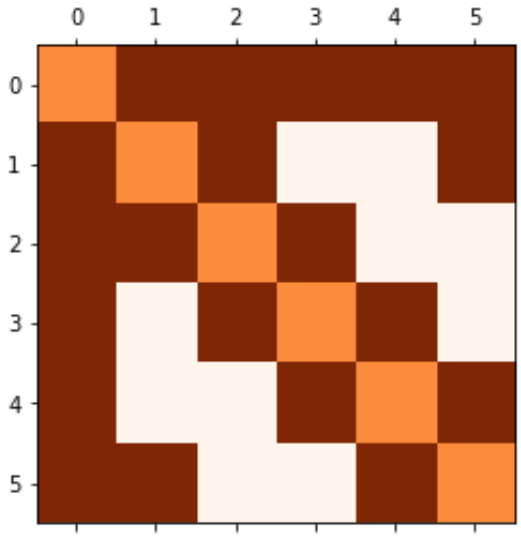
Out[22]: True

3.2 Nombre premier congru à 1 modulo 4

Si q est congru à 1 modulo 4 on peut construire une matrice de Hadamard de taille $2q + 2$:

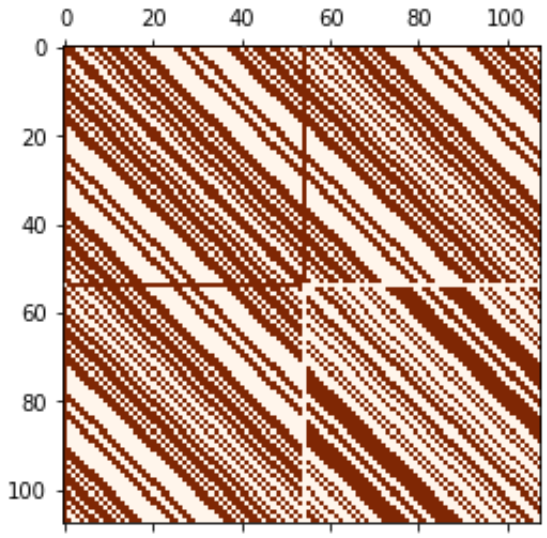
In [23]:
```
def mat_s(q):
    Q = mat_q(q)
    P = [[0 for i in range(q + 1)] for j in range(q + 1)]
    P[0][0] = 0
    for j in range(1, q + 1): P[0][j] = 1
    for i in range(1, q + 1): P[i][0] = 1
    for i in range(1, q + 1):
        for j in range(1, q + 1):
            P[i][j] = Q[i - 1][j - 1]
    return P
```

In [24]:
```
show_mat(mat_s(5))
```



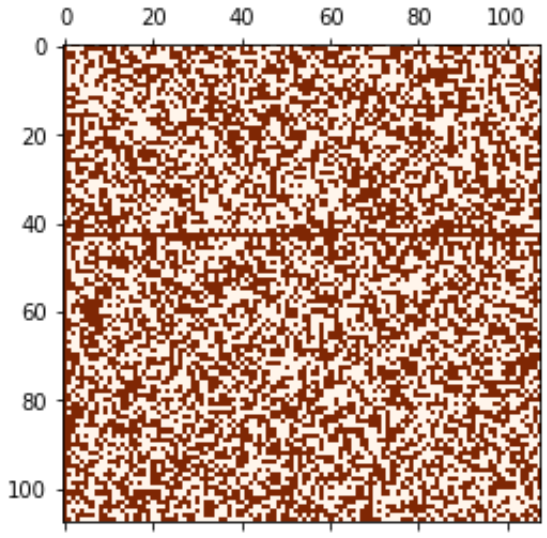
In [26]:
```
def paley2(q):
    S = mat_s(q)
    P = [[0 for i in range(2 * q + 2)] for j in range(2 * q + 2)]
    for i in range(q + 1):
        for j in range(q + 1):
            if i == j: P[i][j] = 1
            else: P[i][j] = S[i][j]
            if i == j: P[i][j + q + 1] = -1
            else: P[i][j + q + 1] = S[i][j]
            if i == j: P[i + q + 1][j] = -1
            else: P[i + q + 1][j] = S[i][j]
            if i == j: P[i + q + 1][j + q + 1] = -1
            else: P[i + q + 1][j + q + 1] = -S[i][j]
    return P
```

In [27]:
```
M = paley2(53)
show_mat(M)
check_hadamard(M)
```



Out[27]: True

In [28]:
```
M = shuffle(paley2(53))
show_mat(M)
check_hadamard(M)
```



Out[28]: True

5 Bilan (provisoire)

On peut regrouper les résultats ci-dessus. Nous savons fabriquer des matrices de Hadamard pour un certain nombre de tailles, en l'occurence pour tout entier qui est :

- une puissance de 2
- de la forme $q+1$ où q est premier congru à 3 modulo 4
- de la forme $2q+2$ où q est premier congru à 1 modulo 4

Nous appellerons (provisoirement) *admissibles* les entiers qui ont l'une des formes ci-dessus. Notre but ultime, il ne sera d'aiileurs pas atteint, est de déclarer admissibles tous les multiples de 4 !

In [29]:
```
def admissible0(n):
    if n <= 2: return True
    if n % 4 != 0: return False
    k = residus.log2(n)
    if k != -1: return True
    q = n - 1
    if residus.is_prime(q) and q % 4 == 3: return True
    q = n // 2 - 1
    if residus.is_prime(q) and q % 4 == 1: return True
    return False
```

Quels sont les multiples de 4 inférieurs à 100 que nous ne savons pas traiter ?

In [30]:
```
[n for n in range(101) if n % 4 == 0 and not admissible0(n)]
```

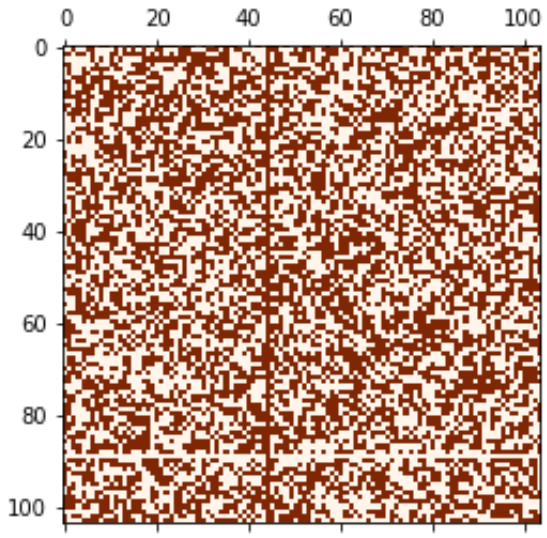
Out[30]:
```
[40, 52, 56, 88, 92, 96, 100]
```

Et voici une fonction provisoire renvoyant une matrice de Hadamard de taille n. Pour tout n admissible, cela va de soi.

In [31]:
```
def hadamard0(n):
    if n > 2 and n % 4 != 0: return False
    k = residus.log2(n)
    if k != -1: return sylvester(k)
    q = n - 1
    if residus.is_prime(q) and q % 4 == 3: return paley1(q)
    q = n // 2 - 1
    if residus.is_prime(q) and q % 4 == 1: return paley2(q)
    return False

def random_hadamard0(n):
    return shuffle(hadamard0(n))
```

In [32]:
```
#M = hadamard0(76)
M = random_hadamard0(104)
show_mat(M)
check_hadamard(M)
```



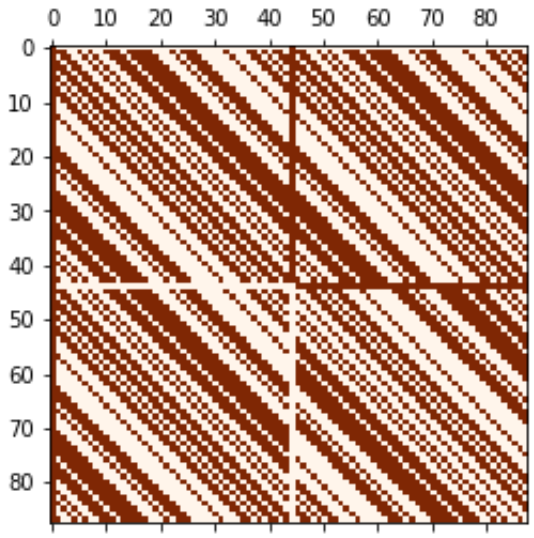
Out[32]: True

6 Allons plus loin

6.1 Une remarque

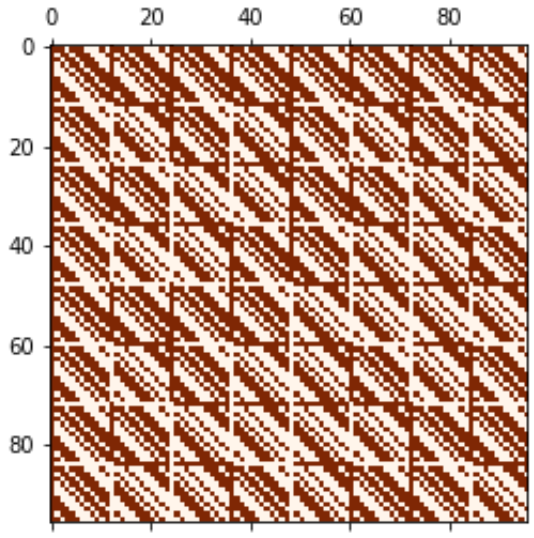
Remarquons que nous sommes capables de créer des matrices de Hadamard de tailles 88 et 96, puisque $88 = 2 \times 44$ et $96 = 8 \times 12$.

In [33]:
```
M88 = tensor_product(hadamard0(2), hadamard0(44))
show_mat(M88)
check_hadamard(M88)
```



Out[33]: True

In [34]:
```
M96 = tensor_product(hadamard0(8), hadamard0(12))
show_mat(M96)
check_hadamard(M96)
```



Out[34]: True

6.2 Entiers admissibles (bis)

On peut automatiser le procédé. Soit $n \in \mathbb{N}^*$. On détermine s'il existe une décomposition de n en produit de nombres admissibles. Si c'est le cas, on fait le produit des matrices de Hadamard pour ces nombres, c'est une matrice de Hadamard de taille n.

La première fonction ci-dessous détermine toutes les décompositions de n en produits de nombres admissibles. La seconde choisit parmi ces décompositions (s'il y en a) celle qui a la plus petite longueur. Elle renvoie la liste vide s'il n'y en a pas.

In [35]:
```
def forall(s, prop):
    for x in s:
        if not prop(x): return False
    return True

def decomp_admissibles(n):
    ps = residus.partitions(n)
    ss = []
    for p in ps:
        if forall(p, admissible0): ss += [p]
    return ss

def minimal_partition(n):
    ss = decomp_admissibles(n)
    if ss == []: return []
    p = ss[0]
    for p1 in ss:
        if len(p1) < len(p): p = p1
    return p
```

In [36]:
```
minimal_partition(88)
```

Out[36]:
```
[2, 44]
```

In [37]:
```
minimal_partition(96)
```

Out[37]:
```
[2, 48]
```

Nous appellerons dorénavant *admissible* tout entier qui est soit admissibble au sens précéddent, soit un produit de tels entiers.

In [38]:
```
def admissible(n):
    return minimal_partition(n) != []
```

Il ne reste plus que 3 multiples de 4 inférieurs ou égaux à 100 qui résistent ...

In [39]:
```
[n for n in range(1, 101) if n % 4 == 0 and not admissible(n)]
```

Out[39]:
```
[52, 92, 100]
```

Et les entiers inférieurs à 1000 ?

In [41]:
```
print([n for n in range(1, 1001) if n % 4 == 0 and not admissible(n)])
```

```
[52, 92, 100, 116, 156, 172, 184, 188, 232, 236, 244, 260, 268, 292,
324, 340, 344, 356, 372, 376, 404, 412, 428, 436, 452, 472, 476, 508
, 520, 532, 536, 580, 584, 596, 604, 612, 652, 668, 680, 688, 712, 7
16, 724, 732, 756, 764, 772, 808, 836, 852, 856, 872, 876, 892, 904,
932, 940, 944, 952, 956, 964, 980, 988, 996]
```

On voit qu'il y a encore du travail à faire. Le plus petit entier pour lequel on ne connaît pas de matrice de Hadamard est 668 ...

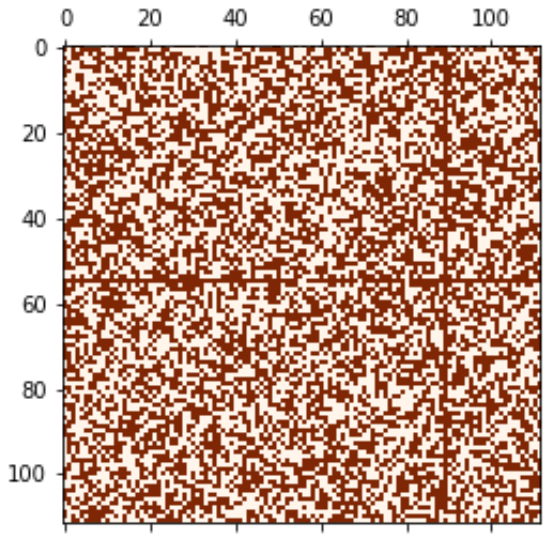
5.3 Nous y sommes ...

Voici pour terminer une fonction qui renvoie une matrice de Hadamard pour tous les entiers admissibles.

In [42]:
```
def hadamard(n):
    if n > 2 and n % 4 != 0:
        print (n, 'doit être multiple de 4')
        return False
    if admissible0(n): return hadamard0(n)
    s = minimal_partition(n)
    if s == []:
        print ('Je ne sais pas')
        return False
    else:
        A = hadamard(s[0])
        for k in s[1:]:
            A = tensor_product(A, hadamard(k))
        return A

def random_hadamard(n):
    M = hadamard(n)
    if M: return shuffle(M)
    else: return False
```

In [43]:
```
#M = hadamard(40)
M = random_hadamard(112)
if M: show_mat(M)
```



6 Références

- K.J Horadam, Hadamard Matrices and their Applications - Princeton University Press, 2007
- Shalom Eliahou, CNRS, images des mathématiques, la conjecture de Hadamard (I) (http://images.math.cnrs.fr/La-conjecture-de-Hadamard-I.html)
- Shalom Eliahou, CNRS, images des mathématiques, la conjecture de Hadamard (II) (http://images.math.cnrs.fr/La-conjecture-de-Hadamard-II.html)

In [ ]:
```
```