# Permanent

April 13, 2025

## 1 Permanent

Marc Lorenzi

10 mars 2023

[1]:
```
import matplotlib.pyplot as plt
import random
import time
import math
```

[2]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

**Notations et conventions.** $\mathbb{K}$ désigne un corps quelconque. Pour toute matrice $A \in \mathcal{M}_n(\mathbb{K})$, nous ferons varier les indices des lignes et colonnes de $A$ entre 0 et $n-1$, plutôt qu'entre 1 et $n$.

Pour tout $n \in \mathbb{N}$, nous noterons $\bar{n} = [\![0, n-1]\!]$ et $\mathfrak{S}_n = \mathfrak{S}(\bar{n})$ l'ensemble des bijections de l'ensemble $\bar{n}$ sur lui-même. Rappelons que le cardinal de $\mathfrak{S}_n$ est $n!$.

### 1.1 1. Introduction

#### 1.1.1 1.1 Mariages parfaits

$n$ cours sont à répartir dans $n$ salles de classe. Certaines contraintes (nombre d'étudiants, équipements) font qu'un cours ne peut avoir lieu que dans certaines salles. Il s'agit d'affecter une salle à chacun des cours. Quelques problèmes se posent d'emblée :

1. Étant donnée une affectation de salles, vérifier que c'est bien une solution.
2. Y a-t-il une solution ?
3. Trouver une solution, s'il y en a une.
4. Trouver toutes les solutions.
5. Combien y a-t-il de solutions ?

Les trois premiers problèmes peuvent être résolus en temps polynomial. C'est clair pour le problème numéro 1, beaucoup moins évident pour les problèmes 2 et 3. Ce qui va nous intéresser ici c'est la résolution du dernier problème : combien y a-t-il de solutions ? Nous reviendrons sur les problèmes numéros 3 et 4, de façon très naïve, tout à la fin du notebook.

Pour mathématiser le problème, donnons-nous un *graphe bipartite* $G$ possédant $2n$ sommets. Qu'est-ce qu'un graphe bipartite, me direz-vous ? Et qu'est-ce qu'un graphe ?

**Définition.** Un *graphe* (orienté) est un couple $G = (S, A)$ où $S$ est un ensemble fini, l'ensemble des *sommets* de $G$ et $A \subseteq S \times S$ est l'ensemble des *arêtes* de $G$. Si $a = (x, y) \in A$, $x$ est *l'origine* de $a$ et $y$ est la *destination* de $a$. Les sommets $x$ et $y$ sont les *extrémités$* de $a$. *Dans le cas où $x = y$, l'arête $a$ est une* boucle* sur le sommet $x$.

**Définition.** Soit $G = (S, A)$ un graphe. Le graphe $G$ est *bipartite* (non orienté) s'il existe une partition $S = X \cup Y$ de l'ensemble $S$ des sommets de $G$ telle que $A \subseteq (X \times Y) \cup (Y \times X)$ et pour tous $x, y \in S$, $(x, y) \in A \iff (y, x) \in A$.

Donnons-nous un graphe bipartite $G$. Nous supposerons de plus, dans ce qui va suivre, que $|X| = |Y|$, où la double barre désigne le cardinal de l'ensemble (le cardinal de $S$ est donc pair).

Écrivons $S = X \cup Y$ où $X = \{x_0, \dots, x_{n-1}\}$ et $Y = \{y_0, \dots, y_{n-1}\}$. Notre problème est maintenant le suivant :

Combien y a-t-il de *mariages parfaits* du graphe $G$, c'est à dire de permutations $\sigma \in \mathfrak{S}_n$ telles que pour tout $i \in \bar{n}$, il y a une arête de $G$ d'extrémités $x_i$ et $y_{\sigma(i)}$ ?

Pour illustrer un peu tout cela, voici une fonction `graphe_bipartite_alea` qui prend en paramètre un entier $n$. Elle renvoie un graphe bipartite $G$ « aléatoire » (enfin pas tout à fait, lisez le code) ayant $2n$ sommets. Le graphe $G$ est modélisé par une liste de longueur $n$. Pour $i \in \bar{n}$, $G[i]$ est une liste d'entiers entre 0 et $n - 1$. L'entier $j$ est dans $G[i]$ si et seulement si il existe une arête d'extrémités $x_i$ et $y_j$.

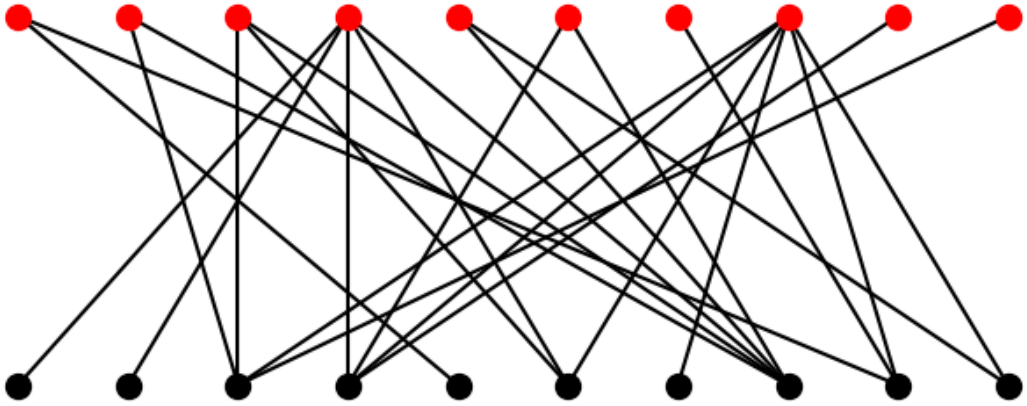
La fonction accepte un paramètre facultatif $r \in [0, 1]$. Plus $r$ est grand, plus $G$ possède d'arêtes.

[3]:
```
def graphe_bipartite_alea(n, r=0.3):
    G = n * [None]
    for i in range(n): G[i] = []
    for i in range(n):
        for j in range(n):
            p = random.uniform(0, 1)
            if p < r: G[i].append(j)
    for i in range(n):
        if G[i] == []:
            j = random.randint(0, n - 1)
            G[i].append(j)
    return G
```

Voici une fonction `plot_graph` qui affiche un graphe bipartite. En noir, l'ensemble $X$ des $x_i$, en rouge l'ensemble $Y$ des $y_j$.

[4]:
```
def plot_graph(G):
    n = len(G)
    xs = list(range(n))
    for i in range(n):
        for j in G[i]: plt.plot([i, j], [0, 1], 'k')
    plt.plot(xs, n * [1], 'or', ms=10)
    plt.plot(xs, n * [0], 'ok', ms=10)
    plt.axis('off')
```

[5]:
```
plot_graph(graphe_bipartite_alea(10, r=0.2))
```



Considérons la matrice $A^G \in \mathcal{M}_n(\{0,1\})$ définie, pour tous $i, j \in \bar{n}$, par $A^G_{ij} = 1$ s'il existe dans le graphe $G$ une arête d'extrémités $x_i$ et $y_j$ et $A^G_{ij} = 0$ sinon. Cette matrice s'appelle la *matrice de biadjacence* de $G$.

Soit $\sigma \in \mathfrak{S}_n$. La permutation $\sigma$ est un mariage parfait du graphe $G$ si et seulement si pour tout $i \in \bar{n}$, $A^G_{i\sigma(i)} = 1$, c'est à dire $A^G_{0\sigma(0)} \dots A^G_{(n-1)\sigma(n-1)} = 1$. Sinon, ce produit vaut 0. Ainsi, le nombre de mariages parfaits de $G$ est

$$\sum_{\sigma \in \mathfrak{S}_n} \prod_{0 \leq i < n} A^G_{i\sigma(i)}$$

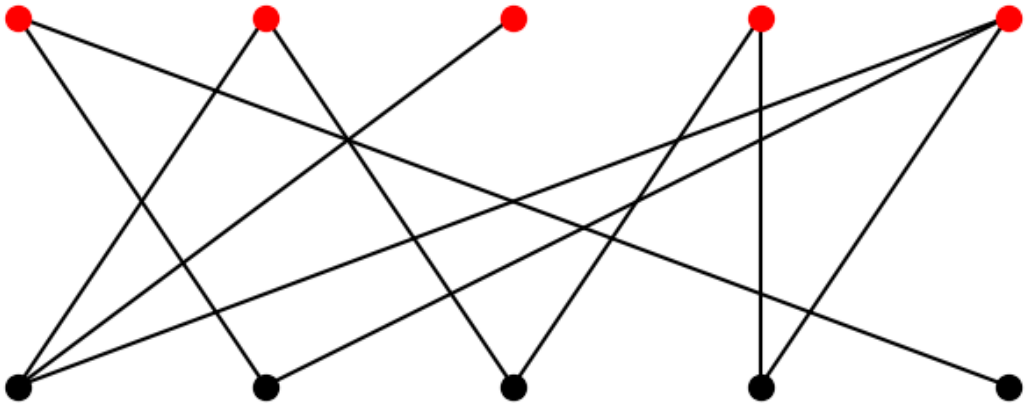
La fonction `matrice_biadjacence` prend en paramètre un graphe bipartite $G$ et renvoie la matrice $A^G$.

[6]:
```
def matrice_biadjacence(G):
    n = len(G)
    A = n * [None]
    for i in range(n):
        A[i] = n * [0]
    for i in range(n):
        for j in G[i]:
            A[i][j] = 1
    return A
```

[7]:
```
G = graphe_bipartite_alea(5)
A = matrice_biadjacence(G)
for i in range(5): print(A[i])
plot_graph(G)
```

```
[0, 1, 1, 0, 1]
[1, 0, 0, 0, 1]
```

```
[0, 1, 0, 1, 0]
[0, 0, 0, 1, 1]
[1, 0, 0, 0, 0]
```



Venons-en à ce qui nous intéresse. Nous reviendrons sur les mariages parfaits à la fin du notebook.

### 1.1.2 1.2 Notion de permanent

**Définition.** Soit $A \in \mathcal{M}_n(\mathbb{K})$. Le *permanent* de la matrice $A$ est

$$\mathrm{perm}(A) = \sum_{\sigma \in \mathfrak{S}_n} \prod_{0 \leq i < n} A_{i\sigma(i)}$$

L'expression du permanent d'une matrice ressemble beaucoup à celle de son déterminant :

$$\det(A) = \sum_{\sigma \in \mathfrak{S}_n} \varepsilon(\sigma) \prod_{0 \leq i < n} A_{i\sigma(i)}$$

où $\varepsilon(\sigma)$ est la *signature* de la permutation $\sigma$. Cependant, alors qu'il existe des algorithmes efficaces (le pivot de Gauss, par exemple) pour calculer un déterminant, on n'en connaît à ce jour aucun pour calculer un permanent. Pour dire les choses très vite (et donc faussement), on dit qu'une fonction est dans la classe $\#P$ lorsqu'elle est calculable en espace polynomial. Le calcul du permanent est un *problème $\#P$-difficile*. Si l'on connaissait un algorithme en temps polynomial pour calculer un permanent, alors toutes les fonctions de la classe $\#P$ pourraient être calculées en temps polynomial. C'est le *théorème de Valiant*. On peut montrer que cela permettrait entre autres de répondre par l'affirmative à l'un des problèmes d'informatique théorique réputés les plus inabordables à ce jour, à savoir que $P = NP$. Autant dire que nous allons faire ce que nous pouvons pour calculer des permanents mais qu'il ne faut pas s'attendre à des miracles. D'ici la fin du notebook, nous serons en mesure de calculer le permanent d'une matrice de taille 20, mais pas vraiment plus.

Nous allons dans ce qui suit décrire trois algorithmes de calcul du permanent.

- L'algorihme naïf, qui repose sur la définition du permanent.

- La formule de Laplace, qui est l'analogue pour les permanents de la formule de développement d'un déterminant par rapport à une ligne ou une colonne.
- La formule de Ryser, plus efficace que la formule de Lagrange.

La formule de Ryser est l'un des meilleurs algorithmes actuellement connus pour calculer la valeur exacte d'un permanent.

### 1.1.3 1.3 Quelques fonctions Python utiles

Passons rapidement sur quelques fonctions Python qui nous serons bien utiles.

La fonction `taille` prend en paramètre une matrice carrée, représentée en Python par une liste de listes. Elle renvoie la taille de la matrice.

[8]:
```
def taille(A): return len(A)
```

La fonction `matrice` renvoie une matrice de taille $n$ dont tous les coefficients sont égaux à $x$.

[9]:
```
def matrice(n, x):
    A = n * [None]
    for i in range(n):
        A[i] = n * [x]
    return A
```

[10]:
```
print(matrice(3, 2))
```

```
[[2, 2, 2], [2, 2, 2], [2, 2, 2]]
```

La fonction `print_mat` afiche joliment une matrice. Le paramètre optionnel $\ell$ définit la largeur d'affichage des coefficients. La valeur par défaut 2 convient si les les coefficients de $A$ sont des entiers de 1 chiffre avec un éventuel signe moins.

[11]:
```
def print_mat(A, l=2):
    n = taille(A)
    s = ''
    for i in range(n):
        s += n * ('+' + (l + 2) * '-') + '+\n'
        for j in range(n):
            s += ('| %' + str(l) + 's ') % A[i][j]
        s += '|\n'
    s += n * ('+' + (l + 2) * '-') + '+'
    print(s)
```

[12]:
```
print_mat(matrice(3, 2))
```

```
+----+----+----+
|  2 |  2 |  2 |
+----+----+----+
|  2 |  2 |  2 |
```

```
+----+----+----+
|  2 |  2 |  2 |
+----+----+----+
```

La fonction `randmat` renvoie une matrice de taille $n$ dont les coefficients sont des entiers aléatoires entre $a$ et $b$ inclus (par défaut, entre $-9$ et 9).

[13]:
```
def randmat(n, a=-9, b=9):
    A = matrice(n, 0)
    for i in range(n):
        for j in range(n):
            A[i][j] = random.randint(a, b)
    return A
```

[14]:
```
A = randmat(5)
print_mat(A)
```

```
+----+----+----+----+----+
|  6 | -1 |  9 | -2 | -5 |
+----+----+----+----+----+
|  8 | -2 | -4 | -5 |  3 |
+----+----+----+----+----+
| -4 |  0 |  8 | -1 |  5 |
+----+----+----+----+----+
| -5 |  3 |  1 | -9 | -6 |
+----+----+----+----+----+
| -7 |  7 | -9 |  2 |  7 |
+----+----+----+----+----+
```

### 1.1.4 1.4 Un calcul naïf du permanent

Commençons par écrire une fonction qui énumère toutes les permutations d'un ensemble $E$. Plus exactement, $E$ est donné par la liste de ses éléments et la fonction énumère toutes les permutations de cette liste. Le code ci-dessous est très naïf (on peut faire mieux que cela) :

- Si $E$ est vide il n'y a qu'une permutation de $E$, à savoir [].
- Sinon, soit $a$ un élément de $E$. Pour chaque permutation $\sigma$ de $E \setminus \{a\}$, on énumère les permutations obtenues en insérant $a$ dans $\sigma$.

Cette fonction est en fait ce que Python appelle un *générateur*. Elle ne crée pas la *liste* des permutations, elle les *énumère*.

[15]:
```
def permutations(E):
    if E == []: yield []
    else:
        n = len(E)
        a = E[0]
        P1 = permutations(E[1:])
        P = []
```

```
        for sigma in P1:
            for k in range(n):
                yield sigma[:k] + [a] + sigma[k:]
```

[16]:
```
for sigma in permutations([1, 2, 3]): print(sigma)
```

```
[1, 2, 3]
[2, 1, 3]
[2, 3, 1]
[1, 3, 2]
[3, 1, 2]
[3, 2, 1]
```

Le calcul naïf du permanent d'une matrice $A$ est maintenant immédiat.

[17]:
```
def perm_naif(A):
    n = taille(A)
    s = 0
    for sigma in permutations(list(range(n))):
        p = 1
        for i in range(n):
            p = p * A[i][sigma[i]]
        s = s + p
    return s
```

Le permanent d'une matrice $A$ de taille 2 est

$$\mathrm{perm}(A) = A_{00}A_{11} + A_{10}A_{01}$$

[18]:
```
A = randmat(2)
print(perm_naif(A))
print_mat(A)
```

```
-50
+----+----+
|  1 |  5 |
+----+----+
| -9 | -5 |
+----+----+
```

Le permanent d'une matrice $A$ de taille 3 est

$$\mathrm{perm}(A) = A_{00}A_{11}A_{22} + A_{01}A_{12}A_{20} + A_{02}A_{10}A_{21} + A_{00}A_{12}A_{21} + A_{01}A_{10}A_{22} + A_{02}A_{11}A_{20}$$

[19]:
```
A = randmat(3, 1, 3)
print(perm_naif(A))
print_mat(A)
```

```
47
+----+----+----+
|  3 |  1 |  1 |
+----+----+----+
|  1 |  2 |  3 |
+----+----+----+
|  2 |  3 |  1 |
+----+----+----+
```

Tentons une matrice de taille 6

[20]:
```
A = randmat(6)
print_mat(A)
print(perm_naif(A))
```

```
+----+----+----+----+----+----+
|  6 |  1 | -8 |  0 |  0 |  0 |
+----+----+----+----+----+----+
| -4 | -3 | -7 | -8 |  7 | -2 |
+----+----+----+----+----+----+
|  0 |  1 | -7 | -2 | -8 | -3 |
+----+----+----+----+----+----+
|  1 | -2 | -5 | -2 |  3 |  6 |
+----+----+----+----+----+----+
|  7 | -1 |  0 | -8 |  6 | -2 |
+----+----+----+----+----+----+
| -3 |  4 |  7 | -5 |  4 |  2 |
+----+----+----+----+----+----+
-114114
```

Évidemment, nous pouvons avoir un doute sur la correction de la réponse. N'avons-nous pas commis une erreur dans notre code ? Un lecteur motivé pourrait calculer à la main les 720 termes de la somme qui définit le permanent de $A$, chacun de ces termes étant un produit de 6 facteurs …

Lorsque nous aurons d'autres algorithmes de calcul du permanent, nous pourrons comparer les valeurs renvoyées.

### 1.1.5 1.5 La complexité de `perm_naif`

Pour tout $n \in \mathbb{N}$, appelons $C_n$ le nombre d'opérations *arithmétiques* sur des coefficients de la matrice (addition, multiplication) effectuées par `perm_naif` lors du calcul du permanent d'une matrice $n \times n$.

Pour chaque permutation $\sigma \in \mathfrak{S}_n$, la fonction effectue $n$ multiplications et 1 addition. Comme il y a $n!$ permutations, on a donc

$$C_n = (n+1) \times n! = (n+1)!$$

Notons que la valeur de $C_n$ ne prend pas en compte les opérations nécessaires pour engendrer les permutations de $\mathfrak{S}_n$.

[21]:
```
def complexite_naif(n): return math.factorial(n + 1)
```

[22]:
```
for n in range(11):
    print('%3d%10d' % (n, complexite_naif(n)))
```

```
  0         1
  1         2
  2         6
  3        24
  4       120
  5       720
  6      5040
  7     40320
  8    362880
  9   3628800
 10  39916800
```

Quel est le temps réel mis par `perm_naif` pour calculer un permanent ? La fonction `temps` prend en paramètres une fonction $f$ et un entier $n$. Elle renvoie le temps mis par le calcul de $f(A)$, où $A$ est une matrice de taille $n$.

[23]:
```
def temps(f, n):
    A = randmat(n, 0, 1)
    t1 = time.time()
    p = f(A)
    t2 = time.time()
    return t2 - t1
```

Notons $T_n$ le temps mis par `perm_naif` pour calculer le permanent d'une matrice de taille $n$. On a $T_n \simeq KC_n$ où $K$ est une constante qui dépend de la machine. Quelle est la valeur de $K$ ?

[24]:
```
t9 = temps(perm_naif, 9)
K_naif = t9 / complexite_naif(9)
print(K_naif)
```

```
2.944681668617948e-07
```

La fonction ci-dessous renvoie une estimation du temps mis par `perm_naif` pour un calcul de permanent.

[25]:
```
def temps_estime_naif(n):
    return K_naif * complexite_naif(n)
```

[26]:
```
for n in range(21):
    print('%3d%10.2es' % (n, temps_estime_naif(n)))
```

```
 0  2.94e-07s
 1  5.89e-07s
 2  1.77e-06s
 3  7.07e-06s
 4  3.53e-05s
 5  2.12e-04s
 6  1.48e-03s
 7  1.19e-02s
 8  1.07e-01s
 9  1.07e+00s
10  1.18e+01s
11  1.41e+02s
12  1.83e+03s
13  2.57e+04s
14  3.85e+05s
15  6.16e+06s
16  1.05e+08s
17  1.89e+09s
18  3.58e+10s
19  7.16e+11s
20  1.50e+13s
```

Ainsi, le temps estimé pour le calcul d'un permanent de taille 20 est $1.5 \times 10^{13}$ secondes, c'est à dire environ … 500000 ans.

## 1.2 2. Quelques propriétés du permanent

Cette section ne contient pas de code. On y montre que le permanent possède des propriétés très analogues à celles du déterminant.

### 1.2.1 2.1 Permanent et transposition

**Proposition.** Soit $A \in \mathcal{M}_n(\mathbb{K})$. On a $\mathrm{perm}(A) = \mathrm{perm}(A^T)$.

**Démonstration.** Dans les produits de la formule du permanent, posons $j = \sigma(i)$. Il vient

$$\mathrm{perm}(A) = \sum_{\sigma \in \mathfrak{S}_n} \prod_{0 \leq j < n} A_{\sigma^{-1}(j)j}$$

Posons ensuite dans la somme $\tau = \sigma^{-1}$. Lorsque $\sigma$ décrit $\mathfrak{S}_n$, $\tau$ fait de même, et donc

$$\mathrm{perm}(A) = \sum_{\tau \in \mathfrak{S}_n} \prod_{0 \leq j < n} A_{\tau(j)j} = \sum_{\tau \in \mathfrak{S}_n} \prod_{0 \leq j < n} (A^T)_{j\tau(j)} = \mathrm{perm}(A^T)$$

**Remarque.** Toutes les propriétés du permanent relatives aux lignes des matrices sont donc également vraies pour les colonnes, et vice versa.

### 1.2.2 2.2 Permanent d'une matrice triangulaire.

**Proposition.** Soit $A \in \mathcal{M}_n(\mathbb{K})$ une matrice triangulaire. On a

$$\text{perm}(A) = \prod_{i=0}^{n-1} A_{ii}$$

**Démonstration.** Supposons par exemple $A$ triangulaire supérieure, de sorte que pour tous $i, j \in \bar{n}$, $i > j \implies A_{ij} = 0$. Soit $\sigma \in \mathfrak{S}_n$. Supposons $\prod_{i=0}^{n-1} A_{i\sigma(i)} \neq 0$. On a alors pour tout $i \in \bar{n}$, $\sigma(i) \leq i$, ce qui entraîne (récurrence sur $i$) que $\sigma = id$. Ainsi, tous les termes de la somme qui définit le permanent sont nuls, sauf peut-être celui pour $\sigma = id$, qui vaut $\prod_{i=0}^{n-1} A_{ii}$.

Si $A$ est triangulaire inférieure, il suffit de considérer $A^T$ pour se ramener à ce qui précède.

### 1.2.3 2.3 Multilinéarité

Nous pouvons voir le permanent d'une matrice comme une fonction de ses $n$ lignes. On dispose donc de la fonction de $n$ variables

$$\text{perm} : \mathcal{M}_{1n}(\mathbb{K}) \times ... \times \mathcal{M}_{1n}(\mathbb{K}) \longrightarrow \mathbb{K}$$

**Proposition.** La fonction perm est $n$-linéaire symétrique.

**Démonstration.** Soient $L_0', L_0'', L_1, ..., L_{n-1} \in \mathcal{M}_{1n}(\mathbb{K})$. Notons $A'$ la matrice dont les lignes sont $L_0', L_1, ..., L_{n-1}$ et $A''$ la matrice dont les lignes sont $L_0'', L_1, ..., L_{n-1}$. On a, pour tout $\sigma \in \mathfrak{S}_n$,

$$(L_0'+L_0'')_{\sigma(0)} \prod_{1\leq i<n} (L_i)_{\sigma(i)} = (L_0')_{\sigma(0)} \prod_{1\leq i<n} (L_i)_{\sigma(i)} + (L_0'')_{\sigma(0)} \prod_{1\leq i<n} (L_i)_{\sigma(i)} = \prod_{0\leq i<n} A'_{i\sigma(i)} + \prod_{0\leq i<n} A''_{i\sigma(i)}$$

De là,

$$\begin{aligned} \text{perm}(L_0' + L_0'', L_1, ..., L_{n-1}) &= \textstyle\sum_{\sigma\in\mathfrak{S}_n} (\prod_{0\leq i<n} A'_{i\sigma(i)} + \prod_{0\leq i<n} A''_{i\sigma(i)}) \\ &= \textstyle\sum_{\sigma\in\mathfrak{S}_n} \prod_{0\leq i<n} A'_{i\sigma(i)} + \sum_{\sigma\in\mathfrak{S}_n} \prod_{0\leq i<n} A''_{i\sigma(i)} \\ &= \text{perm}(A') + \text{perm}(A'') \\ &= \text{perm}(L_0', L_1, ..., L_{n-1}) + \text{perm}(L_0'', L_1, ..., L_{n-1}) \end{aligned}$$

On laisse au lecteur le soin de vérifier que si $\lambda \in \mathbb{K}$ et $L_0, ..., L_{n-1} \in \mathcal{M}_{1n}(\mathbb{K})$, alors

$$\text{perm}(\lambda L_0, L_1, ..., L_{n-1}) = \lambda \text{ perm}(L_0, L_1, ..., L_{n-1})$$

Nous avons donc la linéarité de la fonction perm en sa première variable. Montrons la symétrie, nous aurons en prime la $n$-linéarité. Soient $L_0, ..., L_{n-1} \in \mathcal{M}_{1n}(\mathbb{K})$. Soit $A \in \mathcal{M}_n(\mathbb{K})$ la matrice dont les lignes sont $L_0, ..., L_{n-1}$. Soit $\tau \in \mathfrak{S}_n$. Soit $A'$ la matrice dont les lignes sont $L_{\tau(0)}, ..., L_{\tau(n-1)}$. Montrons que $\text{perm}(A') = \text{perm}(A)$. On a

$$\mathrm{perm}(A') = \sum_{\sigma\in\mathfrak{S}_n} \prod_{0\le i<n} A'_{i\sigma(i)} = \sum_{\sigma\in\mathfrak{S}_n} \prod_{0\le i<n} A_{\tau(i)\sigma(i)}$$

Dans chacun des produits, effectuons le changement d'indice $j = \tau(i)$. Il vient

$$\mathrm{perm}(A') = \sum_{\sigma\in\mathfrak{S}_n} \prod_{0\le i<n} A_{j(\sigma\tau^{-1})(j)}$$

Posons maintenant dans la somme $\gamma = \sigma\tau^{-1}$. Lorsque $\sigma$ décrit $\mathfrak{S}_n$, $\gamma$ fait de même, et donc

$$\mathrm{perm}(A') = \sum_{\gamma\in\mathfrak{S}_n} \prod_{0\le i<n} A_{j\gamma(j)} = \mathrm{perm}(A)$$

Retenons en particulier les deux règles suivantes :

- Échanger deux lignes ou deux colonnes d'une matrice ne change pas son permanent.
- Multiplier une ligne ou une colonne d'une matrice par $\lambda \in \mathbb{K}$ multiplie son permanent par $\lambda$.

### 1.2.4 2.4 Non-propriétés du permanent

Il existe une règle qui est valable pour les déterminants et qui est d'une utilité cruciale pour leur calcul pratique : ajouter à une ligne d'une matrice une combinaison linéaire des autres lignes ne change pas son déterminant. Cette propriété des déterminants est liée à **l'alternance** (conséquence de **l'antisymétrie**) de la fonction déterminant alors que le permanent, quant à lui, est une fonction **symétrique**.

Cette opération n'est donc pas possible pour les permanents. On ne peut pas effectuer d'opérations sur les lignes et les colonnes d'un permanent pour se ramener au permanent d'une matrice triangulaire qui, lui, est facile à calculer. Par exemple, soit $A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$. Ajoutons la deuxième ligne de $A$ à la première pour obtenir $B = \begin{pmatrix} 3 & 7 \\ 2 & 4 \end{pmatrix}$. On a $\mathrm{perm}(A) = 10$ alors que $\mathrm{perm}(B) = 26$.

Notons également que le déterminant est un morphisme pour la multiplication matricielle, mais que le permanent ne l'est pas. Prenons par exemple $A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$ et $B = \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix}$. On a $AB = \begin{pmatrix} 23 & 31 \\ 34 & 46 \end{pmatrix}$. De là,

$$\mathrm{perm}(AB) = 2112$$

alors que

$$\mathrm{perm}(A) \times \mathrm{perm}(B) = 10 \times 82 = 820$$

Autre propriété qui n'est pas partagée par les permanents et les déterminants, deux matrices semblables n'ont pas nécessairement le même permanent. Le permanent n'est donc pas invariant

par changement de base, et il n'est donc pas possible de définir le permanent d'un endomorphisme d'un espace vectoriel, comme on le fait pour le déterminant.

#### 1.2.5 2.5 La formule de Laplace

Nous allons voir que la formule de Laplace, qui permet de *développer* un déterminant par rapport à une ligne ou une colonne, est également vraie pour les permanents. Ceci nous donnera un algorithme de calcul du permanent différent de l'algorithme naïf.

Soit $A \in \mathcal{M}_n(\mathbb{K})$. Pour tous $i, j \in [\![0, n-1]\!]$, notons $A[i,j]$ la matrice obtenue en supprimant la ligne $i$ et la colonne $j$ de $A$. On a donc $A[i,j] \in \mathcal{M}_{n-1}(\mathbb{K})$.

**Proposition.** Soit $A \in \mathcal{M}_n(\mathbb{K})$.

- Développement par rapport à la ligne $i$ : pour tout $i \in [\![0, n-1]\!]$,

$$\mathrm{perm}(A) = \sum_{0 \leq j < n} A_{ij} \mathrm{perm}(A[i,j])$$

- Développement par rapport à la colonne $j$ : pour tout $j \in [\![0, n-1]\!]$,

$$\mathrm{perm}(A) = \sum_{0 \leq i < n} A_{ij} \mathrm{perm}(A[i,j])$$

Admettons provisoirement ce résultat et attaquons nous à l'écriture du code Python.

### 1.3 3. Implémentation de la formule de Laplace

#### 1.3.1 3.1 La fonction Python

Voici la fonction calculant le permanent d'une matrice à l'aide de la formule de Laplace. Elle prend en paramètres une matrice $A$, un ensemble `lignes` de lignes et un ensemble `cols` de colonnes (c'est à dire deux objets Python de type `set`). La fonction renvoie le permanent de la matrice extraite de $A$ en prenant les lignes de $A$ de l'ensemble `lignes` et les colonnes de $A$ de l'ensemble `cols`. Elle utilise un développement suivant la ligne numéro $i$, où $i$ est la valeur renvoyée par `lignes.pop()`, c'est à dire un numéro de ligne « au hasard ».

[27]:
```
def perm_laplace0(A, lignes, cols):
    if not lignes: return 1
    else:
        i = lignes.pop()
        cols1 = cols.copy()
        p = 0
        for j in cols1:
            cols.remove(j)
            pij = perm_laplace0(A, lignes, cols)
            cols.add(j)
            p = p + A[i][j] * pij
```

```
        lignes.add(i)
        return p
```

[28]:
```
def perm_laplace(A):
    n = taille(A)
    return perm_laplace0(A, set(range(n)), set(range(n)))
```

Comparons les résultats renvoyés par `perm_naif` et `perm_laplace`.

[29]:
```
A = randmat(7)
print(perm_naif(A))
print(perm_laplace(A))
```

```
1864402
1864402
```

Tout a l'air d'aller. Remarquons que notre fonction `perm_laplace` permet de calculer presque instantanément des permanents de matrices de tailles inférieures à 8 ou 9. Après, les choses se gâtent.

[30]:
```
for n in range(10):
    print('%3d%10.2es' % (n, temps(perm_laplace, n)))
```

```
  0  3.81e-06s
  1  6.20e-06s
  2  8.11e-06s
  3  1.79e-05s
  4  6.41e-05s
  5  3.29e-04s
  6  2.26e-03s
  7  1.64e-02s
  8  1.04e-01s
  9  9.36e-01s
```

Pour une matrice de taille 9, il faut attendre environ 0.9 seconde (sur ma machine) pour avoir le résultat. Comme nous allons le voir dans l'étude théorique de complexité qui suit, on peut parier sur 9 secondes pour une matrice de taille 10.

[31]:
```
t10 = temps(perm_laplace, 10)
print(t10)
```

```
9.067628622055054
```

Le calcul d'un permanent de taille 20 n'est toujours pas à notre portée, loin s'en faut.

### 1.3.2 3.2 La complexité de la fonction `perm_laplace`

Notons $C'_n$ le nombre d'opérations arithmétiques effectuées par la fonction `perm_laplace` lors du calcul du permanent d'une matrice de taille $n$.

Tout d'abord, $C'_0 = 0$. Puis, pour tout $n \geq 1$, le calcul du permanent d'une matrice de taille $n$ nécessite :

- Le calcul de $n$ permanents de matrices de taille $n-1$.
- $n$ multiplications
- $n$ additions.

Ainsi, $C'_n = nC'_{n-1} + 2n$.

**Proposition.** Pour tout $n \in \mathbb{N}$,

$$C'_n = 2n! \sum_{k=0}^{n-1} \frac{1}{k!}$$

**Démonstration.** Faisons une récurrence sur $n$. Pour $n = 0$, c'est clair puisque $C'_0 = 0$ et que la somme est nulle. Soit $n \in \mathbb{N}^*$. Supposons la propriété vérifiée pour $n-1$. On a alors

$$C'_n = nC'_{n-1} + 2n = n\left(2(n-1)! \sum_{k=0}^{n-2} \frac{1}{k!}\right) + 2n = 2n! \sum_{k=0}^{n-2} \frac{1}{k!} + 2n$$

Il reste à remarquer que

$$2n = 2n! \frac{1}{(n-1)!}$$

Lorsque $n$ tend vers l'infini, la somme tend vers $e$. Ainsi, $C'_n \sim 2en!$.

[32]:
```
def complexite_laplace(n):
    if n == 0: return 0
    else:
        return n * complexite_laplace(n - 1) + 2 * n
```

[33]:
```
for n in range(11):
    print('%3d%10d%10d' % (n, complexite_naif(n), complexite_laplace(n)))
```

```
  0         1         0
  1         2         2
  2         6         8
  3        24        30
  4       120       128
  5       720       650
  6      5040      3912
  7     40320     27398
  8    362880    219200
  9   3628800   1972818
 10  39916800  19728200
```

On peut ainsi espérer que `perm_laplace` donnera de meilleurs résultats que `perm_naif`.

Remarquons toutefois que nous n'avons compté dans le calcul de $C'_n$ que les opérations *arithmétiques* effectuées par la fonction `perm_laplace`. En particulier, la ligne `cols1 = cols.copy()` a un coût en $O(n)$.

### 1.3.3 3.3 Temps estimés

Notons $T'_n$ le temps mis par `perm_lagrange` pour calculer le permanent d'une matrice de taille $n$. On a $T'_n \simeq KC'_n$, où $K$ est une constante qui dépend de la machine utilisée. Quelle est la valeur de $K$ ?

[34]:
```
t10 = temps(perm_laplace, 10)
K_laplace = t10 / complexite_laplace(10)
print(K_laplace)
```

```
4.6059712372567136e-07
```

[35]:
```
def temps_estime_laplace(n):
    return K_laplace * complexite_laplace(n)
```

[36]:
```
for n in range(21):
    print('%3d%10.2es%10.2es' % (n, temps_estime_naif(n),␣
 ↪temps_estime_laplace(n)))
```

```
  0  2.94e-07s  0.00e+00s
  1  5.89e-07s  9.21e-07s
  2  1.77e-06s  3.68e-06s
  3  7.07e-06s  1.38e-05s
  4  3.53e-05s  5.90e-05s
  5  2.12e-04s  2.99e-04s
  6  1.48e-03s  1.80e-03s
  7  1.19e-02s  1.26e-02s
  8  1.07e-01s  1.01e-01s
  9  1.07e+00s  9.09e-01s
 10  1.18e+01s  9.09e+00s
 11  1.41e+02s  1.00e+02s
 12  1.83e+03s  1.20e+03s
 13  2.57e+04s  1.56e+04s
 14  3.85e+05s  2.18e+05s
 15  6.16e+06s  3.27e+06s
 16  1.05e+08s  5.24e+07s
 17  1.89e+09s  8.91e+08s
 18  3.58e+10s  1.60e+10s
 19  7.16e+11s  3.05e+11s
 20  1.50e+13s  6.09e+12s
```

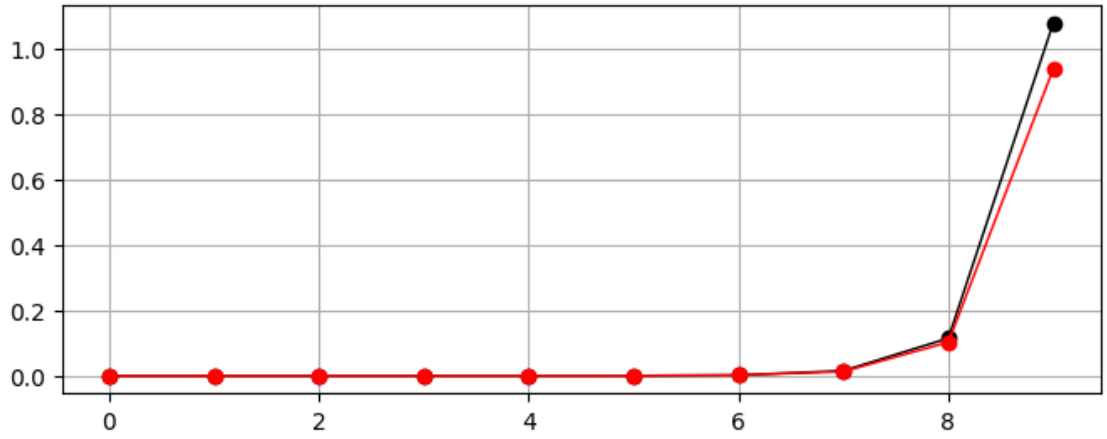
Le calcul du permanent d'une matrice de taille 20 n'est toujours pas possible.

Comparons graphiquement les temps de calcul des fonctions `permanent_naif` et `permanent_laplace`.

[37]:
```
def plot_temps(nmax, liste_fonctions_couleurs, log=True):
    ns = range(nmax + 1)
    funs = []
    for f, c in liste_fonctions_couleurs:
        ts = []
        for n in ns:
            A = randmat(n)
            t1 = time.time()
            p = f(A)
            t2 = time.time()
            ts.append(t2 - t1)
        if log:
            plt.semilogy(ns, ts, c, lw=1)
            plt.semilogy(ns, ts, 'o'+ c, lw=1)
        else:
            plt.plot(ns, ts, c, lw=1)
            plt.plot(ns, ts, 'o'+ c, lw=1)
    plt.grid()
```

En noir, le temps mis par `permanent_naif` pour calculer le permanent de matrices aléatoires de tailles entre 0 et 9. En rouge, le temps mis par `permanent_laplace` pour ces mêmes matrices.

[38]:
```
plot_temps(9, [(perm_naif, 'k'), (perm_laplace, 'r')], log=False)
```




**Moralité** : Sans conviction, petit gain de temps pour la fonction `perm_laplace` ...

### 1.3.4 3.4 La preuve de la formule de Laplace

Montrons la formule de développement par rapport à une ligne. Par transposition, la formule de développement par rapport à une colonne s'en déduit. Soit $A \in \mathcal{M}_n(\mathbb{K})$. Notons $L_0, \dots, L_{n-1}$ les lignes de $A$. Soit $i \in [\![0, n-1]\!]$. Écrivons

$$L_i = \sum_{j=0}^{n-1} A_{ij} L'_j$$

où $L'_j = (0 \dots 0\ 1\ 0 \dots 0)$ a un 1 en position $j$ et des zéros ailleurs. Par la $n$-linéarité du permanent,

$$\mathrm{perm}(A) = \sum_{j=0}^{n-1} A_{ij} \mathrm{perm}(L_0, \dots, L'_j, \dots, L_{n-1})$$

où $L'_j$ se trouve en position $i$ ($L_i$ ne fait donc pas partie de l'énumération). Par la symétrie de la fonction permanent,

$$\mathrm{perm}(L_0, \dots, L'_j, \dots, L_{n-1}) = \mathrm{perm}(L_0, \dots, \dots, L_{n-1}, L'_j)$$

où $L_i$ n'apparaît pas dans l'énumération. Enfin, par des échanges de colonnes (qui conservent le permanent), on obtient

$$\mathrm{perm}(L_0, \dots, \dots, L_{n-1}, L'_j) = \mathrm{perm}(A')$$

où $A'$ est la matrice définie par blocs par

$$A' = \begin{pmatrix} A[i,j] & 0 \\ 0 & 1 \end{pmatrix}$$

Il reste à montrer que $\mathrm{perm}(A') = \mathrm{perm}(A[i,j])$. Pour cela, revenons à la définition du permanent. On a

$$\mathrm{perm}(A') = \sum_{\sigma \in \mathfrak{S}_n} \prod_{0 \le k < n} A'_{k\sigma(k)}$$

Remarquons que si $\sigma(n-1) \neq n-1$, alors $A'_{(n-1)\sigma(n-1)} = 0$. En notant

$$\mathfrak{S}'_n = \{\sigma \in \mathfrak{S}_n : \sigma(n-1) = n-1\}$$

on a donc

$$\mathrm{perm}(A') = \sum_{\sigma \in \mathfrak{S}'_n} \prod_{0 \le k < n} A'_{k\sigma(k)} = \sum_{\sigma \in \mathfrak{S}'_n} \prod_{0 \le k < n-1} A'_{k\sigma(k)} = \sum_{\sigma \in \mathfrak{S}'_n} \prod_{0 \le k < n-1} A[i,j]_{k\sigma(k)}$$

L'ensemble $\mathfrak{S}'_n$ est clairement en bijection avec $\mathfrak{S}_{n-1}$ par la fonction qui à $\sigma$ associe la restriction de $\sigma$ à $[\![0, n-2]\!]$. La somme ci-dessus est donc égale à $\mathrm{perm}(A[i,j])$.

## 1.4 4. La formule de Ryser (I)

### 1.4.1 4.1 La formule

La formule de Ryser permet de calculer le permanent d'une matrice plus efficacement qu'avec la formule de Laplace. Ne soyons pas trop exigeants, la complexité du calcul sera toujours exponentielle. Une implémentation très travaillée de cette formule, écrite dans un langage compilé, permet d'atteindre en une heure le calcul de permanents de matrices de taille environ 35. Voir par exemple le lien ci-dessous. Vous y apprendrez aussi que le calcul efficace de permanents est crucial pour certaines applications militaires.

https://apps.dtic.mil/sti/citations/AD1004183

**Proposition.** Soit $A \in \mathcal{M}_n(\mathbb{K})$. On a

$$\mathrm{perm}(A) = \sum_{S \subseteq \bar{n}} (-1)^{n-|S|} \prod_{i=0}^{n-1} \sum_{j \in S} A_{ij}$$

Nous prouverons cette formule un peu plus loin. Pour prendre un exemple, regardons $n = 2$. L'ensemble $\bar{2}$ contient 4 parties qui sont $\emptyset$, $\{0\}$, $\{1\}$ et $\{0,1\}$. La somme de la formule de Ryser est la somme de 4 termes ci-dessous :

$$0 - A_{00}A_{10} - A_{01}A_{11} + (A_{00} + A_{01})(A_{10} + A_{11})$$

En développant, on obtient $A_{00}A_{11} + A_{01}A_{10}$ qui est bien le permanent de $A$.

**Exercice.** Que donne la formule de Ryser pour une matrice de taille 1 ? De taille 0 ?

### 1.4.2 4.2 Énumérer les parties de $\bar{n}$

Une partie $S$ de $\bar{n}$ peut être caractérisée par un $n$-uplet $(b_0, \ldots, b_{n-1}) \in \{0,1\}^n$. Pour tout $i \in \bar{n}$, $i \in S$ si et seulement si $b_i = 1$. Énumérer les parties de $\bar{n}$ revient donc à énumérer les listes de $n$ bits. Il existe pour cela une façon efficace de procéder : la technique des *codes de Gray*. Un code de Gray est une énumération des éléments de $\{0,1\}^n$ telle que le passage de chaque élément au suivant ne modifie qu'un seul bit. Nous ne montrerons pas le résultat suivant :

**Pour passer du $i$ème code au $(i+1)$ème, on modifie le bit numéro $\nu_2(i+1)$, où $\nu_2$ est la** *valuation dyadique*.

La fonction `nu2` renvoie la valuation dyadique de l'entier $n$, c'est à dire le plus grand entier naturel $k$ tel que $2^k$ divise $n$.

[39]:
```
def nu2(n):
    k = 0
    while n % 2 == 0:
```

```
        k += 1
        n = n // 2
    return k
```

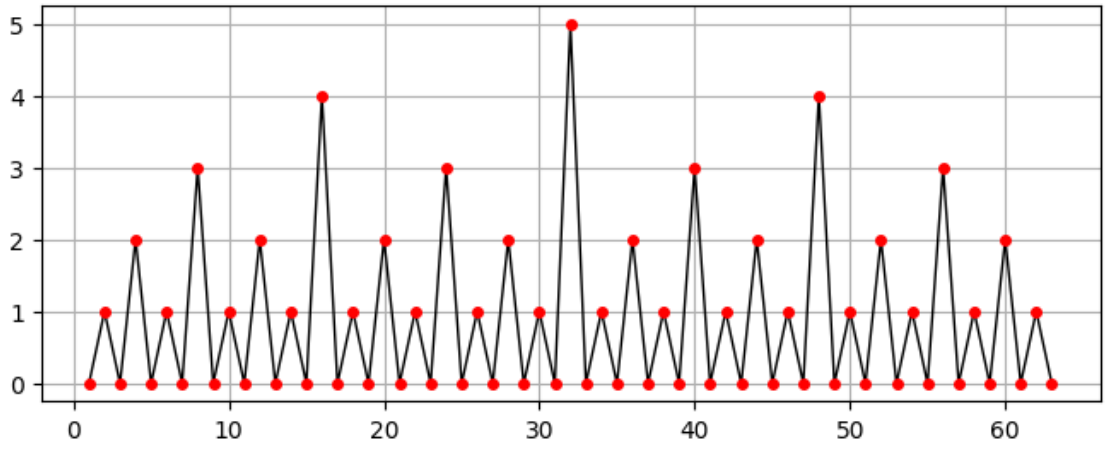
Voici les valeurs de $\nu_2(n)$ pour $n \leq 20$.

[40]:
```
for n in range(1, 21): print('%3d' % n, end='')
print()
for n in range(1, 21): print('%3d' % nu2(n), end='')
```

```
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
  0  1  0  2  0  1  0  3  0  1  0  2  0  1  0  4  0  1  0  2
```

Et voici le graphe de $\nu_2$ sur l'intervalle $[\![1, 63]\!]$.

[41]:
```
xs = range(1, 64)
ys = [nu2(x) for x in xs]
plt.plot(xs, ys, 'k', lw=1)
plt.plot(xs, ys, 'or', ms=4)
plt.grid()
```




La fonction `parties` est un générateur. Elle énumère les éléments de $\{0, 1\}^n$.

[42]:
```
def parties(n):
    code = n * [0]
    maxi = 2 ** n
    c = 0
    for k in range(maxi):
        yield code
        if k != maxi - 1:
            c = c + 1
            j = nu2(c)
```

```
            code[j] = 1 - code[j]
```

[43]:
```
for S in parties(3): print(S)
```

```
[0, 0, 0]
[1, 0, 0]
[1, 1, 0]
[0, 1, 0]
[0, 1, 1]
[1, 1, 1]
[1, 0, 1]
[0, 0, 1]
```

### 1.4.3 4.3 Une implémentation de la formule de Ryser

La fonction `somme_ligne` prend en paramètres

- Une matrice $A$ de taille $n$.
- Un entier $0 \leq i < n$.
- Une partie $S$ de $\bar{n}$ représentée par une liste de $n$ bits.

Elle renvoie $\sum_{j \in S} A_{ij}$.

[44]:
```
def somme_ligne(A, i, S):
    n = taille(A)
    s = 0
    for j in range(n):
        if S[j] == 1: s += A[i][j]
    return s
```

Enfin, la fonction `perm_ryser` renvoie le permanent de la matrice $A$, calculé par la formule de Ryser. Ce code pourrait être amélioré : comme nous utilisons un code de Gray pour énumérer les parties de $\bar{n}$, les sommes portant sur les lignes de $A$ ne sont modifiée que sur 1 terme lorsqu'on passe d'une partie de $\bar{n}$ à la suivante. Avec un peu de soin, nous pourrions calculer la valeur des sommes à partir des sommes précédentes en temps $O(1)$. Ceci diviserait la complexité par un facteur $n$. Nous le ferons à la section suivante.

Remarquons que comme nous énumérons les parties de $\bar{n}$ par un code de Gray, si $S$ et $S'$ sont deux parties successives de l'énumération, on a $|S'| = |S| \pm 1$. Ainsi, $\varepsilon = (-1)^{|S|}$ est changé en $-\varepsilon$.

[45]:
```
def perm_ryser(A):
    n = taille(A)
    perm = 0
    eps = (-1) ** n
    for S in parties(n):
        p = 1
        for i in range(n):
            p = p * somme_ligne(A, i, S)
```

```
        perm = perm + eps * p
        eps = -eps
    return perm
```

Histoire de vérifier, comparons les nombres renvoyés par `prem_lagrange` et `perm_ryser`.

[46]:
```
A = randmat(5)
print(perm_ryser(A), perm_laplace(A))
```

```
18376 18376
```

Tout va bien, deux méthodes totalement différentes donnent le même résultat.

Avec `perm_lagplace`, il ne nous était pas possible de caluler le permanent d'une matrice de taille 10 en moins de 30 secondes. Essayons avec `perm_ryser`.

[47]:
```
A = randmat(10)
print(perm_ryser(A))
```

```
8703832254
```

Cette fois-ci, la réponse est immédiate. Voici les tailles de matrices pour lesquelles le temps de calcul est inférieur à 3 secondes.

[48]:
```
t, n = 0, 0
while t < 3:
    print('%4d %10.2es' % (n, t))
    n = n + 1
    t = temps(perm_ryser, n)
```

```
   0   0.00e+00s
   1   1.31e-05s
   2   1.72e-05s
   3   4.32e-05s
   4   1.15e-04s
   5   3.05e-04s
   6   7.35e-04s
   7   1.85e-03s
   8   4.52e-03s
   9   1.11e-02s
  10   2.87e-02s
  11   5.63e-02s
  12   1.36e-01s
  13   3.11e-01s
  14   6.74e-01s
  15   1.51e+00s
```

### 1.4.4 4.4 Complexité

Notons $C''_n$ le nombre d'opérations arithmétiques effectuées par la fonction `perm_ryser` lors du calcul du permanent d'une matrice de taille $n$ (sans compter les opérations effectuées pour énumérer les parties de $\bar{n}$).

Pour chaque partie $S$ de $\bar{n}$, la fonction `perm_ryser` effectue

- $n+1$ multiplications
- $n|S|+1$ additions.

Ainsi,

$$C''_n = \sum_{S\in\mathcal{P}(\bar{n})} (n|S|+n+2) = (n+2)2^n + n\sum_{S\in\mathcal{P}(\bar{n})} |S|$$

Calculons cette dernière somme. Pour tout $k \leq n$, notons

$$\mathcal{P}_k(\bar{n}) = \{S \in \mathcal{P}(\bar{n}) : |S| = k\}$$

On a

$$\sum_{S\in\mathcal{P}(\bar{n})} |S| = \sum_{k=0}^{n} \sum_{S\in\mathcal{P}_k(\bar{n})} |S| = \sum_{k=0}^{n} k|\mathcal{P}_k(\bar{n})| = \sum_{k=0}^{n} k\binom{n}{k}$$

Remarquons que si $k \geq 1$, alors

$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

On a donc

$$\sum_{k=0}^{n} k\binom{n}{k} = n\sum_{k=1}^{n} \binom{n-1}{k-1} = n2^{n-1}$$

Ainsi,

$$C''_n = (n+2)2^n + n^2 2^{n-1} = (n^2+2n+4)2^{n-1} \sim n^2 2^{n-1}$$

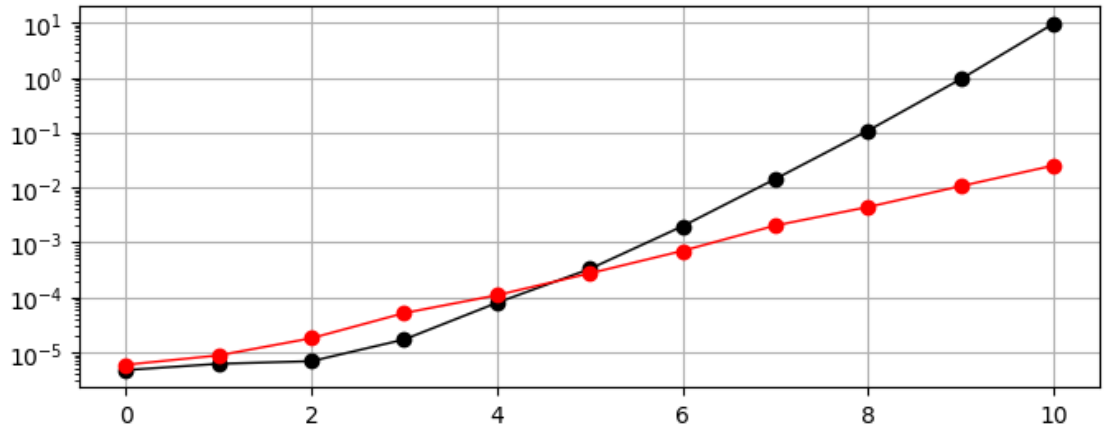
[49]:
```
def complexite_ryser(n):
    return (n ** 2 + 2 * n + 4) * 2 ** (n - 1)
```

[50]:
```
for n in range(20):
    print('%3d%10d' % (n, complexite_ryser(n)))
```

```
 0         2
 1         7
 2        24
 3        76
 4       224
 5       624
 6      1664
 7      4288
 8     10752
 9     26368
10     63488
11    150528
12    352256
13    815104
14   1867776
15   4243456
16   9568256
17  21430272
18  47710208
19 105644032
```

Terminons par une comparaison graphique des temps de calculs des fonctions `perm_laplace` et `perm_ryser`. En abscisse, la taille des matrices. En ordonnées, les temps de calcul en coordonnées logarithmiques. En noir, la formule de Laplace, en rouge la formule de Ryser.

[51]:
```
plot_temps(10, [(perm_laplace, 'k'), (perm_ryser, 'r')], log=True)
```




**Moralité** : Cette fois-ci, le gain de temps est réel. Pour une matrice de taille 10 la fonction `perm_ryser` est environ 1000 fois plus rapide que la fonction `perm_laplace`.

### 1.4.5 4.5 Temps estimés

Notons $T''_n$ le temps mis par `perm_ryser` pour calculer le permanent d'une matrice de taille $n$. On a $T''_n \simeq KC''_n$ où $K$ est une constante. Quelle est la valeur de $K$ ?

[52]:
```
t15 = temps(perm_ryser, 15)
K_ryser = t15 / complexite_ryser(15)
print(K_ryser)
```

```
3.6487073330100784e-07
```

[53]:
```
def temps_estime_ryser(n):
    return K_ryser * complexite_ryser(n)
```

[54]:
```
for n in range(21):
    print('%3d%10.2es%10.2es' % (n, temps_estime_laplace(n),␣
 ↪temps_estime_ryser(n)))
```

```
  0  0.00e+00s  7.30e-07s
  1  9.21e-07s  2.55e-06s
  2  3.68e-06s  8.76e-06s
  3  1.38e-05s  2.77e-05s
  4  5.90e-05s  8.17e-05s
  5  2.99e-04s  2.28e-04s
  6  1.80e-03s  6.07e-04s
  7  1.26e-02s  1.56e-03s
  8  1.01e-01s  3.92e-03s
  9  9.09e-01s  9.62e-03s
 10  9.09e+00s  2.32e-02s
 11  1.00e+02s  5.49e-02s
 12  1.20e+03s  1.29e-01s
 13  1.56e+04s  2.97e-01s
 14  2.18e+05s  6.81e-01s
 15  3.27e+06s  1.55e+00s
 16  5.24e+07s  3.49e+00s
 17  8.91e+08s  7.82e+00s
 18  1.60e+10s  1.74e+01s
 19  3.05e+11s  3.85e+01s
 20  6.09e+12s  8.49e+01s
```

Le calcul d'un permanent de taille 20 par notre implémentation de la formule de Ryser demandera donc environ 1min 20s. C'est tout à fait faisable ! Le lecteur est invité à essayer, en décommentant la cellule ci-dessous. Nous ferons mieux bientôt …

[55]:
```
#A = randmat(20)
#print(perm_ryser(A))
```

### 1.4.6 4.6 La preuve

Il nous reste à prouver la formule de Ryser. Soit $A \in \mathcal{M}_n(\mathbb{K})$. Notons

$$\pi(A) = \sum_S (-1)^{n-|S|} \prod_{i=0}^{n-1} \sum_{j \in S} A_{ij}$$

Où $S$ désigne une partie de $\bar{n}$. On a, en développant le produit,

$$\pi(A) = \sum_S \sum_{(j_0,\dots,j_{n-1}) \in S^n} (-1)^{n-|S|} A_{0j_0} \dots A_{(n-1)j_{n-1}}$$

Échangeons maintenant les deux sommes pour obtenir

$$\pi(A) = \sum_{(j_0,\dots,j_{n-1}) \in \bar{n}^n} \alpha_{j_0 \dots j_{n-1}} A_{0j_0} \dots A_{(n-1)j_{n-1}}$$

où

$$\alpha_{j_0 \dots j_{n-1}} = \sum_{\{j_0,\dots,j_{n-1}\} \subseteq S} (-1)^{n-|S|}$$

Soient $j_0, \dots, j_{n-1} \in \bar{n}$. Soit $k$ le cardinal de l'ensemble $\{j_0, \dots, j_{n-1}\}$. Les ensembles $S$ contenant $j_0, \dots j_{n-1}$ sont de cardinaux $k$, $k+1$, ..., $n$. Pour tout $r \in [|k, n-1|]$, il existe $\binom{n-k}{r-k}$ tels ensembles. Ainsi,

$$\alpha_{j_0 \dots j_{n-1}} = \sum_{r=k}^{n} (-1)^{n-r} \binom{n-k}{r-k}$$

Posons $r' = r - k$ pour obtenir

$$\alpha_{j_0 \dots j_{n-1}} = \sum_{r=0}^{n-k} (-1)^{n-r-k} \binom{n-k}{r} = (-1)^{n-k} \sum_{r=0}^{n-k} (-1)^r \binom{n-k}{r} = (-1)^{n-k}(1-1)^{n-k} = (-1)^{n-k} 0^{n-k}$$

- Si $k = n$, alors $\alpha_{j_0 \dots j_{n-1}} = 1$.
- Si $k < n$, alors $\alpha_{j_0 \dots j_{n-1}} = 0$.

Ainsi,

$$\pi(A) = \sum_{(j_0,\dots,j_{n-1}) \in E_n} A_{0j_0} \dots A_{(n-1)j_{n-1}}$$

où $E_n$ désigne l'ensemble des $n$-uplets $(j_0, \dots, j_{n-1})$ d'entiers distincts entre 0 et $n-1$. Il reste à remarquer que $\mathfrak{S}_n$ est en bijection avec $E_n$ via l'application $\phi : \sigma \longmapsto (\sigma(0), \dots, \sigma(n-1))$. En posant dans la somme ci-dessus $(j_0, \dots, j_{n-1}) = \phi(\sigma)$, on obtient

$$\pi(A) = \sum_{\sigma \in \mathfrak{S}_n} A_{0\sigma(0)} \cdots A_{(n-1)\sigma(n-1)} = \mathrm{perm}(A)$$

## 1.5 5 Une meilleure implémentation

### 1.5.1 5.1 Ryser, le retour

Comme nous l'avons déjà dit, il est possible d'améliorer la complexité temporelle de la fonction `perm_ryser`. L'idée est de garder en mémoire les valeurs des sommes qui interviennent dans la formule de Ryser. En énumérant les parties de $\bar{n}$ par un code de Gray, on ne change qu'un terme de chacune des sommes. Il est donc possible de recalculer *toutes* les sommes en temps $O(n)$, et non plus *chaque* somme en temps $O(n)$. Voici le code, il est un peu plus compliqué que le code de `perm_ryser`, mais nous allons voir que l'on gagne en efficacité.

[56]:
```
def perm_ryser2(A):
    n = taille(A)
    perm = 0
    eps = (-1) ** n
    S = n * [0]
    sommes = n * [0]
    p2n = 2 ** n
    for k in range(p2n):
        p = 1
        for i in range(n):
            p = p * sommes[i]
        perm = perm + eps * p
        eps = -eps
        if k < p2n - 1:
            j = nu2(k + 1)
            for i in range(n):
                if S[j] == 1: sommes[i] = sommes[i] - A[i][j]
                else: sommes[i] = sommes[i] + A[i][j]
            S[j] = 1 - S[j]
    return perm
```

Tout d'abord, constatons que les fonctions `perm_ryser` et `perm_ryser2` renvoient les mêmes résultats.

[57]:
```
A = randmat(10)
print(perm_ryser(A), perm_ryser2(A))
```

```
31146380854 31146380854
```

Comparons les temps mis par nos deux fonctions pour calculer le permanent d'une matrice de taille 17 :

[58]:
```
t1 = temps(perm_ryser, 17)
print('Temps ryser  : %10.2es' % t1)
t2 = temps(perm_ryser2, 17)
print('Temps ryser2 : %10.2es' % t2)
```

```
Temps ryser  :   7.65e+00s
Temps ryser2 :   1.16e+00s
```

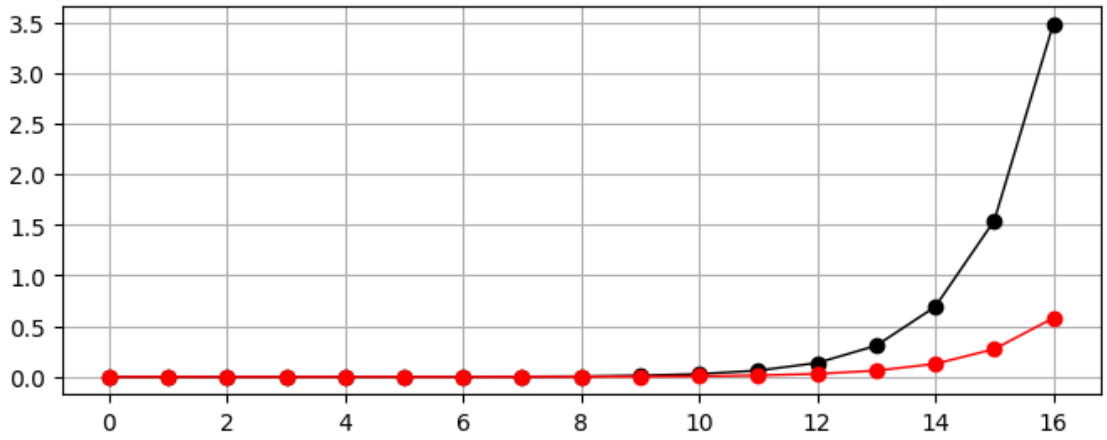
La fonction `perm_ryser2` nous permet de calculer sans trop de difficulté (une dizaine de secondes sur ma machine) le permanent d'une matrice de taille 20. Rappelons-nous que la fonction `perm_ryser` mettait 1min20s pour effectuer ce même calcul.

[59]:
```
print('%10.2es' % temps(perm_ryser2, 20))
```

```
  1.05e+01s
```

Et voici pour terminer le graphique des temps comparés entre les fonctions `perm_ryser` et `perm_ryser2`. Le gain est nettement appréciable.

[60]:
```
plot_temps(16, [(perm_ryser, 'k'), (perm_ryser2, 'r')], log=False)
```




### 1.5.2 5.2 La complexité de `perm_ryser2`

La fonction `perm_ryser2` effectue une boucle de $2^n$ itérations sur la variable $k$. Pour chaque itération, on effectue

- $n$ multiplications dans la boucle sur $i$.
- 1 addition et 1 multiplication juste après.
- Si $k \neq 2^n - 1$, $|S| + 1$ additions.

Pour ne pas trop alourdir notre analyse, supposons que les $|S| + 1$ additions sont aussi effectuées lorsque $k = 2^n - 1$. Notons $C'''_n$ le nombre total d'opérations arithmétiques. On a

$$C'''_n = 2^n(n+2) + \sum_{S\subseteq\bar{n}} (|S|+1) = 2^n(n+3) + \sum_{S\subseteq\bar{n}} |S|$$

Que vaut la dernière somme ? Réordonnons suivant le cardinal de $S$. On a

$$\sum_{S\subseteq\bar{n}} |S| = \sum_{k=0}^{n} \sum_{|S|=k} |S| = \sum_{k=0}^{n} k\binom{n}{k}$$

Cette somme a déjà été calculée lors de l'étude de la complexité de `perm_ryser`. Elle vaut $n2^{n-1}$. Ainsi,

$$C'''_n = (n+3)2^n + n2^{n-1} = 3(n+2)2^{n-1}$$

[61]:
```
def complexite_ryser2(n):
    return 3 * (n + 2) * 2 ** (n - 1)
```

[62]:
```
for n in range(20):
    print('%10d%10d' % (complexite_ryser(n), complexite_ryser2(n)))
```

```
         2         3
         7         9
        24        24
        76        60
       224       144
       624       336
      1664       768
      4288      1728
     10752      3840
     26368      8448
     63488     18432
    150528     39936
    352256     86016
    815104    184320
   1867776    393216
   4243456    835584
   9568256   1769472
  21430272   3735552
  47710208   7864320
 105644032  16515072
```

Notons $T'''_n$ le temps mis par `perm_ryser2` pour calculer le permanent d'une matrice de taille $n$. On a $T'''_n \simeq KC'''_n$, où $K$ est une constante. Quelle est la valeur de $K$ ?

[63]:
```
t17 = temps(perm_ryser2, 17)
K_ryser2 = t17 / complexite_ryser2(17)
print(K_ryser2)
```

```
3.111057786123925e-07
```

[64]:
```
def temps_estime_ryser2(n):
    return K_ryser2 * complexite_ryser2(n)
```

[65]:
```
for n in range(21):
    print('%3d%10.2es%10.2es' % (n, temps_estime_ryser(n),␣
 ↪temps_estime_ryser2(n)))
```

```
  0  7.30e-07s  9.33e-07s
  1  2.55e-06s  2.80e-06s
  2  8.76e-06s  7.47e-06s
  3  2.77e-05s  1.87e-05s
  4  8.17e-05s  4.48e-05s
  5  2.28e-04s  1.05e-04s
  6  6.07e-04s  2.39e-04s
  7  1.56e-03s  5.38e-04s
  8  3.92e-03s  1.19e-03s
  9  9.62e-03s  2.63e-03s
 10  2.32e-02s  5.73e-03s
 11  5.49e-02s  1.24e-02s
 12  1.29e-01s  2.68e-02s
 13  2.97e-01s  5.73e-02s
 14  6.81e-01s  1.22e-01s
 15  1.55e+00s  2.60e-01s
 16  3.49e+00s  5.50e-01s
 17  7.82e+00s  1.16e+00s
 18  1.74e+01s  2.45e+00s
 19  3.85e+01s  5.14e+00s
 20  8.49e+01s  1.08e+01s
```

Pour une matrice de taille 20, le temps estimé correspond bien au temps réel.

Serions-nous maintenant en mesure de calculer le permanent d'une matrice de taille 30 ?

[66]:
```
print(temps_estime_ryser2(30) / 3600)
```

```
4.453963815789473
```

Le calcul d'un permanent de taille 30 par notre seconde implémentation de la formule de Ryser demandera donc environ 4 heures. C'est long, mais c'est faisable.

### 1.6 6. Retour aux mariages parfaits

Le lecteur se sent probablement un peu frustré de connaître le *nombre* de mariages parfaits d'un graphe bipartite est d'être incapable de *déterminer* un mariage parfait. Comme nous l'avons déjà dit au début de ce notebook, il existe des algorithmes en temps polynomial permettant de calculer un mariage parfait, si celui-ci existe. Un de ces algorithmes repose sur l'agorithme de *Ford-Fulkerson*, dont la description demanderait à elle seule tout un article.

Nous allons dans ce qui va suivre nous contenter d'écrire une fonction naïve faisant le travail.

### 1.6.1 6.1 Trouver un mariage parfait

La fonction `est_mariage_parfait` prend en paramètre un graphe bipartite $G$ ayant $2n$ sommets. L'ensemble des sommets de $G$ est $S = X \cup Y$ où $X = \{x_0, \dots, x_{n-1}\}$ et $Y = \{y_0, \dots, y_{n-1}\}$. Les arêtes de $G$ ont une de leurs extrémités dans $X$ et l'autre extrémité dans $Y$. Rappelons que le graphe $G$ est modélisé par une liste Python $G$ de longueur $n$. Pour $i \in \bar{n}$, $G[i]$ est une liste d'entiers entre 0 et $n-1$. L'entier $j$ est dans $G[i]$ si et seulement si il existe une arête d'extrémités $x_i$ et $y_j$.

La fonction prend aussi en paramètre une permutation $\sigma$, représentée par la liste $[\sigma(0), \dots, \sigma(n-1)]$. La fonction renvoie `True` si pour tout $i \in \bar{n}$, il existe une arête de $G$ d'extrémités $x_i$ et $y_{\sigma(i)}$.

[67]:
```
def est_mariage_parfait(G, s):
    n = len(s)
    for i in range(n):
        if s[i] not in G[i]: return False
    return True
```

La fonction `trouver_mariage_parfait` recherche naïvement un mariage parfait pour le graphe bipartite $G$ de taille $2n$ en testant toutes les permutations $\sigma \in \mathfrak{S}_n$.

[68]:
```
def trouver_mariage_parfait(G):
    n = len(G)
    for s in permutations(list(range(n))):
        if est_mariage_parfait(G, s): return s
```

Testons. Dans la cellule ci-dessous, si le graphe bipartite aléatoire $G$ ne possède pas de mariage parfait, la variable $s$ vaut `None`. Dans ce cas, réévaluer la cellule jusquà ce que $s$ soit une liste.

[69]:
```
G = graphe_bipartite_alea(9)
s = trouver_mariage_parfait(G)
print(s)
```

```
[3, 2, 6, 1, 7, 8, 4, 0, 5]
```

Réécrivons une fonction de tracé de graphe en faisant aussi passer en paramère un mariage parfait du graphe. La fonction affiche les arêtes du mariage parfait en gras et les autres arêtes en pointillés fins.

[70]:
```
def plot_graph2(G, s):
    n = len(G)
    xs = list(range(n))
    for i in range(n):
        for j in G[i]: plt.plot([i, j], [0, 1], '--k', lw=0.5)
    if s != None:
        for i in range(n):
            plt.plot([i, s[i]], [0, 1], 'k', lw=1)
    plt.plot(xs, n * [1], 'or', ms=10)
```

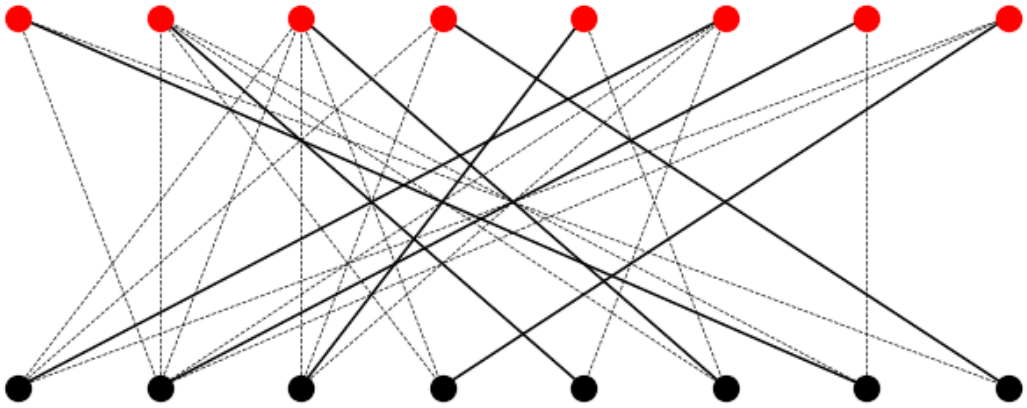
```
    plt.plot(xs, n * [0], 'ok', ms=10)

    plt.axis('off')
```

Évaluer la cellule ci-dessous jusqu'à ce que le graphe aléatoire $G$ possède un mariage parfait, puis passer à la cellule suivante qui fait le tracé …

[113]:
```
G = graphe_bipartite_alea(8, 0.3)
s = trouver_mariage_parfait(G)
print(s)
```

```
[5, 6, 4, 7, 1, 2, 0, 3]
```

[114]:
```
plot_graph2(G, s)
```



### 1.6.2 6.2 Trouver tous les mariages parfaits

Pour trouver **tous** les mariages parfaits, rien de plus facile ni de plus inefficace. Il suffit de modifier un peu le code de `trouver_mariage_parfait`.

[73]:
```
def tous_mariage_parfaits(G):
    n = len(G)
    mp = []
    for s in permutations(list(range(n))):
        if est_mariage_parfait(G, s): mp.append(s)
    return mp
```

Testons sur un graphe bipartite ayant 16 sommets.

[74]:
```
G = graphe_bipartite_alea(8, 0.4)
mp = tous_mariage_parfaits(G)
print(perm_ryser2(matrice_biadjacence(G)), len(mp))
```

```
for s in mp: print(s)
```

```
0 0
```

## 1.7 7. Pour terminer - Couverture d'un graphe par des cycles

Cette dernière section a deux objectifs.

- Montrer que la notion de permanent permet de compter autre chose que des mariages parfaits.
- Calculer explicitement le permanent de quelques matrices particulières.

### 1.7.1 7.1 Introduction

Si $G = (S, A)$ est un graphe, un *sous-graphe* de $G$ est un graphe $G' = (S', A')$ tel que $S' \subseteq S$ et $A' \subseteq A$. Notons que comme $G'$ est un graphe, ceci implique $A' \subseteq S' \times S'$.

Si $G = (S, A)$ est un graphe et $x, y \in S$, notons $x \to y$ lorsque $(x, y) \in A$.

**Définition.** Soit $k \geq 1$. Un *cycle de longueur* $k$ du graphe $G$ est un sous-graphe $\Gamma = (S', A')$ de $G$ tel que $|S'| = k$ et il existe une numérotation $S' = \{x_0, \dots, x_{k-1}\}$ de l'ensemble des sommets de $\Gamma$ telle que, dans le graphe $\Gamma$,

- $x_0 \to x_1 \to x_{k-1} \to x_0$.
- Les seules arêtes de $\Gamma$ sont celles données ci-dessus ($\Gamma$ possède donc $k$ arêtes).

L'ensemble $S'$ est le *support* du cycle $\Gamma$. Nous noterons $\Gamma = [x_0, \dots, x_{n-1}]$. Remarquons la non-unicité de cette notation : un cycle n'a pas de « début ».

**Exemples.** Un cycle de longueur 1 est un graphe $\Gamma = [x]$, c'est à dire une boucle sur le sommet $x$. Un cycle de longueur 2 est un graphe $\Gamma = [x, y]$ où $x, y$ sont deux sommets distincts.

**Définition.** Une *couverture* du graphe $G$ (par des cycles de supports disjoints) est un ensemble de cycles de $G$ dont les supports forment une partition de l'ensemble des sommets de $G$.

**Question.** Soit $G$ un graphe. Combien $G$ possède-t-il de couvertures ?

### 1.7.2 7.2 Matrice d'adjacence

Soit $G$ un graphe orienté. Numérotons les sommets de $G$ en notant $S = \{x_0, \dots, x_{n-1}\}$ l'ensemble des sommets du graphe.

**Définition.** La *matrice d'adjacence* de $G$ est la matrice $A^G \in \mathcal{M}_n(\{0, 1\})$ définie, pour tous $i, j \in \bar{n}$ par $A_{ij} = 1$ si $x_i \to x_j$, et $A_{ij} = 0$ sinon.

### 1.7.3 7.3 Quelques exemples

Donnons quelques exemples de graphes très classiques.

**7.3.1 Le graphe complet $K_n$** Le graphe complet à $n$ sommets, que l'on note classiquement $K_n$, a $\bar{n}$ pour ensemble des sommets. Pour tous $i, j \in \bar{n}$ tels que $i \neq j$, $i \to j$. La fonction `complet` renvoie la matrice d'adjacence d'un tel graphe.

[75]:
```
def complet(n):
    A = matrice(n, 1)
    for i in range(n): A[i][i] = 0
    return A
```

Voici la matrice d'adjacence de $K_5$.

[76]:
```
A = complet(5)
print_mat(A)
```

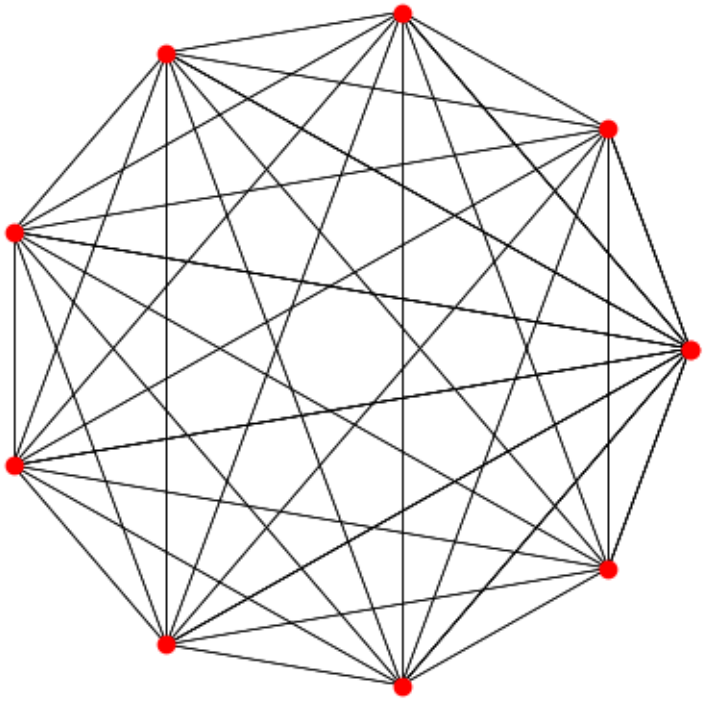
```
+----+----+----+----+----+
|  0 |  1 |  1 |  1 |  1 |
+----+----+----+----+----+
|  1 |  0 |  1 |  1 |  1 |
+----+----+----+----+----+
|  1 |  1 |  0 |  1 |  1 |
+----+----+----+----+----+
|  1 |  1 |  1 |  0 |  1 |
+----+----+----+----+----+
|  1 |  1 |  1 |  1 |  0 |
+----+----+----+----+----+
```

Voici une illustration graphique.

[77]:
```
def plot_Kn(n):
    ts = [2 * k * math.pi / n for k in range(n + 1)]
    xs = [math.cos(t) for t in ts]
    ys = [math.sin(t) for t in ts]
    for i in range(n + 1):
        for j in range(n + 1):
            if i != j:
                plt.plot([xs[i], xs[j]], [ys[i], ys[j]], 'k', lw=0.5)
    plt.plot(xs, ys, 'or')
    plt.axis('off')
```

[78]:
```
plt.rcParams['figure.figsize'] = (5, 5)
```

[79]:
```
plot_Kn(9)
```



**7.3.2 Le cycle $C_n$** Soit $n \geq 2$. Le cycle (non orienté) à $n$ sommets $C_n$ a $\bar{n}$ pour ensemble des sommets. Ses arêtes sont

- $0 \to 1 \to ... \to n-1 \to 0$
- $0 \to n-1 \to ... \to 1 \to 0$

Remarquons que $C_n$ n'est pas un cycle, au sens défini plus haut : il n'est pas *orienté*.

[80]:
```
def cycle(n):
    A = matrice(n, 0)
    for i in range(n):
        if i > 0: A[i][i - 1] = 1
        if i < n - 1: A[i][i + 1] = 1
    A[n - 1][0] = 1
    A[0][n - 1] = 1
    return A
```

Voici la matrice d'adjacence de $C_6$.

[81]:
```
A = cycle(6)
print_mat(A)
```

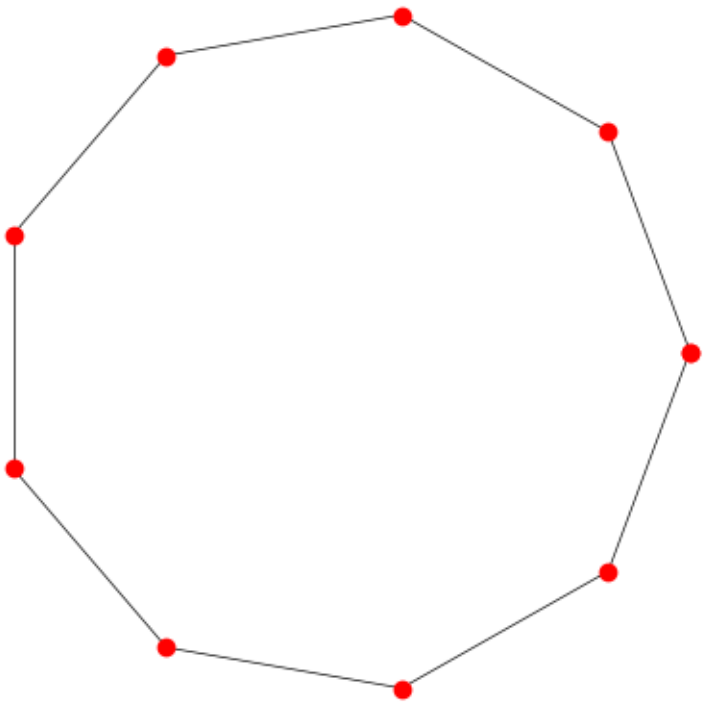
```
+----+----+----+----+----+----+
```

```
|  0 |  1 |  0 |  0 |  0 |  1 |
+----+----+----+----+----+----+
|  1 |  0 |  1 |  0 |  0 |  0 |
+----+----+----+----+----+----+
|  0 |  1 |  0 |  1 |  0 |  0 |
+----+----+----+----+----+----+
|  0 |  0 |  1 |  0 |  1 |  0 |
+----+----+----+----+----+----+
|  0 |  0 |  0 |  1 |  0 |  1 |
+----+----+----+----+----+----+
|  1 |  0 |  0 |  0 |  1 |  0 |
+----+----+----+----+----+----+
```

Voici une illustration graphique.

[82]:
```
def plot_Cn(n):
    ts = [2 * k * math.pi / n for k in range(n + 1)]
    xs = [math.cos(t) for t in ts]
    ys = [math.sin(t) for t in ts]
    for i in range(n):
        plt.plot([xs[i], xs[i + 1]], [ys[i], ys[i + 1]], 'k', lw=0.5)
    plt.plot(xs, ys, 'or')
    plt.axis('off')
```

[83]:
```
plot_Cn(9)
```



**7.3.3 Le graphe bipartite complet $K_{n,n}$** Soit $n \geq 1$. Le graphe bipartite complet $K_{n,n}$ possède $2n$ sommets qui sont les entiers de $\overline{2n}$. Pour tout $i \in \overline{n}$ et tout $j \in [\![n, 2n-1]\!]$, $i \to j$ et $j \to i$.

[84]:
```
def bipartite_complet(n):
    A = matrice(2 * n, 0)
    for i in range(n):
        for j in range(n, 2 * n):
            A[i][j] = 1
            A[j][i] = 1
    return A
```

Voici la matrice d'adjacence de $K_{3,3}$.

[85]:
```
A = bipartite_complet(3)
print_mat(A)
```

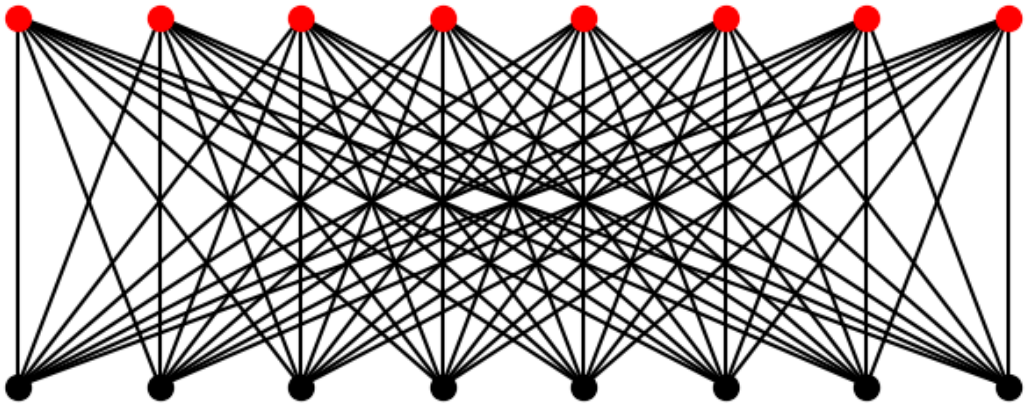
```
+----+----+----+----+----+----+
|  0 |  0 |  0 |  1 |  1 |  1 |
+----+----+----+----+----+----+
|  0 |  0 |  0 |  1 |  1 |  1 |
+----+----+----+----+----+----+
|  0 |  0 |  0 |  1 |  1 |  1 |
+----+----+----+----+----+----+
|  1 |  1 |  1 |  0 |  0 |  0 |
+----+----+----+----+----+----+
|  1 |  1 |  1 |  0 |  0 |  0 |
+----+----+----+----+----+----+
|  1 |  1 |  1 |  0 |  0 |  0 |
+----+----+----+----+----+----+
```

Voici une illustration graphique.

[86]:
```
def plot_Knn(n):
    G = n * [None]
    for i in range(n):
        G[i] = list(range(n))
    plot_graph(G)
```

[87]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

[88]:
```
plot_Knn(8)
```



**7.3.4 Le cube $Q_n$** Terminons par un exemple un peu plus compliqué, celui de l'hypercube $Q_n$. Il y a de nombreuses façons de présenter celui-ci, nous choisissons la plus simple pour ce que nous voulons faire. Pour tout $n \geq 0$, l'ensemble des sommets de $Q_n$ est $S = \overline{2^n}$. Pour tous $i, j \in S$, $i \to j$ (et $j \to i$) si et seulement si les entiers $i$ et $j$ diffèrent d'un seul bit.

La fonction `bits` prend en paramètres deux entiers $x$ et $n$ tels que $x < 2^n$. Elle renvoie une liste $[b_0, \dots, b_{n-1}]$ telle que les $b_k$ valent 0 ou 1 et

$$\sum_{k=0}^{n-1} b_k 2^k = x$$

[89]:
```
def bits(x, n):
    s = []
    for k in range(n):
        s.append(x % 2)
        x = x // 2
    return s
```

La fonction `nombre_bits_distincts` prend en paramètres deux listes $x$ et $y$ de même longueur. Elle renvoie le nombre de termes distincts des deux listes.

[90]:
```
def nombre_bits_distincts(x, y):
    n = len(x)
    c = 0
    for k in range(n):
        if x[k] != y[k]: c += 1
    return c
```

Enfin, la fonction `cube` renvoie la matrice d'adjacence de $Q_n$.

[91]:
```
def cube(n):
    p2n = 2 ** n
    A = matrice(p2n, 0)
    for i in range(p2n):
        bi = bits(i, n)
        for j in range(p2n):
            bj = bits(j, n)
            if nombre_bits_distincts(bi, bj) == 1:
                A[i][j] = 1
    return A
```

Voici la matrice d'adjacence de $Q_3$.

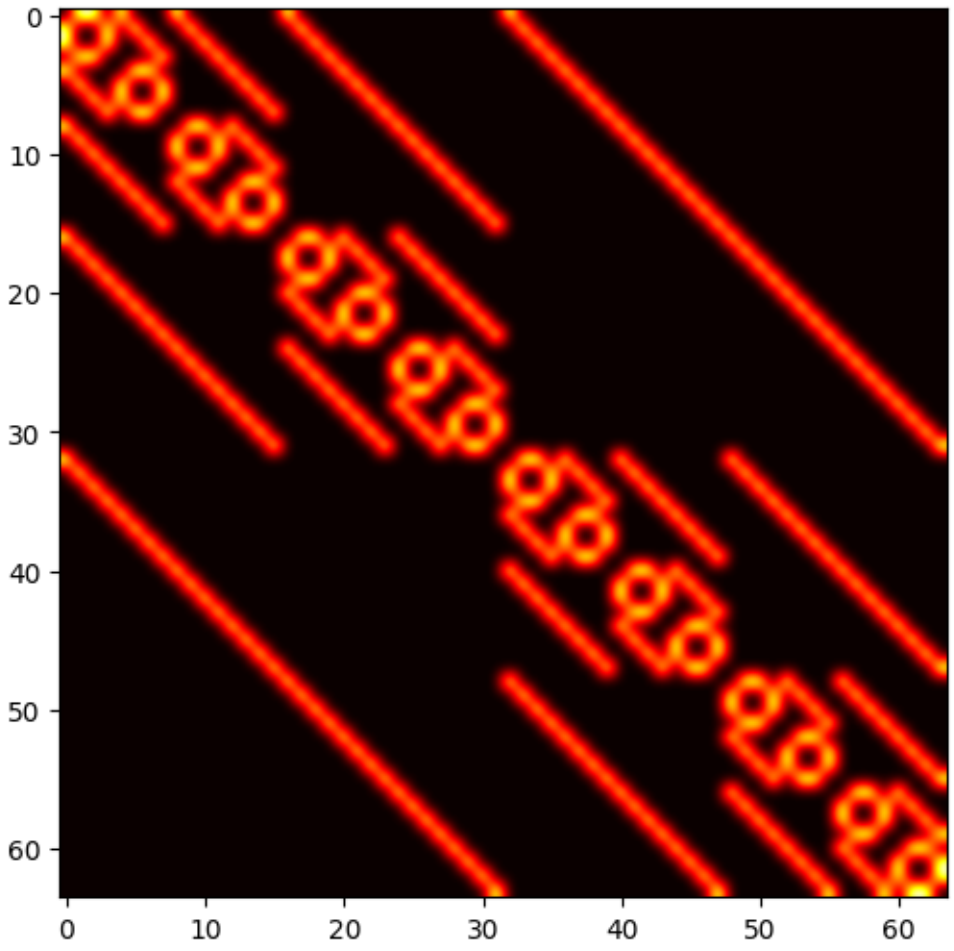
[92]:
```
print_mat(cube(3))
```

```
+---+---+---+---+---+---+---+---+
|  0 |  1 |  1 |  0 |  1 |  0 |  0 |  0 |
+---+---+---+---+---+---+---+---+
|  1 |  0 |  0 |  1 |  0 |  1 |  0 |  0 |
+---+---+---+---+---+---+---+---+
|  1 |  0 |  0 |  1 |  0 |  0 |  1 |  0 |
+---+---+---+---+---+---+---+---+
|  0 |  1 |  1 |  0 |  0 |  0 |  0 |  1 |
+---+---+---+---+---+---+---+---+
|  1 |  0 |  0 |  0 |  0 |  1 |  1 |  0 |
+---+---+---+---+---+---+---+---+
|  0 |  1 |  0 |  0 |  1 |  0 |  0 |  1 |
+---+---+---+---+---+---+---+---+
|  0 |  0 |  1 |  0 |  1 |  0 |  0 |  1 |
+---+---+---+---+---+---+---+---+
|  0 |  0 |  0 |  1 |  0 |  1 |  1 |  0 |
+---+---+---+---+---+---+---+---+
```

Voici une vision sympa de la matrice d'adjacence de $Q_6$.

[93]:
```
plt.rcParams['figure.figsize'] = (6, 6)
A = cube(6)
plt.imshow(A, cmap='hot', interpolation='bicubic')
```

[93]:
```
<matplotlib.image.AxesImage at 0x11aaad750>
```




### 1.7.4 7.4 Le lien avec le permanent

Soit $G = (S, A)$ un graphe. Notons $S = \{x_0, \dots, x_{n-1}\}$. À chaque couverture $\mathcal{C}$ de $G$ par des cycles on peut associer la permutation $\sigma \in \mathfrak{S}_n$ définie pour $i, j \in \bar{n}$ par $\sigma(i) = j$ si et seulement si il existe un cycle de la couverture tel que $x_i \to x_j$ dans ce cycle. On définit ainsi une application $F : \mathcal{C} \longmapsto \sigma$ de l'ensemble des couvertures de $G$ vers $\mathfrak{S}_n$.

**Proposition.** $F$ est une injection.

**Démonstration.** Preuve « facile ».

**Proposition.** Soit $\sigma \in \mathfrak{S}_n$. La permutation $\sigma$ a un antécédent par $F$ si et seulement si pour tout $i \in \bar{n}$, $i \to \sigma(i)$ dans $G$.

**Démonstration.** Par définition de $F$, cette condition est évidemment nécessaire. Inversement, supposons la condition réalisée. Décomposons $\sigma = \gamma_1 \dots \gamma_r$ en produit de cycles de supports disjoints $\gamma_1, \dots, \gamma_r$. Pour $k \in [\![1, r]\!]$, écrivons $\gamma_k = (i_1 \dots i_p)$. Le cycle (en tant que permutation) $\gamma_k$ correspond à un cycle $\Gamma_k = [i_1, \dots, i_p]$ (en tant que graphe) du graphe $G$. L'ensemble $\mathcal{C} = \{\Gamma_1, \dots, \Gamma_r\}$

est alors une couverture de $G$ par des cycles, et $F(\mathcal{C}) = \sigma$. Ainsi, $\sigma$ a un antécedent par $F$.

Notons $A$ la matrice d'adjacence de $G$.

**Corollaire.** Soit $\sigma \in \mathfrak{S}_n$. La permutation $\sigma$ a un antécédent par $F$ si et seulement si $\prod_{i=0}^{n-1} A_{i\sigma(i)} = 1$.

**Démonstration.** Par la proposition précédente, $\sigma$ a un antécédent par $F$ si et seulement si pour tout $i \in \bar{n}$, $A_{i\sigma(i)} = 1$, ce qui équivaut à dire que le produit est égal à 1.

**Corollaire.** Le nombre de couvertures de $G$ est $\mathrm{perm}(A)$.

**Démonstration.** On a

$$\mathrm{perm}(A) = \sum_{\sigma \in \mathfrak{S}_n} \prod_{i=0}^{n-1} A_{i\sigma(i)}$$

Les termes de cette somme valent 0 ou 1. Ils valent 1 si et seulement si $\sigma$ a un (unique) antécédent par $F$, d'où le résultat.

### 1.7.5 7.5 Retour aux exemples

Nous pouvons maintenant revenir à nos exemples de graphes. Pour chacun d'entre-eux, déterminons le nombre de couvertures du graphe.

**7.5.1 Graphes complets** Tout d'abord, les graphes complets.

[94]:
```
for n in range(11):
    print(n, perm_ryser2(complet(n)))
```

```
0 1
1 0
2 1
3 2
4 9
5 44
6 265
7 1854
8 14833
9 133496
10 1334961
```

La matrice d'adjacence du graphe complet $K_n$ est

$$A_n = \begin{pmatrix} 0 & 1 & 1 & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 1 & 1 & \dots & 0 \end{pmatrix}$$

Érant donnée $\sigma \in \mathfrak{S}_n$, on a $\prod_{i=0}^{n-1} A_{i\sigma(i)} = 1$ si et seulement si pour tout $i \in \bar{n}$, $\sigma(i) \neq i$, c'est à dire lorsque $\sigma$ est un *dérangement*. Sinon, ce produit est nul. Ainsi, $\mathrm{perm}(A)$ est le nombre de dérangements de $\mathfrak{S}_n$. On peut ainsi démontrer (nous ne le ferons pas ici) que

$$\mathrm{perm}(A_n) = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$$

Cette quantité équivaut, lorsque $n$ tend vers l'infini, à $\frac{1}{e}n!$.

[95]:
```
def nombre_derangements(n):
    s = 0
    for k in range(n + 1):
        s += (-1) ** k * math.factorial(n) // math.factorial(k)
    return s
```

[96]:
```
for n in range(11):
    print('%3d%10d%10d' % (n, perm_ryser2(complet(n)), nombre_derangements(n)))
```

```
  0         1         1
  1         0         0
  2         1         1
  3         2         2
  4         9         9
  5        44        44
  6       265       265
  7      1854      1854
  8     14833     14833
  9    133496    133496
 10   1334961   1334961
```

**7.5.2 Cycles** Passons aux cycles.

[97]:
```
for n in range(2, 15):
    print(n, perm_ryser2(cycle(n)))
```

```
2 1
3 2
4 4
5 2
6 4
7 2
8 4
9 2
10 4
11 2
12 4
```

```
13 2
14 4
```

Par exemple, le nombre de couvertures de $C_6$ est égal à 4. En effet, les couvertures de $C_6$ sont

- $\{[0,1,2,3,4,5]\}$.
- $\{[5,4,3,2,1,0]\}$.
- $\{[0,1],[2,3],[4,5]\}$.
- $\{[1,2],[3,4],[5,0]\}$.

On laisse au lecteur le soin de montrer que pour tout $n \geq 3$,

- Si $n$ est pair, $C_n$ possède 4 couvertures.
- Si $n$ est impair, $C_n$ possède 2 couvertures.

**Exercice.** Montrer que si $n$ est pair, le graphe $C_n$ est bipartite.

**7.5.3 Graphes bipartites complets** Voici le nombre de couvertures de $K_{n,n}$ pour $1 \leq n \leq 9$. On affiche aussi dans la dernière colonne le nombre de mariages parfaits.

[98]:
```
def Knn(n):
    G = n * [None]
    for i in range(n):
        G[i] = list(range(n))
    return G
```

[99]:
```
for n in range(1, 10):
    print('%3d%15d%15d' % (n, perm_ryser2(bipartite_complet(n)),␣
 ↪perm_ryser2(matrice_biadjacence(Knn(n)))))
```

```
  1              1              1
  2              4              2
  3             36              6
  4            576             24
  5          14400            120
  6         518400            720
  7       25401600           5040
  8     1625702400          40320
  9   131681894400         362880
```

Cela ne saute peut-être pas aux yeux, mais les éléments de la dernière colonne sont les carrés de ceux de la colonne précédente. Pourquoi ?

**Proposition.** Soit $G$ un graphe bipartite. Soient $A$ la matrice d'adjacence de $G$ et $B$ sa matrice de biadjacence. On a

$$\mathrm{perm}(A) = \mathrm{perm}(B)^2$$

**Démonstration.** On a la décomposition par blocs

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix}$$

Par des échanges de colonnes, le permanent de cette matrice est aussi celui de

$$A' = \begin{pmatrix} B & 0 \\ 0 & B^T \end{pmatrix}$$

Nous avons vu que le permanent d'une matrice triangulaire est le produit de ses coefficients diagonaux. On peut en fait montrer que ce résultat s'étend à des matrices triangulaires par blocs, ce qui est le cas de $A'$. Ainsi,

$$\mathrm{perm}(A) = \mathrm{perm}(A') = \mathrm{perm}(B)\ \mathrm{perm}(B^T) = \mathrm{perm}(B)^2$$

Remarquons que la matrice $B$ de biadjacence de $K_{n,n}$ ne contient que des 1. Ainsi, $\mathrm{perm}(B) = n!$. De là, le graphe $K_{n,n}$ possède $n!^2$ couvertures.

**7.5.4 Hypercubes** Terminons par le nombre de couvertures des cubes. Attention, le nombre de sommets de $Q_n$ est $2^n$. Comme $2^5 = 32$ et que nous sommes incapables de calculer un permanent de taille 32 en moins de 16 heures, arrêtons-nous à $n = 4$ … pour l'instant.

[100]:
```
for n in range(1, 5):
    print(n, perm_ryser2(cube(n)))
```

```
1 1
2 4
3 81
4 73984
```

Le lecteur aura remarqué que les nombres de la dernière colonne sont des carrés. C'est en fait vrai pour **tout** $n \geq 1$ :

**Proposition.** Pour tout $n \geq 1$, $Q_n$ est un graphe bipartite.

**Démonstration.** Soit $S$ l'ensemble des sommets de $Q_n$. Écrivons $S = X \cup Y$ où $X$ est l'ensemble des entiers $i \in [\![0, 2^n - 1]\!]$ ayant un nombre pair de bits égaux à 1 et $Y$ est l'ensemble des entiers $i \in [\![0, 2^n - 1]\!]$ ayant un nombre impair de bits égaux à 1. Avec notre définition de $Q_n$, il est immédiat que les arêtes de $Q_n$ ont pour extrémités un élément de $X$ et un élément de $Y$. Ainsi, $Q_n$ est bipartite.

**Remarque.** Le lecteur est invité à montrer que $|X| = |Y| = 2^{n-1}$.

Maintenant que nous savons que $Q_n$ est bipartite, nous pouvons obtenir le nombre de couvertures de $Q_5$. En effet, la matrice de biadjacence de $Q_5$ est de taille 16, taille tout à fait abordable pour un calcul de permanent avec la formule de Ryser.

Avant tout, nous devons évidemment écrire une fonction qui renvoie la matrice de biadjacence en question. Le code ci-dessous est peu efficace mais suffira pour $n = 5$.

Tout d'abord, la fonction `somme_bits` renvoie la somme des $n$ bits de l'entier $x \in [\![0, 2^n - 1]\!]$.

[101]:
```
def somme_bits(x, n):
    return sum(bits(x, n))
```

[102]:
```
print([x for x in range(16) if somme_bits(x, 4) % 2 == 0])
print([x for x in range(16) if somme_bits(x, 4) % 2 == 1])
```

```
[0, 3, 5, 6, 9, 10, 12, 15]
[1, 2, 4, 7, 8, 11, 13, 14]
```

La fonction `entier` prend en paramètre une liste $s = [b_0, \dots, b_{n-1}]$ de bits. Elle renvoie l'entier

$$\sum_{k=0}^{n-1} b_k 2^k$$

[103]:
```
def entier(s):
    n = len(s)
    x = 0
    for k in range(n):
        x += s[k] * 2 ** k
    return x
```

Voici une fonction très mal écrite. `indice(x, s)` renvoie l'indice de l'objet $x$ dans la liste $s$.

[104]:
```
def indice(x, s):
    i = 0
    while s[i] != x: i = i + 1
    return i
```

Enfin, la matrice de biadjacence du cube $Q_n$.

[105]:
```
def matrice_biadjacence_cube(n):
    A = matrice(2 ** (n - 1), 0)
    xs = [x for x in range(2 ** n) if somme_bits(x, n) % 2 == 0]
    ys = [x for x in range(2 ** n) if somme_bits(x, n) % 2 == 1]
    for i in range(2 ** (n - 1)):
        s = bits(xs[i], n)
        for k in range(n):
            s[k] = 1 - s[k]
            y = entier(s)
            j = indice(y, ys)
            A[i][j] = 1
            s[k] = 1 - s[k]
    return A
```

Testons tout d'abord pour $n = 4$.

[106]:
```
A = matrice_biadjacence_cube(4)
print(perm_ryser2(A) ** 2)
print_mat(A)
```

```
73984
+----+----+----+----+----+----+----+----+
|  1 |  1 |  1 |  0 |  1 |  0 |  0 |  0 |
+----+----+----+----+----+----+----+----+
|  1 |  1 |  0 |  1 |  0 |  1 |  0 |  0 |
+----+----+----+----+----+----+----+----+
|  1 |  0 |  1 |  1 |  0 |  0 |  1 |  0 |
+----+----+----+----+----+----+----+----+
|  0 |  1 |  1 |  1 |  0 |  0 |  0 |  1 |
+----+----+----+----+----+----+----+----+
|  1 |  0 |  0 |  0 |  1 |  1 |  1 |  0 |
+----+----+----+----+----+----+----+----+
|  0 |  1 |  0 |  0 |  1 |  1 |  0 |  1 |
+----+----+----+----+----+----+----+----+
|  0 |  0 |  1 |  0 |  1 |  0 |  1 |  1 |
+----+----+----+----+----+----+----+----+
|  0 |  0 |  0 |  1 |  0 |  1 |  1 |  1 |
+----+----+----+----+----+----+----+----+
```

Nous retrouvons pour $\mathrm{perm}(A)^2$ la valeur du permanent de la matrice d'adjacence de $Q_4$.

Nous pouvons maintenant calculer le nombre de mariages parfaits et le nombre de couvertures de $Q_5$.

[107]:
```
A = matrice_biadjacence_cube(5)
p5 = perm_ryser2(A)
print('Nombre de mariages parfaits :', p5)
print('Nombre de couvertures       :', p5 ** 2)
```

```
Nombre de mariages parfaits : 589185
Nombre de couvertures       : 347138964225
```

**Bilan :** le graphe $Q_5$ possède environ 350 milliards de couvertures.

Allons, terminons ce notebook par un dessin du graphe $Q_n$. Mais ce ne sera peut-être pas ce à quoi le lecteur s'attend …

Tout d'abord, voici une fonction qui renvoie le graphe bipartite $Q_n$.

[108]:
```
def graphe_cube(n):
    m = 2 ** (n - 1)
    A = matrice_biadjacence_cube(n)
    G = m * [None]
    for i in range(m): G[i] = []
    for i in range(m):
        for j in range(m):
```

```
            if A[i][j] == 1: G[i].append(j)
    return G
```

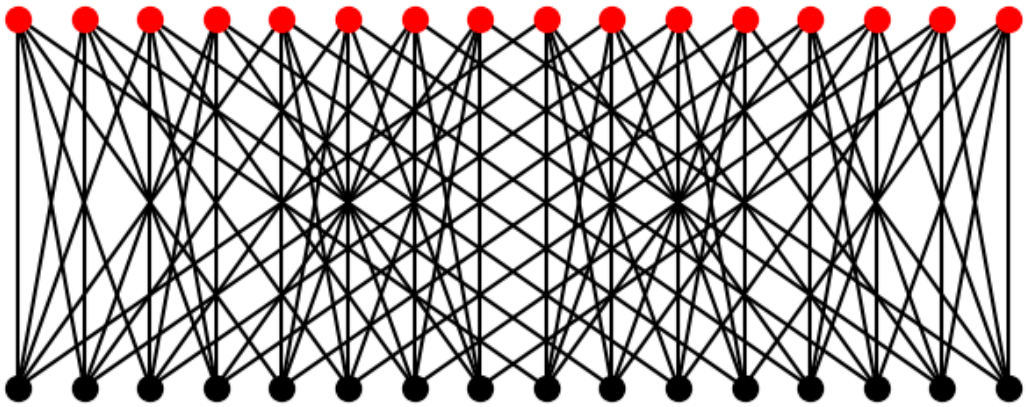
[109]:
```
print(graphe_cube(3))
```

```
[[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]]
```

Et maintenant, traçons $Q_5$.

[110]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

[111]:
```
plot_graph(graphe_cube(5))
```



Peut-être vous attendiez-vous à voir un hypercube en dimension 5 ? Ce que nous voyons ici est **une** des façons de dessiner $Q_5$, celle qui met en évidence sa structure de graphe bipartite. Il existe de nombreuses façons de dessiner un graphe donné, chacune ayant ses avantages et ses inconvénients, mais ceci est un autre sujet …