Enum_Permutations

April 1, 2023

1 Ordonner et énumérer les permutations

Marc Lorenzi

1er avril 2023

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

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

L'ensemble $\mathfrak{S}_n$ des permutations de l'ensemble $[|0, n-1|]$ est un ensemble de cardinal $n!$. Énumérer les permutations de $\mathfrak{S}_n$ demande donc un temps au moins de l'ordre de $n!$. Si l'on désire stocker ces permutations, un stockage naïf demandera un espace en $\mathcal{O}(n!\, n)$, puisque chaque permutation nécessite un espace en $\mathcal{O}(n)$. Nous allons dans ce notebook décrire un algorithme permettant d'énumérer les permutations mais **sans les stocker**. Plus précisément nous allons définir une relation d'ordre total sur $\mathfrak{S}_n$ et un algorithme permmetant, pour chaque permutation $\sigma \in \mathfrak{S}_n$, de calculer la permutation suivante pour cet ordre à partir de σ seulement.

1.1 1. Opérations élémentaires sur les listes

1.1.1 1.1 Échanges et comparaisons

Nous allons dans ce qui suit décrire des algorithmes agissant sur des listes d'entiers. Les opérations que nous allons dénombrer sont :

- L'échange de deux éléments d'une liste
- La comparaison de deux éléments d'une liste

Nous qualifierons ces opérations d'opérations « élémentaires ».

Voici les deux fonctions d'échange et de comparaison.

- `echanger(s, i, j)` échange $s[i]$ et $s[j]$
- `inferieur(s, i, j)` renvoie `True` si $s[i] < s[j]$ et `False` sinon.

Remarquez les ligne `cmpt_....incr` dans le code ci-dessous. Ces lignes incrémentent des compteurs (nous en reparlons tout de suite). Les compteurs comptent donc les opérations élémentaires, à

condition bien entendu que nous nous obligions à utiliser les fonctions `echanger` et `comparer` pour effectuer ces opérations !

[3]:
```
def echanger(s, i, j):
    global cmpt_ech
    cmpt_ech.incr()
    s[i], s[j] = s[j], s[i]
```

[4]:
```
def inferieur(s, i, j):
    global cmpt_comp
    cmpt_comp.incr()
    return s[i] < s[j]
```

1.1.2 1.2 La classe `Compteur`

Voici la classe `Compteur`. Elle possède un constructeur qui crée un compteur de valeur initiale 0. Elle possède également une méthode `reset` qui remet le compteur à zéro et une méthode `incr` qui ajoute 1 au compteur. La valeur du compteur est stockée dans son champ `val`.

[5]:
```
class Compteur:

    def __init__(self): self.val = 0
    def reset(self): self.val = 0
    def incr(self): self.val += 1
```

Voici deux compteurs, appelés `cmpt_ech` et `cmpt_comp`. Comme leur nom l'indique, ils permettront dans la suite de compter des échanges et des comparaisons.

[6]:
```
cmpt_ech = Compteur()
cmpt_comp = Compteur()
```

Exemple ...

[7]:
```
for i in range(10):
    cmpt_ech.incr()
print(cmpt_ech.val)
cmpt_ech.reset()
print(cmpt_ech.val)
```

```
10
0
```

1.2 2. L'ordre lexicographique sur $\mathfrak{S}_n$

1.2.1 2.1 Un ordre total sur les permutations

Dans tout le notebook n désigne un entier naturel non nul.

Définition. : Soient $\sigma, \sigma' \in \mathfrak{S}_n$. On dit que $\sigma < \sigma'$ lorsqu'il existe $k \in [|0, n-1|]$ tel que - $\forall i < k$, $\sigma(i) = \sigma'(i)$ - $\sigma(k) < \sigma'(k)$.

Remarque. Un tel entier k est clairement unique : c'est le plus petit entier tel que $\sigma(k) \neq \sigma'(k)$.

Notation : Nous noterons également $\sigma \leq \sigma'$ lorsque $\sigma < \sigma'$ ou $\sigma = \sigma'$. Nous utiliserons également les notations $>$ et $\geq$ pour désigner les relations inverses de $<$ et $\leq$.

Proposition. La relation $\leq$ est une relation d'ordre total sur $\mathfrak{S}_n$. On l'appelle *l'ordre lexicographique.*

Démonstration.

- Cette relation est clairement réflexive.
- Soient $\sigma, \sigma' \in \mathfrak{S}_n$. Supposons $\sigma \neq \sigma'$. Soit k le plus petit entier tel que $\sigma(k) \neq \sigma'(k)$.
  - Si $\sigma(k) < \sigma'(k)$, alors $\sigma \leq \sigma'$, et $\sigma' \not\leq \sigma$.
  - Sinon, c'est le contraire. Ainsi, la relation est antisymétrique, et totale.
- Soient $\sigma, \sigma', \sigma'' \in \mathfrak{S}_n$. Supposons $\sigma \leq \sigma'$ et $\sigma' \leq \sigma''$. Si deux des trois permutations sont égales, on a clairement $\sigma \leq \sigma''$. Supposons les maintenant distinctes. Soit k le plus petit entier tel que $\sigma(k) \neq \sigma'(k)$. Soit ℓ le plus petit entier tel que $\sigma'(\ell) \neq \sigma''(\ell)$. On distingue ensuite les trois cas $k < \ell$, $k > \ell$, $k = \ell$. On conclut facilement que, dans les trois cas, $\sigma < \sigma''$.

1.2.2 2.2 Permutations et Python

Soit $\sigma \in \mathfrak{S}_n$. Nous modélisons σ en Python par la liste $[\sigma(0), \dots, \sigma(n-1)]$. L'identité id, par exemple, est représentée par la liste $[0, 1, \dots, n-1]$.

Remarque.

- id est le plus petit élément de $\mathfrak{S}_n$ pour l'ordre lexicographique.
- Le plus grand élément de $\mathfrak{S}_n$ est $\omega = [n-1, n-2, \dots, 0]$. La notation ω sera utilisée dans tout le notebook.

Dans toute la suite, nous confondrons les éléments de $\mathfrak{S}_n$ et leur représentation en Python.

[8]:
```
def id(n): return list(range(n))
def omega(n): return list(range(n - 1, -1, -1))
```

[9]:
```
print(id(10))
print(omega(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

La fonction `lexico` renvoie `True` si $s_1 \leq s_2$ et `False` sinon.

[10]:
```
def lexico(s1, s2):
    n = len(s1)
    k = 0
    while k < n and s1[k] == s2[k]: k = k + 1
```

```
    return k == n or s1[k] < s2[k]
```

[11]:
```
lexico([5, 2, 4, 3, 1, 0], [5, 2, 3, 4, 1, 0])
```

[11]: False

[12]:
```
lexico([5, 2, 3, 4, 1, 0], [5, 2, 4, 3, 1, 0])
```

[12]: True

1.2.3 2.3 Successeur d'une permutation

Définition. Soit $(E, \leq)$ un ensemble totalement ordonné. Soit $x \in E$. On suppose que $x \neq \max E$. Soit

$$x^+ = \{y \in E, x < y\}$$

Le *successeur* de x est le plus petit élément de x^+, s'il existe.

Lemme. Tout ensemble totalement ordonné fini non vide possède un plus petit et un plus grand élément.

Démonstration. Faisons une récurrence sur le cardinal n de l'ensemble E. Si $n = 1$, c'est évident. Soit donc $n \geq 1$. Supposons que tout ensemble fini de cardinal n a un plus grand élément. Soit E un ensemble de cardinal $n + 1$. Écrivons $E = E' \cup \{a\}$ où E' est de cardinal n. Par l'hypothèse de récurrence, $\max E' = a'$ existe. On vérifie alors aisément que $\max E$ existe, et est égal au plus grand des deux éléments a et a' (ordre total). L'existence du min se prouve de façon identique.

Proposition. Soit E un ensemble fini totalement ordonné. Tout élément de E différent de $\max E$ possède un successeur.

Démonstration. L'ensemble x^+ est

- fini car inclus dans E,
- non vide car $x \neq \max E$,
- totalement ordonné par l'ordre induit par celui de E.

Il a donc un plus petit élément.

Revenons à $\mathfrak{S}_n$. Prenons par exemple $n = 6$. Soit $\sigma = [2, 3, 5, 4, 1, 0]$. Quel est le successeur de σ ? C'est $[2, 4, 0, 1, 3, 5]$. Comment l'obtient-on ? Voici la recette. Comme vous n'êtes pas en train de lire un livre de cuisine, il s'agira évidemment de prouver la correction de cette recette, mais remettons cela à plus tard.

- On commence par trouver le plus long suffixe de σ qui est strictement décroissant. Sur notre exemple, c'est $[5, 4, 1, 0]$. L'indice de 5 dans la permutation σ est 2, nous l'appellerons le **pivot** de σ. Notons-le i.
- On cherche dans ce suffixe strictement décroissant le plus petit entier strictement supérieur à $\sigma[i - 1]$, c'est à dire 3 pour notre exemple. Ici, c'est 4. On échange 3 et 4, pour obtenir $\sigma_1 = [2, 4, 5, 3, 1, 0]$.

- On renverse le suffixe de cette permutation commençant à l'indice du pivot (indice 2 ici). On obtient $\sigma' = [2, 4, 0, 1, 3, 5]$, le successeur cherché.

1.3 3. Le code de la fonction `successeur`

1.3.1 3.1 Recherche du pivot

La fonction `pivot` prend en paramètre une permutation s. Elle renvoie, s'il existe, le plus grand entier $i \geq 1$ tel que $i[k-1] < s[i]$. Un tel entier existe si et seulement si $s \neq \omega$. Si $s = \omega$ la fonction renvoie 0.

[13]:
```
def pivot(s):
    n = len(s)
    i = n - 1
    while i > 0 and inferieur(s, i, i - 1): i = i - 1
    return i
```

[14]:
```
print(pivot([2, 3, 5, 4, 1, 0]))
```

```
2
```

Le pivot de ω est 0, et c'est la seule permutation à avoir 0 pour pivot.

[15]:
```
pivot(omega(6))
```

[15]: 0

Le pivot de id est $n-1$. Mais bien d'autres permutations ont $n-1$ pour pivot !

[16]:
```
pivot(id(6))
```

[16]: 5

[17]:
```
pivot([5, 4, 3, 2, 0, 1])
```

[17]: 5

1.3.2 3.2 Recherche de l'élément à échanger avec le pivot

La fonction `next_piv` prend en paramètre une permutation s. Elle calcule un entier i, qui est son pivot. Elle renvoie le couple (i, j) où j est le plus grand entier $j \geq i$ tel que $s[j] > s[i-1]$. Si s est ω, le max de $\mathfrak{S}_n$, la fonction renvoie le couple $(0, 0)$.

[18]:
```
def next_piv(s):
    i = pivot(s)
    if i > 0:
        j = len(s) - 1
```

```
        while inferieur(s, j, i - 1):
            j = j - 1
        return (i, j)
    else: return (0, 0)
```

[19]:
```
next_piv([2, 3, 5, 4, 1, 0])
```

[19]: (2, 3)

[20]:
```
next_piv(id(6))
```

[20]: (5, 5)

[21]:
```
next_piv(omega(6))
```

[21]: (0, 0)

[22]:
```
next_piv([5, 4, 3, 2, 0, 1])
```

[22]: (5, 5)

1.3.3 3.3 Renversements

La fonction `renverser` renverse la liste s à partir de l'indice i. Cette fonction modifie s sur place.

[23]:
```
def renverser(s, i):
    n = len(s)
    j = i
    k = n - 1
    while j < k:
        echanger(s, j, k)
        j = j + 1
        k = k - 1
```

[24]:
```
s = [0, 1, 2, 3, 4, 5, 6, 7, 8]
renverser(s, 2)
print(s)
```

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

1.3.4 3.4 La fonction successeur

Voici la fonction successeur. Elle prend une permutation en paramètre et renvoie la permutation suivante pour l'ordre lexicographique. Si $s = \omega$, `successeur(s)` renvoie `None`. Remarquez la ligne `s1=s[:]`. On travaille sur une copie de s. Il est en effet important pour la suite que s ne soit pas modifiée.

[25]:
```
def successeur(s):
    s1 = s[:]
    i, j = next_piv(s1)
    if i > 0:
        echanger(s1, j, i - 1)
        renverser(s1, i)
        return s1
    else: return None
```

[26]:
```
s = list(range(4))
```

[27]:
```
s = successeur(s)
print(s)
```

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

Réévaluez la cellule ci-dessus … vous verrez apparaître les permutations les unes après les autres dans l'ordre lexicographique.

1.3.5 3.5 Correction de l'algorithme

La preuve de la correction est truffée de cas. En voici une ébauche, la fin de la preuve étant laissée au lecteur.

Soit $\sigma \in \mathcal{S}_n \quad \{\omega\}$. Soit σ' définie par l'algorithme. Soit $i \geq 1$ le pivot de σ. Nous avons donc $\sigma'(\ell) = \sigma(\ell)$ pour $\ell = 0, \dots, i-2$.

Soit $\gamma > \sigma$. Il existe donc $k \in [0, n-1]$ tel que

- $\gamma(\ell) = \sigma(\ell)$ pour $\ell = 0, \dots, k-1$.
- $\gamma(k) > \sigma(k)$.

Discutons suivant la position de k par rapport à $i-1$.

- Si $k < i-1$, alors $\gamma > \sigma'$ puisque σ' et σ coïncident sur $[0, k]$.
- $k > i-1$ est impossible. On aurait en effet dans ce cas que σ et γ coïncident sur $[0, k-1]$. $\gamma(k) = \sigma(k')$ pour un entier $k' > k$, mais la suite $(\sigma(k), \dots, \sigma(n-1))$ est strictement décroissante. On aurait donc $\gamma(k) < \sigma(k)$, ce qui ne'est pas.
- Regardons enfin le cas où $k = i-1$. Soit j le plus grand indice, $j \geq i$ tel que $\sigma(j) > \sigma(i)$. Par la décroissance de σ à partir de l'indice i, on a $\gamma(k) = \sigma(i), \sigma(i+1), \dots,$ ou $\sigma(j)$, puisque les valeurs suivantes sont strictement inférieures à $\sigma(i-1)$. Si $\gamma(k) \neq \sigma(j)$, alors, comme $\sigma'(i-1) = \sigma(j)$, on a $\gamma > \sigma$. Supposons donc $\gamma(k) = \sigma(j)$. γ et σ' coïncident sur $[0, i-1]$. Soit ℓ le premier indice en lequel γ et σ' ne coïncident pas. On laisse au lecteur le soin de terminer la démonstration. Considérer les cas $\ell < j$, $\ell > j$ et $\ell = j$. Dans les deux premiers cas on conclut que $\gamma > \sigma'$. Dans le troisième cas il faut encore discuter et on trouve que $\gamma \geq \sigma'$.

En résumé, pour tout $\gamma \in \mathfrak{S}_n$, si $\gamma > \sigma$ alors $\gamma \geq \sigma'$. La permutation σ' est bien le successeur de σ dans $\mathfrak{S}_n$ pour l'ordre lexicographique.

1.4 4. Énumérer les permutations

1.4.1 4.1 Le code

La fonction `permutations` ci-dessous énumère les permutations de $\mathfrak{S}_n$ dans l'ordre lexicographique à partir de *id*, à cela près qu'à chaque itération un appel à `yield` est effectué. Explications à suivre …

[28]:
```
def permutations(n):
    s = list(range(n))
    while True:
        yield s
        s = successeur(s)
        if s == None: break
```

Un petit test ?

[29]:
```
permutations(4)
```

[29]:
```
<generator object permutations at 0x11347c7b0>
```

Euh, oui, bon, et alors ? Un second test ?

[30]:
```
permutations(1000)
```

[30]:
```
<generator object permutations at 0x11347c890>
```

1.4.2 4.2 Générateurs

Là on n'y croit plus du tout. Le cardinal de $\mathfrak{S}_{1000}$ est largement supérieur au nombre d'atomes dans l'univers. Mais alors, que fait la fonction `permutations` ? Elle renvoie un *générateur*. Exemple de création d'un générateur en Python ? La fonction `range`.

[31]:
```
range(10 ** 10)
```

[31]:
```
range(0, 10000000000)
```

A-t-on créé un objet avec 10^{10} entiers dedans ? Pas du tout. Mais si on applique certaines fonctions comme `list` à cet objet, ou que l'on écrit une ligne du genre `for i in range(10 ** 10):`, il va *engendrer* tous les entiers de 0 à $10^{10} - 1$.

Il en va de même pour notre fonction `permutations` :

[32]:
```
for s in permutations(4):
    print(s)
```

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

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

Remarquons que le cardinal de $\mathfrak{S}_n$ est $n!$. Demander *toutes* les permutations prend donc un temps au minimum « factoriel ». Alors soyons modestes.

[33]:
```
%%time
len(list(permutations(9)))
```

```
CPU times: user 1.98 s, sys: 33.1 ms, total: 2.01 s
Wall time: 2.01 s
```

[33]:
```
362880
```

Un appel à `list(permutations(10))` prendrait au moins 10 fois plus de temps. Et si l'on met 11 au lieu de 9, au moins 110 fois plus de temps ! Ce temps prohibitif n'est pas dû au fait que notre fonction `successeur` est mauvaise, mais au fait que le nombre d'objets à calculer est lui-même prohibitif. En fait nous allons voir que `successeur` est une TRÈS bonne fonction.

1.5 5. La complexité de la fonction `permutations`

1.5.1 5.1 Meilleur cas et pire cas de la fonction `successeur`

Soit $s \in \mathfrak{S}_n$, $s \neq \omega$.

- Soit i le pivot de s. La recherche de i nécessite $n - i$ comparaisons.
- La recherche du j maximal tel que $i \leq j$ et $s[i-1] < s[j]$ nécessite $n - j$ comparaisons.
- L'échange de $s[i-1]$ et $s[j]$ nécessite … 1 échange.
- Le renversement de la queue de liste nécessite $\lfloor \frac{n-i}{2} \rfloor$ échanges.

Nous avons donc au total $2n - i - j$ comparaisons et $\lfloor \frac{n-i}{2} \rfloor + 1$ échanges.

Si $i = n - 1$, alors j aussi et il y a 2 comparaisons et 1 échange, qui sont clairement minimaux.

Comme $s \neq \omega$, on a $i \geq 1$. Le pire cas survient lorsque $i = j = 1$, auquel cas il y a $2n - 2$ comparaisons et $\lfloor \frac{n-1}{2} \rfloor + 1 = \lfloor \frac{n+1}{2} \rfloor$ échanges.

La complexité de la fonction `successeur` est donc $\mathcal{O}(1)$ en meilleur cas et $\mathcal{O}(n)$ en pire cas (en termes d'échanges et/ou de comparaisons).

La complexité de la fonction `permutations` est ainsi $\mathcal{O}(n!\, n)$ puisqu'il y a $n!$ appels à la fonction successeur.

Nous allons maintenant montrer qu'on peut dire beaucoup mieux que cela.

1.5.2 5.2 Nombre d'échanges

Proposition. la complexité de `permutations` en nombre d'échanges est équivalente, lorsque n tend vers l'infini, à

$$n! \cosh 1$$

Démonstration. Notons I_n le nombre d'échanges d'éléments de liste effectués lors de l'appel `permutations(n)`. On a $I_1 = 0$ et, pour tout $n \geq 2$,

$$I_n = nI_{n-1} + (n-1) \left\lfloor \frac{n+1}{2} \right\rfloor$$

En effet :

- Pour $k = 0, \dots, n-1$, la fonction énumère les permutations s telles que $s[0] = k$, ce qui demande, pour chaque valeur de k, I_{n-1} échanges.
- Pour $k = 0, \dots, n-2$, la fonction calcule le successeur de la plus grande permutation s telle que $s[0] = k$. Cette permutation est $s = [k, n-1, n-2, \dots, 0]$ (où k n'apparaît pas dans les pointillés). Le pivot de s est $i = 1$, Le renversement nécessite donc $\lfloor \frac{n-1}{2} \rfloor$, et passer de s à son successeur nécessite ainsi $\lfloor \frac{n+1}{2} \rfloor$ échanges.
- Pour $k = n - 1$, le calcul du successeur de ω (enfin, plutôt l'échec de ce calcul) ne nécessite aucun échange.

Une petite vérification s'impose.

[34]:
```
def nombre_echanges0(n):
    if n == 1: return 0
    else:
        return n * nombre_echanges0(n - 1) + (n - 1) * ((n + 1) // 2)
```

[35]:
```
n = 6
cmpt_ech.reset()
s = list(permutations(n))
print(cmpt_ech.val)
```

```
print(nombre_echanges0(n))
```

```
1107
1107
```

Tout va pour le mieux. Attaquons nous au calcul explicite de I_n.

Posons

$$S_n = I_n + \left\lfloor \frac{n+1}{2} \right\rfloor$$

On laisse au lecteur le soin de vérifier que $S_1 = 1$ et, pour tout $n \geq 2$,

$$S_n = n(S_{n-1} + \varepsilon_{n-1})$$

où $\varepsilon_n = 0$ si n est impair et 1 si n est pair. On montre alors facilement par récurrence sur n que

$$S_n = n! \left(1 + \frac{1}{2!} + \frac{1}{4!} + \frac{1}{6!} + ... + \frac{1}{2\lfloor (n-1)/2 \rfloor}\right)$$

d'où

$$I_n = n! \sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} \frac{1}{(2j)!} - \left\lfloor \frac{n+1}{2} \right\rfloor$$

Remarquons que $\sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} \frac{1}{(2j)!}$ est une somme partielle de série, qui tend, lorsque n tend vers l'infini, vers

$$\sum_{j=0}^{\infty} \frac{1}{(2j)!} = \cosh 1 \simeq 1.54308$$

Ainsi,

$$I_n \sim n! \cosh 1$$

[36]:
```
math.cosh(1)
```

[36]:
```
1.5430806348152437
```

Testons tout cela.

[37]:
```
def nombre_echanges(n):
    p = (n + 1) // 2
    s = 0
    for j in range(p):
```

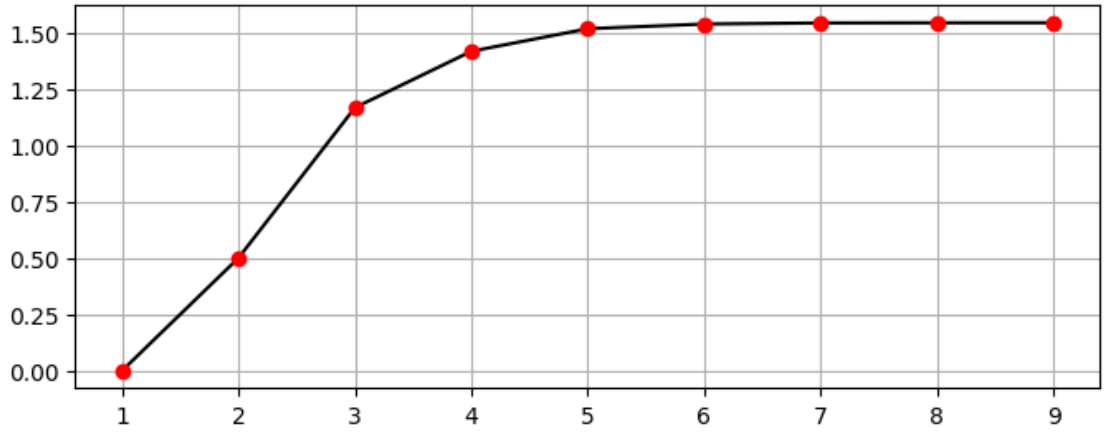
```
        s += 1 / math.factorial(2 * j)
    return math.factorial(n) * s - p
```

[38]:
```
n = 6
cmpt_ech.reset()
s = list(permutations(n))
print(cmpt_ech.val)
print(nombre_echanges(n))
```

```
1107
1107.0
```

Pas d'erreur, notre formule pour I_n donne bien la valeur exacte du nombre d'échanges. Voici le graphe de la complexité de `permutations(n)` divisé par $n!$ en fonction de n.

[39]:
```
cmpts = []
for k in range(1, 10):
    cmpt_ech.reset()
    s = list(permutations(k))
    cmpts.append(cmpt_ech.val / math.factorial(k))
plt.plot(range(1, 10), cmpts, 'k')
plt.plot(range(1, 10), cmpts, 'or')
plt.grid()
```



1.5.3 5.3 Nombre de comparaisons

Proposition. la complexité de `permutations` en nombre de comparaisons d'éléments de listes est équivalente, lorsque n tend vers l'infini, à

$$\left(\frac{3e}{2} - 1\right) n!$$

Démonstration. Notons C'_n le nombre de comparaisons d'éléments de liste effectués lors de l'appel `permutations(n)`, en excluant le calcul (qui échoue) du successeur de ω. On a $C'_1 = 0$ et, pour tout $n \geq 2$,

$$C'_n = nC'_{n-1} + \frac{1}{2}(n-1)(3n-2)$$

En effet :

- Pour $k = 0, \dots, n-1$, la fonction énumère les permutations s telles que $s[0] = k$, ce qui demande, pour chaque valeur de k, C_{n-1} comparaisons.
- Pour $k = 0, \dots, n-2$, la fonction calcule le successeur de la plus grande permutation s telle que $s[0] = k$. Cette permutation est $s = [k, n-1, n-2, \dots, 0]$ (où k n'apparaît pas dans les pointillés). Le pivot de s est $i = 1$. Passer de s à son successeur demande donc
  - $n-1$ comparaisons pour trouver le pivot, égal à 1.
  - $k+1$ comparaisons pour trouver l'indice j maximal tel que $s[j] > s[0]$. Au total, cela donne

$$n(n-1) + \sum_{k=0}^{n-2} k = n(n-1) + \frac{1}{2}(n-2)(n-1) = \frac{1}{2}(n-1)(3n-2)$$

Remarquons que le calcul du successeur de ω, quant à lui, nécessite $n-1$ comparaisons.

Avant de calculer C'_n en fonction de n, testons notre récurrence !

[40]:
```
def nombre_comparaisons0(n):
    if n == 1: return 0
    else:
        return n * nombre_comparaisons0(n - 1) + (n - 1) * (3 * n - 2) / 2
```

[41]:
```
n = 8
cmpt_comp.reset()
list(permutations(n))
print(cmpt_comp.val)
print(nombre_comparaisons0(n))
```

```
124076
124069.0
```

Un écart de 7 entre le nombre de comparaisons donné par la récurrence et le nombre de comparaisons réel. Normal, il manque les comparaisons effectuées lors du calcul du successeur (inexistant) de ω.

Posons maintenant

$$T_n = C'_n + \frac{1}{2}(3n+1)$$

On obtient facilement que $T_1 = 2$, et pour tout $n \geq 2$,

$$T_n = nT_{n-1} + 1$$

On montre alors par récurrence sur n que pour tout $n \geq 1$,

$$T_n = n! \left(\frac{3}{2} \sum_{k=0}^{n} \frac{1}{k!} - 1 \right)$$

et donc

$$C'_n = n! \left(\frac{3}{2} \sum_{k=0}^{n} \frac{1}{k!} - 1 \right) - \frac{1}{2}(3n + 1)$$

Il reste à ajouter le nombre de comparaisons effectuées par le calcul (qui échoue) du successeur de ω. Ces comparaisons se font lors de la recherche du pivot, il en a $n-1$. Pour ceux qui cherchent la petite bête, il y en a $n-1$ et pas n parce que, dans le code de la fonction `pivot`, le test `k>0` de la ligne `while k > 0 and inferieur(s, k, k - 1):` échoue lorsque $k = 0$ et, donc, le second test `inferieur(s, k, k - 1)` n'est pas exécuté dans ce cas.

Le nombre de comparaisons effectuées par `permutations`, que nous appellerons C_n, est donc

$$C_n = n! \left(\frac{3}{2} \sum_{k=0}^{n} \frac{1}{k!} - 1 \right) - \frac{1}{2}(n + 3)$$

La quantité entre parenthèses tend vers $\frac{3}{2}e - 1 \simeq 3.07742$ lorsque n tend vers l'infini. Ainsi,

$$C_n \sim \left(\frac{3e}{2} - 1 \right) n!$$

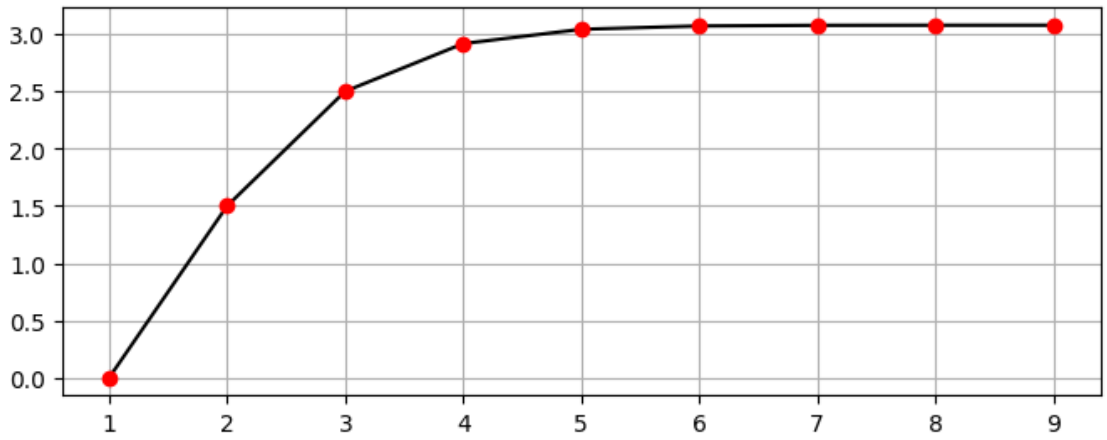
Illustrons tout cela avec un peu de code, comme nous l'avons fait pour le nombre d'échanges.

[42]:
```
def nombre_comparaisons(n):
    s = 0
    for k in range(n + 1):
        s = s + 1 / math.factorial(k)
    s = 3 / 2 * s - 1
    return math.factorial(n) * s - (n + 3) / 2
```

[43]:
```
n = 8
cmpt_comp.reset()
list(permutations(n))
print(cmpt_comp.val)
print(nombre_comparaisons(n))
```

```
124076
124076.00000000001
```

[44]:
```
cmpts = []
for k in range(1, 10):
    cmpt_comp.reset()
    s = list(permutations(k))
    cmpts.append(cmpt_comp.val / math.factorial(k))
plt.plot(range(1, 10), cmpts, 'k')
plt.plot(range(1, 10), cmpts, 'or')
plt.grid()
```



1.5.4 5.3 La complexité en moyenne de `successeur`

Les deux paragraphes précédents nous montrent que les $n!$ appels à la fonction `successeur` pour les permutations de $\mathfrak{S}_n$ nécessitent un nombre total d'opérations élémentaires (comparaisons et échanges) équivalent à

$$I_n + C_n \sim \left(\cosh 1 + \frac{3e}{2} - 1\right) n!$$

En munissant $\mathfrak{S}_n$ de la probabilité uniforme, la complexité moyenne de la fonction `successeur` est

$$\frac{I_n + C_n}{n!} \sim \mu$$

où

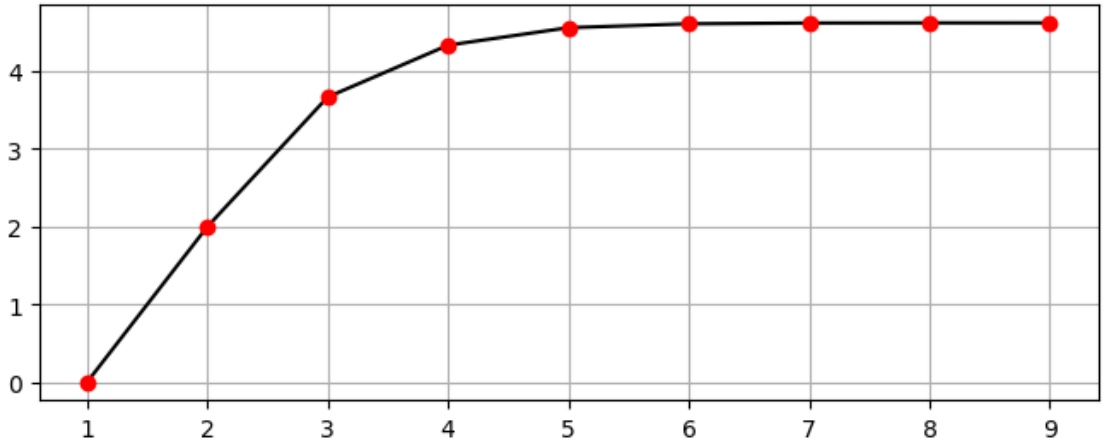
$$\mu = \left(\cosh 1 + \frac{3e}{2} - 1\right) = 2e + \frac{1}{2e} - 1 \simeq 4.6205$$

c'est à dire une complexité … en $\mathcal{O}(1)$! Ci-dessous, le graphe de cette complexité moyenne.

[45]:
```
print(2 * math.e + 1 / (2 * math.e) - 1 )
```

```
4.620503377503812
```

[46]:
```
cmpts = []
for k in range(1, 10):
    cmpt_comp.reset()
    cmpt_ech.reset()
    s = list(permutations(k))
    cmpts.append((cmpt_comp.val + cmpt_ech.val) / math.factorial(k))
plt.plot(range(1, 10), cmpts, 'k')
plt.plot(range(1, 10), cmpts, 'or')
plt.grid()
```



[47]:
```
print(cmpts)
```

```
[0.0, 2.0, 3.6666666666666665, 4.333333333333333, 4.558333333333334,
4.608333333333333, 4.618650793650794, 4.620238095238095, 4.620472332451499]
```

Pour terminer, faisons quelques statistiques, histoire de voir que notre belle théorie fonctionne parfaitement en pratique.

1.5.5 5.4 Statistiques

La fonction `random_perm` ci-dessous renvoie une permutation aléatoire des entiers entre 0 et $n-1$. Elle utilise *l'algorithme de Fisher-Yates*, que nous ne détaillerons pas ici.

[48]:
```
def random_perm(n):
    s = list(range(n))
    for i in range(n - 1, -1, -1):
        j = random.randint(i, n - 1)
        s[i], s[j] = s[j], s[i]
    return s
```

[49]:
```
random_perm(10)
```

[49]:
```
[1, 9, 0, 8, 5, 3, 2, 6, 7, 4]
```

Proposition. Pour toute permutation $\sigma \in \mathfrak{S}_n$, la fonction `random_perm` renvoie σ avec une probabilité $\frac{1}{n!}$.

Démonstration. admise.

La fonctions `test_succ` tire N permutations aléatoires et calcule leur successeur. Elle renvoie la moyenne du nombre d'opérations élémentaires effectuées.

[50]:
```
def test_succ(n, N=50000):
    count = 0
    for k in range(N):
        cmpt_comp.reset()
        cmpt_ech.reset()
        s = random_perm(n)
        s1 = successeur(s)
        count += cmpt_comp.val + cmpt_ech.val
    return count / N
```

[51]:
```
test_succ(10)
```

[51]:
```
4.60638
```

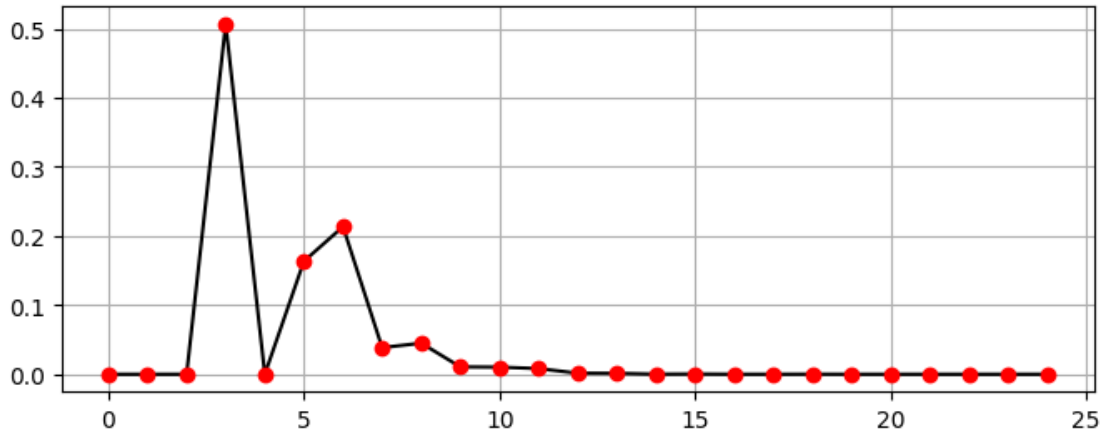
Rappelons que la théorie prévoit une moyenne d'environ 4.62. Eh bien la théorie fait une bonne prévision :-).

Quelle est la **loi** de la variable aléatoire « nombre d'opérations » ? Nous ne ferons pas de théorie ici, contentons nous d'un histogramme.

[52]:
```
def histo_succ(n, N=10000):
    p = (5 * n) // 2
    histo = p * [0]
    for k in range(N):
        cmpt_comp.reset()
        cmpt_ech.reset()
        s = random_perm(n)
        s1 = successeur(s)
        count = cmpt_comp.val + cmpt_ech.val
        histo[count] += 1
    for k in range(p):
        histo[k] = histo[k] / N
    return histo
```

[53]:
```
h = histo_succ(10)
plt.plot(h, 'k')
plt.plot(h, 'or')
```

```
plt.grid()
```



Une dernière petite remarque avant de clore ce notebook. Apparemment, la probabilité que le nombre d'opérations élémentaires soit égal à 3 est $\frac{1}{2}$. Comment cela se fait-il ?

Soit $n \geq 3$. Soit $s \in \mathfrak{S}_n$. Supposons que le pivot de s ne soit pas $n-1$. Soit s' le successeur de s. On a $s[n-2] > s[n-1]$ et donc (voir la fonction `successeur`), $s'[n-2] < s'[n-1]$. Ainsi, le pivot de s' est $n-1$, et donc la complexité de `successeur` pour le calcul de s'', successeur de s', est 3. De plus (exercice), le pivot de s'' n'est pas $n-1$. Comme le pivot de id est $1 \neq n-1$, exactement la moitié des permutations ont un pivot i égal à $n-1$, donc

$$\mathbb{P}(i = n-1) = \frac{1}{2}$$