Coef_Binom

October 31, 2020

1 Les coefficients binomiaux

Marc Lorenzi

27 octobre 2020

Pari : Calculer $\binom{2 \text{ millions}}{1 \text{ million}}$ en moins d'une minute.

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

[2]:
```
plt.rcParams['figure.figsize'] = (10, 6)
```

1.1 1. Introduction

1.1.1 1.1 Qu'allons nous faire ?

Nous allons dans ce notebook nous intéresser aux **coefficients binomiaux**. Pour tous entiers $n, k \in \mathbb{N}$, nous allons définir un nombre $\binom{n}{k} \in \mathbb{N}$. Nous allons étudier quelques propriétés bien connues de ces nombres, et décrire quelques algorithmes permettant de les calculer.

Dans la dernière section nous obtiendrons un algorithme très efficace permettant de factoriser les coefficients binomiaux. De cet algorithme de factorisation nous déduirons facilement un algorithme de calcul des coefficients binomiaux. Celui-ci nous permattra de calculer en un temps très raisonnable le coefficient $\binom{2000000}{1000000}$, qui est un nombre d'environ 600000 chiffres.

1.1.2 1.2 Compteurs

Définissons une classe `Compteur`. Un objet de cette classe est un *compteur*. On peut l'incrémenter ou le décrémenter (de 1 par défaut) par les méthodes `incr` et `decr`, ou lui donner une valeur précise (0 par défaut) par la méthode `reset`. On peut également obtenir la valeur du compteur par la méthode `val`.

[3]:
```
class Compteur:

```

```
    def __init__(self, val=0): self.val = val

    def incr(self, step=1): self.val = self.val + step
    def decr(self, step=1): self.val = self.val - step
    def reset(self, val=0): self.val = val
    def val(self): return self.val

    def __str__(self): return str(self.val)
```

Définissons un compteur global `CPTR`. Celui-ci nous servira dans quelques unes de nos expériences futures.

[4]:
```
CPTR = Compteur()
```

Faisons un petit test.

[5]:
```
CPTR.reset()
for k in range(10000):
    r = random.randint(0,1)
    if r == 0: CPTR.incr()
    else: CPTR.decr()
print(CPTR)
CPTR.reset()
print(CPTR)
```

```
58
0
```

1.2 2. Coefficients binomiaux

Nous allons dans ce qui suit parler de cardinaux. Voici ce qu'il y a à savoir :

- Un ensemble A est dit **fini** lorsqu'il existe $n \in \mathbb{N}$ tel que A soit en bijection avec $[|1, n|]$. Un tel entier n est alors unique. On l'appelle le cardinal de A et on le note $|A|$.
- Deux ensembles finis A et B ont le même cardinal si et seulement si il existe une bijection de A sur B.
- Une réunion d'ensembles finis **disjoints** est encore un ensemble fini, et son cardinal est la somme de leurs cardinaux.
- Des « évidences », comme par exemple le fait que le cardinal d'un sous-ensemble est plus petit que le cardinal de l'ensemble.

Tout ceci sera détaillé dans le cours de dénombrement.

1.2.1 2.1 Parties d'un ensemble

2.1.1 Parties quelconques **Notation.** Soit E un ensemble fini. On note $\mathcal{P}(E)$ l'ensemble des parties de E.

Proposition. Soit E un ensemble fini de cardinal n. On a

$$|\mathcal{P}(E)| = 2^n$$

Démonstration. on procède par récurrence sur n.

- Le seul ensemble de cardinal 0 est $\emptyset$, qui possède $2^0 = 1$ partie : lui-même.
- Soit $n \in \mathbb{N}$. Supposons que pour tout ensemble fini E de cardinal n, on a $|\mathcal{P}(E)| = 2^n$. Soit E un ensemble fini de cardinal $n+1$. Donnons-nous $a \in E$ et écrivons $E = E' \cup \{a\}$ où E' est de cardinal n. On a alors

$$\mathcal{P}(E) = \mathcal{P}'(E) \cup \mathcal{P}''(E)$$

où

$$\mathcal{P}'(E) = \{X \in \mathcal{P}(E), a \in X\}$$

$$\mathcal{P}''(E) = \{X \in \mathcal{P}(E), a \notin X\} = \mathcal{P}(E')$$

Clairement, l'application $X \mapsto X \cup \{a\}$ est une bijection de $\mathcal{P}(E')$ sur $\mathcal{P}'(E)$. Ainsi, $|\mathcal{P}(E')| = |\mathcal{P}'(E)|$. De là,

$$|\mathcal{P}(E)| = |\mathcal{P}'(E)| + |\mathcal{P}''(E)| = 2|\mathcal{P}(E')| = 2^{n+1}$$

en appliquant l'hypothèse de récurrence à E', qui est de cardinal n.

2.1.2 Parties de cardinal donné Notation. Soit E un ensemble fini. Pour tout $k \in \mathbb{N}$, on note $\mathcal{P}_k(E)$ l'ensemble des parties de E de cardinal k. On note également

$$\binom{n}{k} = |\mathcal{P}_k(E)|$$

Les entiers $\binom{n}{k}$ sont appelés les **coefficients binomiaux** (lire « k parmi n », c'est le « nombre de façons » de « choisir » k objets parmi n objets). Ils sont définis pour $n, k \in \mathbb{N}$.

2.1.3 Deux formules bien connues Proposition. Soit $n \in \mathbb{N}$. On a

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

Démonstration. Soit E un ensemble de cardinal n. On a

$$\mathcal{P}(E) = \bigcup_{k=0}^{n} \mathcal{P}_k(E)$$

et cette réunion est disjointe. Ainsi,

$$2^n = |\mathcal{P}(E)| = \sum_{k=0}^{n} |\mathcal{P}_k(E)| = \sum_{k=0}^{n} \binom{n}{k}$$

Proposition. Soit $n \in \mathbb{N}$. Soit $0 \leq k \leq n$. On a

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

Démonstration. Soit E un ensemble de cardinal n. L'application $A \mapsto E \setminus A$ est une bijection de $\mathcal{P}_k(E)$ sur $\mathcal{P}_{n-k}(E)$. Ces deux ensembles ont donc le même cardinal.

1.2.2 2.2 Une première fonction

Notre première fonction de calcul des coefficients binomiaux recherche toutes les parties à k éléments d'un ensemble à n éléments. Puis elle compte combien elle a trouvé de parties.

Comment créer toutes les parties à k éléments d'un ensemble E de cardinal n ?

- Si $k < 0$, il n'y a aucune telle partie.
- Si $k = 0$, il y a une seule partie, $\emptyset$.
- Si $k \geq 1$ et $n = 0$ il n'y a aucune partie.
- Si $k \geq 1$ et $n \geq 1$, prenons $a \in E$ et écrivons

$$E = E' \cup \{a\}$$

où E' est de cardinal $n - 1$. Posons

$$\mathcal{P}'_k(E) = \{X \in \mathcal{P}_k(E), a \in X\}$$

et

$$\mathcal{P}''_k(E) = \{X \in \mathcal{P}_k(E), a \notin X\}$$

Ces deux ensembles sont disjoints, et

$$\mathcal{P}_k(E) = \mathcal{P}'_k(E) \cup \mathcal{P}''_k(E)$$

La fonction `parties` prend en paramètres un ensemble E représenté par une liste Python et un entier $k \in \mathbb{Z}$. Elle renvoie la liste des parties de E de cardinal k. Dans le cas intéressant $(n, k \geq 1)$, cette fonction utilise ce que nous venons de dire, en prenant $a = E[0]$ et $E' = E \setminus \{a\} = E[1:]$.

Remarque. Nous comptons, grâce au compteur `CPTR` le nombre de concaténations de listes effectuées par la fonction. Ainsi, à l'avant dernière ligne, `CPTR` est incrémenté de 2 car à la ligne suivante on effectue deux concaténations de listes (deux signes +).

[6]:
```
def parties(E, k):
    if k == 0: return [[]]
    else:
        n = len(E)
        if n == 0: return []
        else:
            a = E[0]
            E1 = E[1:]
            P1 = parties(E1, k - 1)
            P2 = parties(E1, k)
            CPTR.incr(2)
            return P2 + [[a] + X for X in P1]
```

[7]:
```
print(parties([1, 2, 3, 4, 5], 3))
```

```
[[3, 4, 5], [2, 4, 5], [2, 3, 5], [2, 3, 4], [1, 4, 5], [1, 3, 5], [1, 3, 4],
[1, 2, 5], [1, 2, 4], [1, 2, 3]]
```

Voici notre première fonction de calcul des coefficients binomiaux.

[8]:
```
def binomial0(n, k):
    return len(parties(list(range(n)), k))
```

Testons, en affichant le **triangle de Pascal** (la fonction `fmt` nous servira pour afficher de façon jolie des listes d'entiers. Nous verrons plus loin comment nous en servir).

[9]:
```
def fmt(n, d=6):
    return n * ('%' + str(d) + 'd')
```

[10]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 4) % tuple([binomial0(n, k) for k in range(n + 1)]))
```

```
   1
   1   1
   1   2   1
   1   3   3   1
   1   4   6   4   1
   1   5  10  10   5   1
   1   6  15  20  15   6   1
   1   7  21  35  35  21   7   1
```

```
  1   8  28  56  70  56  28   8   1
  1   9  36  84 126 126  84  36   9   1
  1  10  45 120 210 252 210 120  45  10   1
```

Bien entendu, cette fonction est terriblement inefficace, en temps comme en espace.

[11]:
```
CPTR.reset()
print(binomial0(20, 10))
print('CPTR =', CPTR)
```

```
184756
CPTR = 1233330
```

Le calcul de $\binom{20}{10}$ nécessite environ 1 million de concaténations de listes ! De plus, la fonction crée 184756 listes d'entiers de longueur 10, ce qui nécessite le stockage de 1847560 entiers.

Il va nous falloir faire mieux que cela. Première idée, nous n'avons pas besoin de **chercher** les parties d'un ensemble, mais seulement de les **compter**. Mais avant, faisons une petite digression.

1.2.3 2.3 La formule du binôme

Si les coefficients binomiaux s'appellent ainsi, c'est parce qu'ils apparaissent dans la **formule du binôme**. On se propose de montrer cette formule uniquement avec ce qui précède dans le notebook … c'est à dire la définition des coefficients binomiaux.

Donnons-nous $2n$ réels (ou complexes, ou autres $(*)$) $a_1, \dots, a_n, b_1, \dots, b_n$ et développons le produit

$$P = (a_1 + b_1)(a_2 + b_2) \dots (a_n + b_n) = F_1 F_2 \dots F_n$$

P est un produit de sommes. Développer P, c'est écrire P comme une somme de produits. Chacun des termes de la somme développée est un produit de n facteurs. Ces facteurs sont obtenus en choisissant, pour chacun des facteurs F_i de P, l'un des termes a_i ou b_i. On fait cela pour tous les choix possibles. En gros,

$$P = \sum_{\text{tous les choix}} x_1 \dots x_n$$

où x_i vaut a_i ou b_i. Comment préciser ces choix ? Il suffit de dire quels sont les facteurs de P dans lesquels on choisit a_i (pour les autres, on choisit b_i). Un tel choix de facteurs est caractérisé par la donnée d'une partie A de l'ensemble $E = [|1, n|]$: A est l'ensemble des numéros i des facteurs F_i pour lesquels on choisit a_i. On somme ensuite pour toutes les parties A possibles. On a ainsi

$$P = \sum_{A \in \mathcal{P}(E)} \prod_{i \in A} a_i \prod_{i \notin A} b_i$$

Prenons maintenant tous les a_i égaux à un réel a, et tous les b_i égaux à un réel b. On a

$$P = \sum_{A\in\mathcal{P}(E)} \prod_{i\in A} a \prod_{i\notin A} b = \sum_{A\in\mathcal{P}(E)} a^{|A|}b^{|E\setminus A|} = \sum_{A\in\mathcal{P}(E)} a^{|A|}b^{n-|A|}$$

Réordonnons cette somme en regroupant les ensembles A de même cardinal k, puis en sommant sur toutes les valeurs de k (c'est à dire $0, 1, \ldots, n$). Il vient

$$\begin{array}{rcl} P & = & \sum_{k=0}^{n} \sum_{A\in\mathcal{P}_k(E)} a^{|A|}b^{n-|A|} \\ & = & \sum_{k=0}^{n} \sum_{A\in\mathcal{P}_k(E)} a^k b^{n-k} \\ & = & \sum_{k=0}^{n} \left(\sum_{A\in\mathcal{P}_k(E)} 1\right) a^k b^{n-k} \end{array}$$

Mais $\sum_{A\in\mathcal{P}_k(E)} 1 = |\mathcal{P}_k(E)| = \binom{n}{k}$. Ainsi,

$$P = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

Nous venons de montrer la formule du binôme.

Théorème. Soient $a, b \in \mathbb{R}$. Soit $n \in \mathbb{N}$. On a

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

$(*)$ *De façon très générale, la formule du binôme est vraie lorsque* $a, b \in \mathbb{A}$ *où* $\mathbb{A}$ *est un* ***anneau****, et que de plus* $ab = ba$.

1.2.4 2.4 Une relation de récurrence

Proposition. Pour tous $n, k \in \mathbb{N}^*$,

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

Démonstration. Reprenons les notations vues dans le paragraphe 2.2. Prenons $a \in E$ et écrivons

$$E = E' \cup \{a\}$$

où E' est de cardinal $n-1$. Posons

$$\mathcal{P}'_k(E) = \{X \in \mathcal{P}_k(E), a \in X\}$$

et

$$\mathcal{P}''_k(E) = \{X \in \mathcal{P}_k(E), a \notin X\}$$

Ces deux ensembles sont disjoints, et

$$\mathcal{P}_k(E) = \mathcal{P}'_k(E) \cup \mathcal{P}''_k(E)$$

Faisons les remarques suivantes :

- $\mathcal{P}''_k(E) = \mathcal{P}_k(E')$, donc $|\mathcal{P}''_k(E)| = |\mathcal{P}_k(E')| = \binom{n-1}{k}$.
- $\mathcal{P}'_k(E) = \{X \cup \{a\}, X \in \mathcal{P}_{k-1}(E')\}$. L'application $X \mapsto X \cup \{a\}$ est une bijection de $\mathcal{P}_{k-1}(E')$ sur $\mathcal{P}'_k(E)$, donc $|\mathcal{P}'_k(E)| = |\mathcal{P}_{k-1}(E')| = \binom{n-1}{k-1}$.

De là,

$$\binom{n}{k} = |\mathcal{P}_k(E)| = |\mathcal{P}'_k(E)| + |\mathcal{P}''_k(E)| = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Cette relation permet d'écrire une fonction qui calcule les coefficients binomiaux. Appelons-la `binomial1`.

[12]:
```
def binomial1(n, k):
    if k == 0: return 1
    elif n == 0: return 0
    else:
        CPTR.incr()
        return binomial1(n - 1, k - 1) + binomial1(n - 1, k)
```

[13]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 4) % tuple([binomial1(n, k) for k in range(n + 1)]))
```

```
   1
   1   1
   1   2   1
   1   3   3   1
   1   4   6   4   1
   1   5  10  10   5   1
   1   6  15  20  15   6   1
   1   7  21  35  35  21   7   1
   1   8  28  56  70  56  28   8   1
   1   9  36  84 126 126  84  36   9   1
   1  10  45 120 210 252 210 120  45  10   1
```

Sans surprise, on retrouve la même table que celle que nous avions obtenue avec `binomial0`.

1.2.5 2.5 Complexité de la fonction `binomial1`

Pour tout entiers naturels n et k, notons $A_{n,k}$ le nombre d'additions effectuées par `binomial1` lors du calcul de $\binom{n}{k}$. On a

- Pour tout $n \geq 0$, $A_{n,0} = 0$.

- Pour tout $k > 0$, $A_{0,k} = 0$.
- Pour tout $n \geq 1$ et tout $k \geq 1$, $A_{n,k} = A_{n-1,k-1} + A_{n-1,k} + 1$

[14]:
```
def A(n, k):
    if k == 0: return 0
    elif n == 0: return 0
    else: return A(n - 1, k - 1) + A(n - 1, k) + 1
```

Voici les premières valeurs de $A_{n,k}$.

[15]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 5) % tuple([A(n, k) for k in range(n + 1)]))
```

```
    0
    0    1
    0    2    3
    0    3    6    7
    0    4   10   14   15
    0    5   15   25   30   31
    0    6   21   41   56   62   63
    0    7   28   63   98  119  126  127
    0    8   36   92  162  218  246  254  255
    0    9   45  129  255  381  465  501  510  511
    0   10   55  175  385  637  847  967 1012 1022 1023
```

Histoire de nous rassurer, contrôlons par exemple le nombre d'additions effectuées pour le calcul de $\binom{10}{7}$. Le tableau nous dit 967. Et dans la réalité ?

[16]:
```
CPTR.reset()
x = binomial1(10, 7)
print(CPTR)
```

```
967
```

Nous voici rassurés.

Peut-on obtenir une formule explicite pour $A_{n,k}$? Oui, plus ou moins. Posons $B_{n,k} = A_{n,k} + 1$. On a alors

- Pour tout $n \geq 0$, $B_{n,0} = 1$.
- Pour tout $k > 0$, $B_{0,k} = 1$.
- Pour tout $n \geq 1$ et tout $k \geq 1$, $B_{n,k} = B_{n-1,k-1} + B_{n-1,k}$

[17]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 5) % tuple([A(n, k) + 1 for k in range(n + 1)]))
```

```
    1
    1    2
```

```
1     3     4
1     4     7     8
1     5    11    15    16
1     6    16    26    31    32
1     7    22    42    57    63    64
1     8    29    64    99   120   127   128
1     9    37    93   163   219   247   255   256
1    10    46   130   256   382   466   502   511   512
1    11    56   176   386   638   848   968  1013  1023  1024
```

Proposition. On a pour tous entiers $n, k \geq 0$

$$B_{n,k} = \sum_{j=0}^{k} \binom{n}{j}$$

Démonstration. On fait une récurrence sur n. C'est clair pour $n = 0$ puisque dans ce cas $B_{n,k} = 1$. Soit $n \in \mathbb{N}^*$. Supposons que pour tout $k \in \mathbb{N}$, $B_{n-1,k} = \sum_{j=0}^{k} \binom{n-1}{j}$. Montrons que ceci est encore vrai pour n.

Pour $k = 0$, la propriété est vraie puisque $B_{n,0} = 1$. Soit $k \in \mathbb{N}^*$. On a

$$\begin{aligned} B_{n,k} &= B_{n-1,k-1} + B_{n-1,k} \\ &= \sum_{j=0}^{k-1} \binom{n-1}{j} + \sum_{j=0}^{k} \binom{n-1}{j} \\ &= \sum_{j=1}^{k} \binom{n-1}{j-1} + \sum_{j=0}^{k} \binom{n-1}{j} \\ &= \binom{n-1}{0} + \sum_{j=1}^{k} \left(\binom{n-1}{j-1} + \binom{n-1}{j} \right) \\ &= 1 + \sum_{j=1}^{k} \binom{n}{j} \\ &= \sum_{j=0}^{k} \binom{n}{j} \end{aligned}$$

Corollaire. On a pour tous entiers $n, k \geq 0$

$$A_{n,k} = \sum_{j=1}^{k} \binom{n}{j}$$

La valeur obtenue pour $A_{n,k}$ n'est pas du tout rassurante. Par exemple, dans le cas où $k = n$, on obtient $A_{n,n} = 2^n - 1$. Il faut $2^n - 1$ additions pour calculer $\binom{n}{n}$ avec notre fonction `binomial1`.

[18]:
```
CPTR.reset()
x = binomial1(20, 20)
print(CPTR)
print(2 ** 20 - 1)
```

```
1048575
1048575
```

Inutile de dire que ce n'est pas avec une telle fonction que nous calculerons des coefficients binomiaux où $n = 2000000$.

1.3 3. La taille des coefficients binomiaux

Avant de nous plonger dans de nouveaux algorithmes, posons nous quelques questions sur les coefficients binomiaux. Pour un entier n fixé, quel est le comportement de la suite $\left(\binom{n}{k}\right)_{0\leq k\leq n}$? Quelle est la valeur maximale de cette suite ? Peut-on obtenir un équivalent simple de ce maximum ?

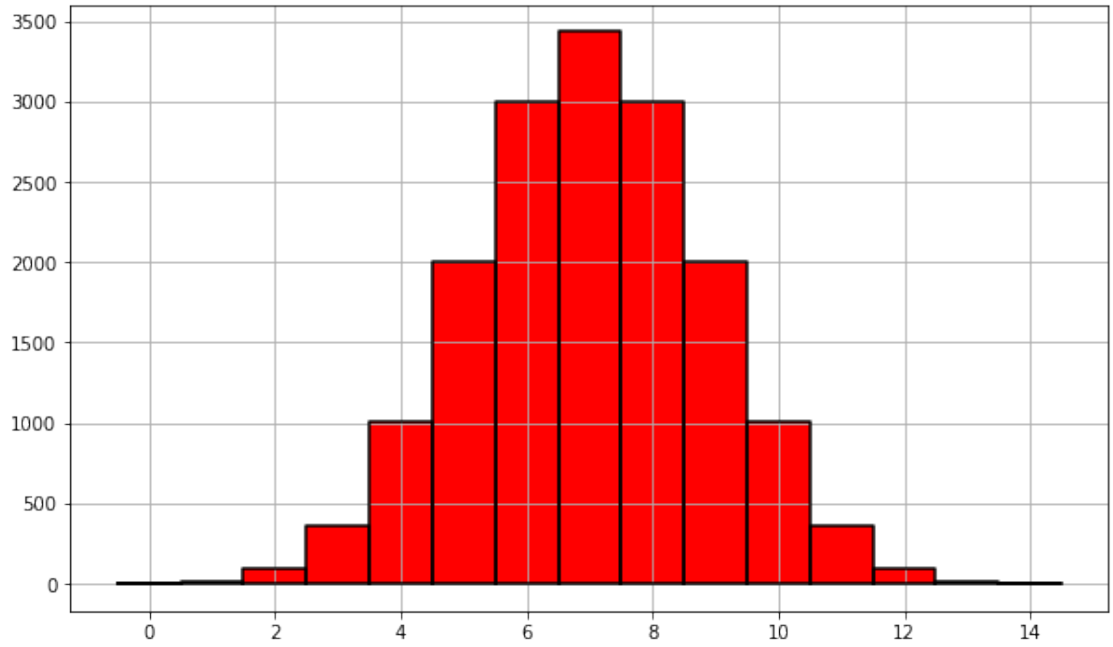
1.3.1 3.1 Représentation graphique

La fonction `plot_binom` prend en paramètre un entier n. Elle trace le graphe des coefficients binomiaux $\binom{n}{k}$ pour $0 \leq k \leq n$.

[19]:
```
def plot_binom(n):
    xs = list(range(n + 1))
    ys = [binomial1(n, k) for k in range(n + 1)]
    zs = [1]
    for k in range(n + 1):
        plt.fill([k - 0.5, k + 0.5, k + 0.5, k - 0.5, k - 0.5], [0, 0, ys[k],␣
↪ys[k], 0], 'r')
        plt.plot([k - 0.5, k + 0.5, k + 0.5, k - 0.5, k - 0.5], [0, 0, ys[k],␣
↪ys[k], 0], 'k')
    plt.grid()
```

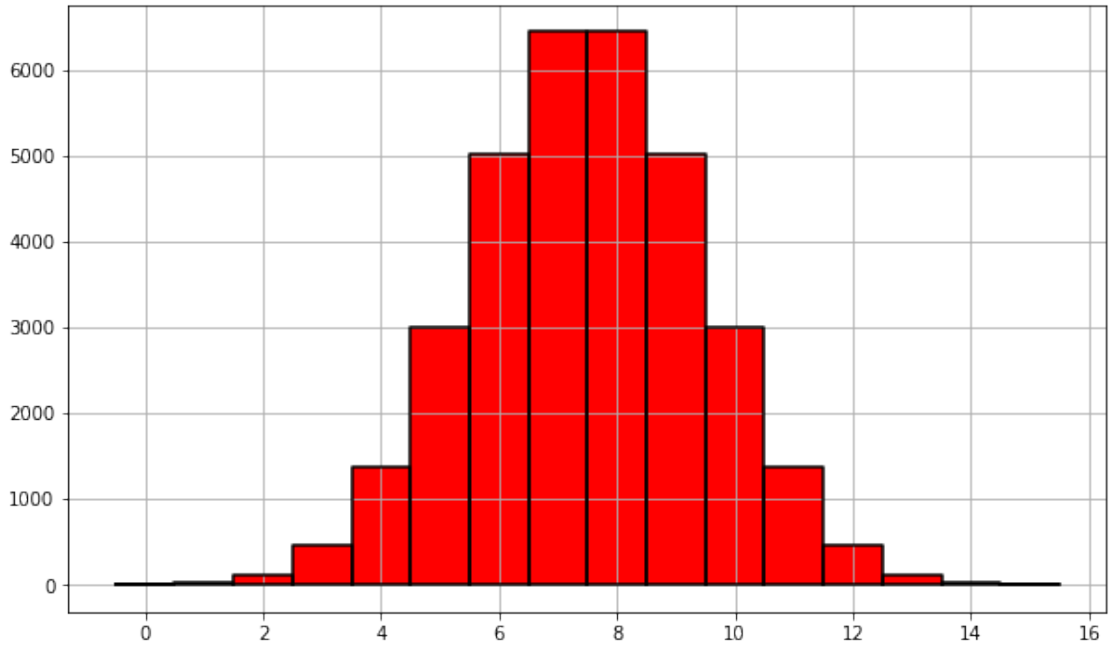
Traçons pour $n = 14$.

[20]:
```
plot_binom(14)
```



Et aussi pour $n = 15$.

[21]:
```
plot_binom(15)
```



Ça monte puis ça descend. Et le maximum est au milieu, avec une légère différence selon que n est pair (max en un unique point) ou impair (max en deux points).

1.3.2 3.2 Confirmations

Soient $n, k \in \mathbb{N}$, $n \geq 1$, $0 \leq k \leq n-1$. On a

$$\begin{array}{rcl} \binom{n}{k+1} - \binom{n}{k} & = & \frac{n!}{(k+1)!(n-1-k)!} - \frac{n!}{k!(n-k)!} \\ & = & \frac{n!}{(k+1)!(n-k)!}\,(n-2k-1) \end{array}$$

Cette quantité est du signe de $n-2k-1$. Discutons selon la parité de n.

- Cas 1, $n = 2p$ où $p \geq 1$. On a $n-2k-1 > 0$ si et seulement si $k < p - \frac{1}{2}$ c'est à dire, puisque k est entier, $k \leq p-1 = \frac{n}{2} - 1$. Ainsi, la suite $\left(\binom{n}{k}\right)_{0 \leq k \leq n}$ croît strictement pour $0 \leq k \leq p$, passe par un maximum pour $k = p$, puis décroît strictement.
- Cas 2, $n = 2p+1$ où $p \geq 0$. On a $n-2k-1 > 0$ si et seulement si $k < p$. Ainsi, la suite $\left(\binom{n}{k}\right)_{0 \leq k \leq n}$ croît strictement pour $0 \leq k \leq p$, prend deux valeurs égales pour $k = p$ et $k = p+1$, puis décroît strictement.

1.3.3 3.3 La valeur du maximum

Que « vaut » le coefficient maximal ? Regardons le cas où $n = 2p$ est pair (le cas n impair est similaire). Le coefficient maximal est obtenu pour $k = p$, et vaut $\binom{2p}{p}$. Calculons un équivalent de ce coefficient lorsque p tend vers l'infini, en utilisant la formule de Stirling. Cette formule nous dit que

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

On a donc

$$(2p)! \sim \left(\frac{2p}{e}\right)^{2p} \sqrt{2\pi 2p}$$

et

$$p!^2 \sim \left(\frac{p}{e}\right)^{2p} 2\pi p$$

Ainsi, après simplifications,

$$\binom{2p}{p} = \frac{(2p)!}{p!^2} \sim \frac{2^{2p}}{\sqrt{\pi p}}$$

[22]:
```
def equiv_max(n):
    p = n / 2
    return 2 ** n / math.sqrt(math.pi * n / 2)
```

[23]:
```
print(equiv_max(100))
```

```
1.0114388424145895e+29
```

Les coefficients binomiaux peuvent donc être énormes.

1.4 4. Une méthode un peu moins naïve

1.4.1 4.1 Une formule pour les coefficients binomiaux

Proposition. Pour tous entiers $n, k \in \mathbb{N}$ tels que $0 \leq k \leq n$, on a

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Démonstration. On procède par récurrence sur n.

- $n = 0$ impose $k = 0$ et la formule est clairement vraie dans ce cas, puisque $\binom{0}{0} = 1$ et $0! = 1$.

- Soit $n \geq 1$. Supposons la propriété vraie pour l'entier $n-1$. Soit $0 \leq k \leq n$. Supposons tout d'abord $k \neq 0$ et $k \neq n$. On a alors

$$\begin{array}{rcl} \binom{n}{k} & = & \binom{n-1}{k-1} + \binom{n-1}{k} \\ & = & \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-1-k)!} \\ & = & \frac{(n-1)!}{k!(n-k)!}\,(k + (n-k)) \\ & = & \frac{n!}{k!(n-k)!} \end{array}$$

Si $k = 0$, ou $k = n$, la formule est clairement vraie.

1.4.2 4.2 Fonction Python

Posons, pour $n \geq 0$ et $k \geq 1$,

$$n^{\underline{k}} = n(n-1)\dots(n-k+1)$$

Posons également $n^{\underline{0}} = 1$. Nous appellerons $n^{\underline{k}}$ la $k$ième **puissance descendante** de n. On a pour tous entiers $n, k \geq 0$ tels que $0 \leq k \leq n$,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n^{\underline{k}}}{k!} = \frac{n^{\underline{k}}}{k^{\underline{k}}}$$

Le calcul des coefficients binomiaux se ramène donc à celui des puissances descendantes.

[24]:
```
def power_down(n, k):
    p = 1
    for j in range(k): p = p * (n - j)
    return p
```

[25]:
```
def binomial2(n, k):
    if k < 0 or k > n: return 0
    else:
        return power_down(n, k) // power_down(k, k)
```

[26]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 4) % tuple([binomial2(n, k) for k in range(n + 1)]))
```

```
   1
   1   1
   1   2   1
   1   3   3   1
   1   4   6   4   1
   1   5  10  10   5   1
   1   6  15  20  15   6   1
   1   7  21  35  35  21   7   1
```

```
   1   8  28  56  70  56  28   8   1
   1   9  36  84 126 126  84  36   9   1
   1  10  45 120 210 252 210 120  45  10   1
```

Le calcul de `power_down(n, k)` demande k multiplications. Celui de $\binom{n}{k}$ par cette méthode demande donc $2k$ multiplications et une division. Évidemment, ces multiplications peuvent être très coûteuses, car les factorielles deviennent énormes lorsque n augmente.

Notre fonction `binomial2` est nettement plus efficace que `binomial1`. Nous pouvons maintenant calculer des coefficients binomiaux de taille plus importante.

Rappelez-vous notre estimation de $\binom{100}{50}$, qui était 1.01×10^{29}. Quelle est la valeur exacte de ce coefficient ?

[27]:
```
x = binomial2(100, 50)
print(x)
print(float(x))
```

```
100891344545564193334812497256
1.008913445455642e+29
```

[28]:
```
binomial2(2000, 1000)
```

[28]:
```
204815162698948971433516250298082504439642488798139703382038263767174818620208375582893299418261020620146476631999802369241548179800452479201804754976926157856301289663432064714851152395251651227768588611539546256147907378668464154444533617613770073855673814589630071306510455959514479888746206368718514551828551173166276253663773084682932255389049743859481431755030783796444370810085163724827462791417016619883764840843541430817785947037746565188475514680749694674923803033101818723298009668567458560252549910118113525353465888794196665367490451130611009631190627034250229315591110897673396399114912 0
```

Regardons le temps mis par `binomial2` pour calculer de « grands » coefficients binomiaux. Pour la seconde cellule ci-dessous un peu de patience est nécessaire.

[29]:
```
t1 = time.time()
x = binomial2(100000, 50000)
t2 = time.time()
print(t2 - t1)
```

```
1.9948289394378662
```

[30]:
```
t1 = time.time()
x = binomial2(200000, 100000)
t2 = time.time()
print(t2 - t1)
```

```
9.004955053329468
```

J'ai tenté `binomial2(2000000, 1000000)` et arrêté le calcul après une dizaine de minutes. Voyez un peu plus loin pour une estimation du temps nécessaire à ce calcul avec la fonction `binomial2`.

1.4.3 2.3 Temps réel de calcul

Le calcul de $\binom{2n}{n}$ nécessite donc $4n$ multiplications et une division. Pour $n = 10^6$, il faut donc effectuer 4 millions de multiplications (et une division). Mais nous faisons des multiplications de **grands** nombres !

Un algorithme naïf pour multiplier un nombre de m chiffres par un nombre de n chiffres demande (sans entrer dans les détails) de l'ordre de $m \times n$ opérations. Voir à ce sujet votre cours de CM2.

Quel est le nombre de chiffres en base 2 de l'entier n ? Soit K ce nombre de chiffres. On a

$$2^{K-1} \leq n < 2^K$$

Passant au logarithme en base 2, et ajoutant 1, il vient

$$K \leq \log_2 n + 1 < K + 1$$

Ainsi,

$$K = \lfloor \log_2(n) \rfloor + 1$$

La fonction `nb_chiffres` renvoie le nombre de chiffres de l'écriture de n en base 2. Elle renvoie 0 si n est nul.

[31]:
```
def nb_chiffres(n):
    if n == 0: return 0
    else:
        return math.floor(math.log(n, 2)) + 1
```

Réécrivons `power_down` en incrémentant `CPTR` de la valeur adéquate à chaque multiplication.

[32]:
```
def power_down2(n, k):
    p = 1
    for j in range(k):
        CPTR.incr(nb_chiffres(p) * nb_chiffres(n - j))
        p = p * (n - j)
    return p
```

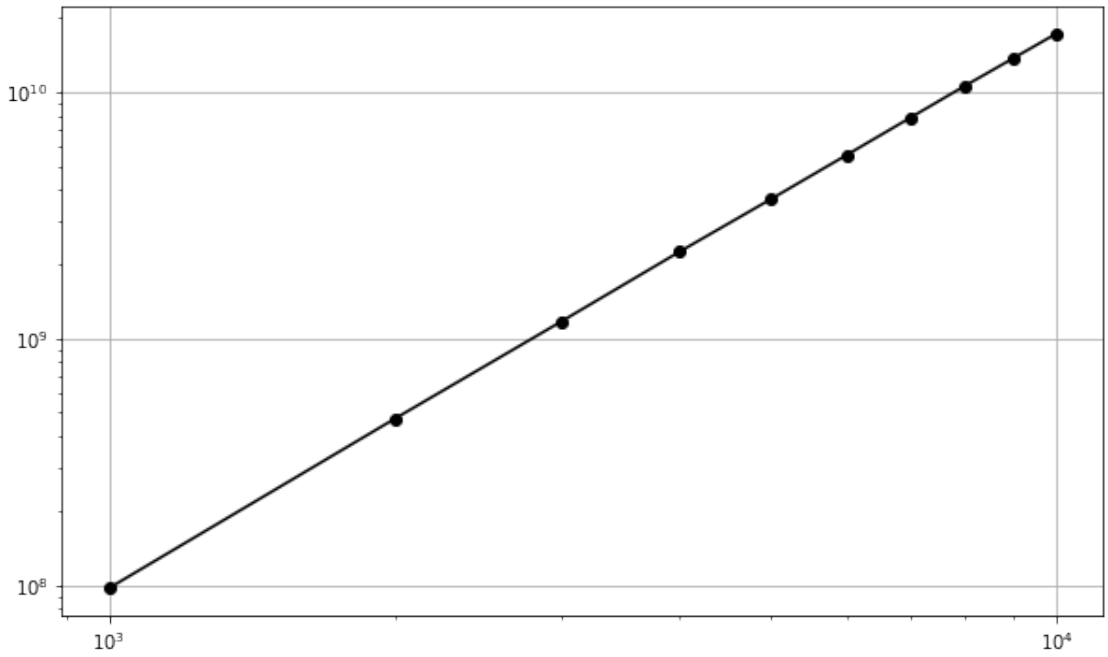
[33]:
```
def binomial3(n, k):
    if k < 0 or k > n: return 0
    else:
        CPTR.incr() # pour la division
        return power_down2(n, k) // power_down2(k, k)
```

[34]:
```
def test_binomial3(N):
    s = []
    ns = list(range(N, 11 * N, N))
    for n in ns:
```

```
        CPTR.reset()
        x = binomial3(2 * n, n)
        s.append(CPTR.val)
    plt.loglog(ns, s, '-ok')
    plt.grid()
```

Le tracé ci-dessous est en coordonnées logarithmiques.

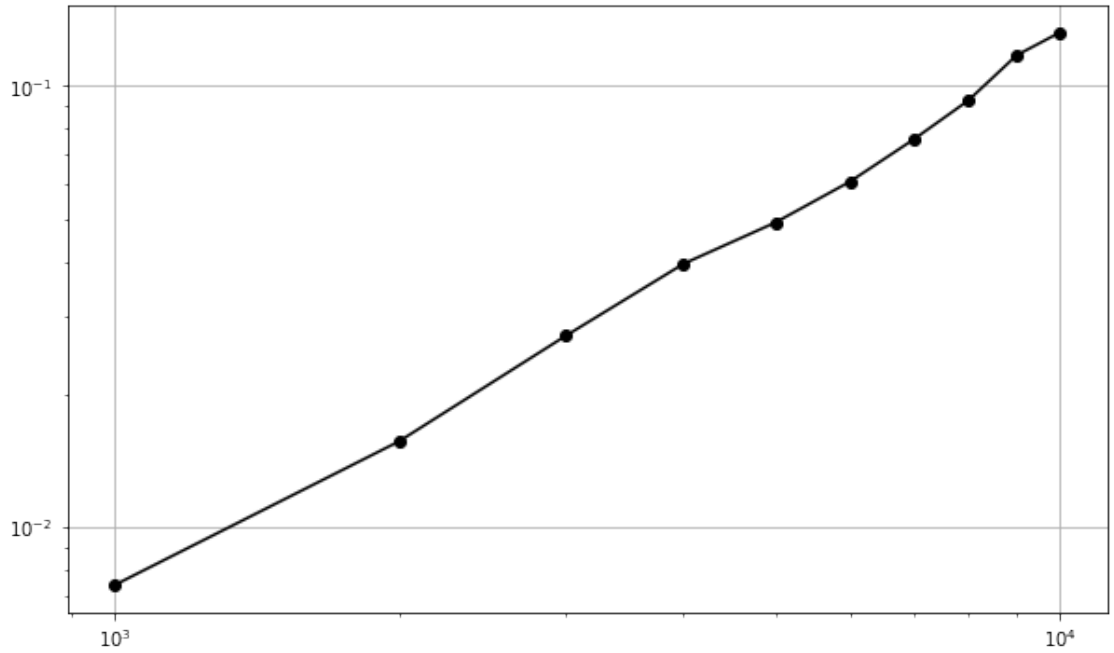
[35]:
```
test_binomial3(1000)
```



Plus réalistement, nous pouvons tracer la courbe des « vrais » temps d'exécution. Voici ce que l'on obtient (cela dépend évidemment de la machine utilisée).

[36]:
```
def test_binomial3bis(N):
    s = []
    ns = list(range(N, 11 * N, N))
    for n in ns:
        t1 = time.time()
        x = binomial3(2 * n, n)
        t2 = time.time()
        s.append(t2 - t1)
    plt.loglog(ns, s, '-ok')
    plt.grid()
```

[37]:
```
test_binomial3bis(1000)
```



On constate que la courbe est approximativement une droite. Si nous appelons T_n le temps mis pour calculer $\binom{2n}{n}$ on a donc à peu près

$$\ln T_n = c + \alpha \log n$$

où c et α sont deux constantes. α est la pente de la droite.

[38]:
```
t1 = time.time()
na = 50000
x = binomial2(2 * na, na)
t2 = time.time()
ta = t2 - t1

t1 = time.time()
nb = 100000
x = binomial2(2 * nb, nb)
t2 = time.time()

tb = t2 - t1

alpha = (math.log(tb) - math.log(ta)) / (math.log(nb) - math.log(na))
print(alpha)
```

```
2.2420261795687937
```

Obtenir la constante c est maintenant facile.

[39]:
```
c = math.log(tb) - alpha * math.log(nb)
print(c)
```

```
-23.549184020297048
```

Maintenant, $T_n = e^c n^\alpha$.

[40]:
```
def temps_binomial2(n):
    k = math.exp(c)
    return k * n ** alpha
```

Vérifions la pertinence de tout cela avec des valeurs de n pour lesquelles nous avons déjà mesuré le temps de calcul.

[41]:
```
temps_binomial2(100000)
```

[41]:
```
9.612807035446147
```

[42]:
```
temps_binomial2(150000)
```

[42]:
```
23.858956057190834
```

Nous retrouvons à peu près les temps que nous avions mesuré. Notre modèle n'est donc pas trop mauvais. Et si $n = 10^6$?

[43]:
```
temps_binomial2(1000000) / 60
```

[43]:
```
27.972105250560052
```

Il faudrait laisser tourner notre machine un peu plus de 20 minutes. Je vous laisse tenter l'expérience ...

1.5 5. Factorisation des coefficients binomiaux

1.5.1 5.1 Valuation p-adique d'un entier

Notons $\mathcal{P}$ l'ensemble des nombres premiers. Pour tout $n \in \mathbb{N}^*$ et tout nombre premier p, posons

$$\nu_p(n) = \max\{k \in \mathbb{N}, p^k \mid n\}$$

$\nu_p(n)$ est appelé la **valuation p-adique** de n. C'est la plus grande puissance de p que l'on puisse factoriser dans l'entier n. Tout entier naturel $n \geq 1$ peut s'écrire sous la forme

$$n = \prod_{p \in \mathcal{P}} p^{\nu_p(n)}$$

Le produit ci-dessus est en réalité fini, puisque si p est assez grand, p ne divise pas n et donc $\nu_p(n) = 0$.

On montre facilement les propriétés suivantes :

- Pour tous entiers $m, n \geq 1$, $\nu_p(mn) = \nu_p(m) + \nu_p(n)$.
- Pour tous entiers $m, n \geq 1$ tels que m divise n, $\nu_p(\frac{n}{m}) = \nu_p(n) - \nu_p(m)$.

1.5.2 5.2 Valuation p-adique des coefficients binomiaux

La clé de ce qui va suivre est la proposition suivante :

Proposition. Soient $n \in \mathbb{N}$ et p un nombre premier. On a

$$\nu_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$$

Démonstration. Posons $E = [|1, n|]$. On a $E = \bigcup_{k\geq 0} E_k$ où $E_k = \{x \in E, \nu_p(x) = k\}$. On a alors

$$\begin{array}{rcl} \nu_p(n!) & = & \sum_{x\in E} \nu_p(x) \\ & = & \sum_{k\geq 0} \sum_{x\in E_k} \nu_p(x) \\ & = & \sum_{k\geq 0} k|E_k| \end{array}$$

Considérons maintenant, pour tout $k \geq 0$, l'ensemble $F_k = \{x \in E, p^k | x\}$. On a

- $F_{k+1} \subset F_k$
- $E_k = F_k \setminus F_{k+1}$.

Ainsi $|E_k| = |F_k| - |F_{k+1}|$. De là,

$$\begin{array}{rcl} \nu_p(n!) & = & \sum_{k\geq 0} k(|F_k| - |F_{k+1}|) \\ & = & \sum_{k\geq 0} k|F_k| - \sum_{k\geq 0} k|F_{k+1}| \\ & = & \sum_{k\geq 0} k|F_k| - \sum_{k\geq 1} (k-1)|F_k| \\ & = & 0|F_0| + \sum_{k\geq 1} (k - (k-1))|F_k| \\ & = & \sum_{k\geq 1} |F_k| \end{array}$$

Pour terminer on remarque que, pour tout $k \geq 1$, $F_k = \{p^k, 2p^k, \ldots, \alpha p^k\}$ où $\alpha p^k \leq n < (\alpha+1)p^k$. Le cardinal de F_k est donc égal à $\alpha = \left\lfloor \frac{n}{p^k} \right\rfloor$.

La fonction `val_fact` ci-dessous renvoie la valuation p-adique de $n!$. Elle se contente d'appliquer la formule.

[44]:
```
def val_fact(n, p):
    s = 0
    while n != 0:
        n = n // p
        s = s + n
    return s
```

Quelle est la valuation dyadique de 1000! ?

[45]:
```
val_fact(1000, 2)
```

[45]: 994

1000! est divisible par 2^{994}, mais pas par 2^{995}.

Un corollaire immédiat du théorème précédent nous donne la valuation des coefficients binomiaux.

Corollaire. Soient $n, k \in \mathbb{N}$ et p un nombre premier. On a

$$\nu_p\left(\binom{n}{k}\right) = \sum_{i=1}^{\infty}\left(\left\lfloor\frac{n}{p^i}\right\rfloor - \left\lfloor\frac{k}{p^i}\right\rfloor - \left\lfloor\frac{n-k}{p^i}\right\rfloor\right)$$

Nous allons voir que l'on peut déduire de cette formule des relations de récurrence permettant de calculer la valuation p-adique de $\binom{n}{k}$ récursivement.

1.5.3 5.3 Une relation de récurrence

Notation. Pour tout entier n et tout nombre premier p, notons $\widehat{n} = \left\lfloor\frac{n}{p}\right\rfloor$. L'entier $\widehat{n}$ est le quotient de la division euclidienne de n par p.

Soit $n \in \mathbb{N}$. Soit p un nombre premier. Posons $n = p\widehat{n} + c_n$ où $0 \leq c_n < p$. On a pour tout $i \geq 1$,

$$\left\lfloor\frac{n}{p^i}\right\rfloor = \left\lfloor\frac{\widehat{n}}{p^{i-1}} + \frac{c_n}{p^i}\right\rfloor = \left\lfloor\frac{\widehat{n}}{p^{i-1}}\right\rfloor$$

De là,

$$\nu_p(n!) = \sum_{i\geq 1}\left\lfloor\frac{n}{p^i}\right\rfloor = \widehat{n} + \sum_{i\geq 2}\left\lfloor\frac{\widehat{n}}{p^{i-1}}\right\rfloor = \widehat{n} + \nu_p(\widehat{n}!)$$

Ainsi, on a pour tous entiers n, k tels que $0 \leq k \leq n$,

$$\nu_p\binom{n}{k} = \widehat{n} - \widehat{k} - \widehat{n-k} + \nu_p(\widehat{n}!) - \nu_p(\widehat{k}!) - \nu_p(\widehat{n-k}!)$$

Remarquons que $n - k = p(\widehat{n} - \widehat{k}) + (c_n - c_k)$. Plusieurs cas se présentent alors.

- Cas 0 : $k = 0$. Dans ce cas, $\nu_p\binom{n}{k} = 0$.
- Cas 1 : $c_n \geq c_k$. Alors, $\widehat{n-k} = \widehat{n} - \widehat{k}$ et donc $\nu_p\binom{n}{k} = \nu_p\binom{\widehat{n}}{\widehat{k}}$.
- Cas 2 : $c_n < c_k$. On a alors $n - k = p(\widehat{n} - \widehat{k} - 1) + (c_n - c_k + p)$. Comme $0 \leq c_n - c_k + p < p$, on a donc $\widehat{n-k} = \widehat{n} - \widehat{k} - 1$. Posons $\widehat{n} = p\widehat{\widehat{n}} + b_n$ où $0 \leq b_n < p$, et de même pour $\widehat{k}$.
  - Cas 2.1 : $b_n \neq b_k$. $\widehat{n} - \widehat{k}$ n'est pas un multiple de p, et donc $\nu_p((\widehat{n} - \widehat{k} - 1)!) = \nu_p((\widehat{n} - \widehat{k})!)$. Dans ce cas,

$$\nu_p\binom{n}{k} = \nu_p\binom{\widehat{n}}{\widehat{k}} + 1$$

– Cas 2.2 : $b_n = b_k \neq 0$. $\widehat{n}$ n'est pas un multiple de p, donc $\nu_p(\widehat{n}!) = \nu_p((\widehat{n}-1)!)$. Dans ce cas,

$$\nu_p\binom{n}{k} = \nu_p\binom{\widehat{n}-1}{\widehat{k}} + 1$$

– Cas 2.3 : $b_n = b_k = 0$. $\widehat{k}$ est un multiple de p, donc $\widehat{k}+1$ n'en est pas un. Ainsi, $\nu_p(\widehat{k}!) = \nu_p((\widehat{k}+1)!)$. Dans ce cas,

$$\nu_p\binom{n}{k} = \nu_p\binom{\widehat{n}}{\widehat{k}+1} + 1$$

Nous sommes donc ramenés, pour calculer $\nu_p\binom{n}{k}$, au calcul de la valuation p adique d'un coefficient binomial avec des paramètres strictement plus petits que n et k (ce n'est pas tout à fait vrai, voir le paragraphe 5.5).

1.5.4 5.4 La fonction Python

Pour tout entier n et tout nombre premier p, posons

$$n = a_n p^2 + b_n p + c_n$$

où $a_n = \widehat{\widehat{n}} \in \mathbb{N}$ et $b_n, c_n \in [|0, p-1|]$. On a $c_n = n \bmod p$ et $b_n = (\frac{1}{p}(n - c_n)) \bmod p$.

La fonction `decomp` ci-dessous prend un entier n en paramètre et renvoie le triplet (a_n, b_n, c_n).

[46]:
```
def decomp(n, p):
    c = n % p
    n1 = n // p
    b = n1 % p
    a = n1 // p
    return (a, b, c)
```

La fonction `val_binom` utilise les relations que nous avons montrées dans le paragraphe précédent pour calculer $\nu_p\binom{n}{k}$.

[47]:
```
def val_binom(n, k, p):
    if k == 0: return 0
    else:
        an, bn, cn = decomp(n, p)
        ak, bk, ck = decomp(k, p)
        if cn >= ck: return val_binom(n // p, k // p, p)
        elif bn != bk: return val_binom(n // p, k // p, p) + 1
        elif bn != 0: return val_binom(n // p - 1, k // p, p) + 1
```

```
        else: return val_binom(n // p, k // p + 1, p) + 1
```

Remarque. Tout ce que nous pouvons dire pour l'instant c'est que **SI** `val_binom(n, k, p)` termine, alors il renvoie le bon résultat. À ceux qui trouveraient cette phrase bizarre, je propose la définition universelle suivante : `def f(x): return f(x)`. Eh bien, si l'appel `f(x)` termine, il renvoie bien $f(x)$. Mais il ne termine pas .

Avant de prouver la terminaison de `val_binom`, prenons un exemple pour nous rassurer.

[48]:
```
print(fmt(11) % tuple([binomial2(10, k) for k in range(11)]))
print(fmt(11) % tuple([val_binom(10, k, 2) for k in range(11)]))
```

```
     1     10     45    120    210    252    210    120     45     10      1
     0      1      0      3      1      2      1      3      0      1      0
```

1.5.5 5.5 La terminaison et la complexité de `val_binom`

Notons $C_{n,k}$ le nombre d'appels récursifs effectués par `val_binom(n, k, p)`, en convenant que $C_{n,k} = \infty$ si l'appel ne termine pas. Quelles sont les équations vérifiées par $C_{n,k}$? Il suffit de renprendre le code de `val_binom`. Avec les notations de cette fonction (et en convenant que $\infty + 1 = \infty$),

- $C_{n,0} = 0$.
- Si $k \neq 0$ et $c_n \geq c_k$, $C_{n,k} = C_{\widehat{n,k}} + 1$.
- Si $k \neq 0$, $c_n < c_k$ et $b_n \neq b_k$, $C_{n,k} = C_{\widehat{n,k}} + 1$.
- Si $k \neq 0$, $c_n < c_k$ et $b_n = b_k \neq 0$, $C_{n,k} = C_{\widehat{n-1,k}} + 1$.
- Si $k \neq 0$, $c_n < c_k$ et $b_n = b_k = 0$, $C_{n,k} = C_{\widehat{n-1,k}} + 1$.

[49]:
```
def complex_val_binom(n, k, p):
    if k == 0: return 0
    elif k > n: return 0
    else:
        an, bn, cn = decomp(n, p)
        ak, bk, ck = decomp(k, p)
        if cn >= ck: return complex_val_binom(n // p, k // p, p) + 1
        elif bn != bk: return complex_val_binom(n // p, k // p, p) + 1
        elif bn != 0: return complex_val_binom(n // p - 1, k // p, p) + 1
        else: return complex_val_binom(n // p, k // p + 1, p) + 1
```

[50]:
```
N = 30
p = 2
for n in range(1, N):
    print(fmt(N, 2) % tuple([complex_val_binom(n, k, p) for k in range(N)]))
```

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 2 2 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

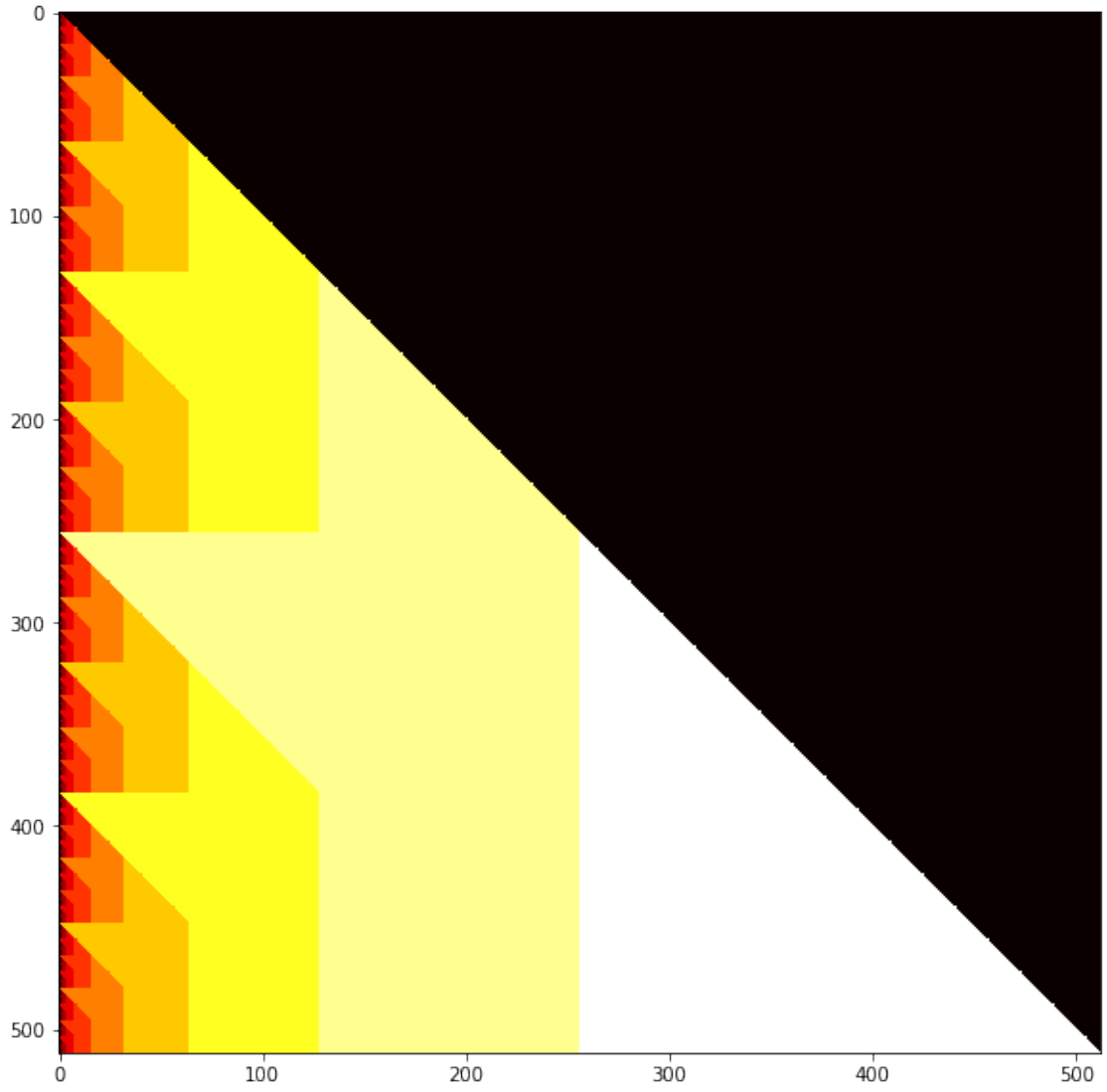
```
0 1 2 2 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 3 3 3 3 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 3 3 3 3 3 3 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 3 3 3 3 3 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 2 2 2 3 3 3 3 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0
0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 0 0 0 0 0 0 0 0 0 0 0
0 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 0 0 0 0 0 0 0 0 0 0 0
0 1 2 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 0 0 0 0 0 0 0 0 0 0
0 1 2 2 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 0 0 0 0 0 0 0 0 0
0 2 2 2 3 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 0 0 0 0 0 0 0 0
0 1 2 2 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 0 0 0 0 0 0 0
0 1 2 2 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 0 0 0 0 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 0 0 0 0 0 0
0 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 0 0 0 0 0
0 1 3 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 0 0 0 0
0 1 2 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 0 0 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 0 0
0 2 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 0
0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Plus « graphiquement » :

[51]:
```
plt.rcParams['figure.figsize'] = (10, 10)
```

[52]:
```
def plot_complex(N, p):
    m = (N + 1) * [None]
    for i in range(N + 1): m[i] = (N + 1) * [None]
    for n in range(N + 1):
        for k in range(N + 1):
            m[n][k] = complex_val_binom(n, k, p)
    plt.imshow(m, interpolation='none', cmap='hot')
```

[53]:
```
plot_complex(511, 2)
```



Tout ceci est très joli mais il est temps de se mettre au travail.

Proposition. Pour tout entier $n \geq 1$ et tout entier $k \leq n$, $C_{n,k} \neq \infty$ et

$$C_{n,k} \leq \log_p n + 1$$

Démonstration. On fait une récurrence forte sur n.

- Pour $n = 1$ c'est évident.
- Soit $n > 1$. Supposons la propriété vérifiée pour $1 \leq n' < n$.

$$C_{n,k} = C_{\widehat{n-\varepsilon},k'} + 1$$

où ε vaut 0 ou 1 et, **admettons-le provisoirement**, $k' \leq \widehat{n} - \varepsilon$. Clairement, $\widehat{n} - \varepsilon < n$. Par l'hypothèse de récurrence, $C_{\widehat{n}-\varepsilon,k'} \neq \infty$. Deux cas se présentent.

- Si $\widehat{n} \geq 1$, on applique l'hypothèse de récurrence :

$$C_{n,k} \leq \log_p \widehat{n} + 1 + 1 \leq \log_p \frac{n}{p} + 2 = \log_p n + 1$$

- Si $\widehat{n} = 0$, nous avons $0 \leq k \leq n < p$. Avec les notations de la fonction `val_binom`, on a $c_n = n$, $c_k = k$. Ainsi, `val_binom` s'appelle récursivement sur `val_binom(0, 0, p)` qui renvoie imméfiatement 0. Il y a donc 1 appel récursif. Ainsi, $C_{n,k} = 1 \leq \log_p n + 1$ puisque $\log_p n < 1$.

La terminaison et la complexité de ***`val_binom`*** *seront donc prouvées dès que nous aurons montré la proposition suivante.*

Lemme. Soient $n, k \in \mathbb{N}$ vérifiant $k \leq n$. Si `val_binom(n, k, p)` se rappelle récursivement, c'est avec deux paramètres $n', k' \in \mathbb{N}$ tels que $k' \leq n'$.

Démonstration. Un certain nombre de cas sont à considérer :

1. $k \neq 0$ et $c_n \geq c_k$: $n' = \widehat{n}$, $k' = \widehat{k}$.
2. $k \neq 0$, $c_n < c_k$ et $b_n \neq b_k$: $n' = \widehat{n}$, $k' = \widehat{k}$.
3. $k \neq 0$, $c_n < c_k$ et $b_n = b_k \neq 0$: $n' = \widehat{n} - 1$, $k' = \widehat{k}$.
4. $k \neq 0$, $c_n < c_k$ et $b_n = b_k = 0$: $n' = \widehat{n}$, $k' = \widehat{k} + 1$.

Puisque $k \leq n$, on a $\frac{k}{p} \leq \frac{n}{p}$. Par croissance de la partie entière on obtient $\widehat{k} \leq \widehat{n}$. Les cas 1 et 2 sont donc réglés.

Peut-on avoir $\widehat{k} = \widehat{n}$? Supposons que cela soit le cas, appelons a la valeur commune de ces deux nombres. On a alors

$$a \leq \frac{k}{p} < a + 1 \text{ et } a \leq \frac{n}{p} < a + 1$$

On en tire facilement que $n - k < p$. Si l'on est dans le cas 3 ou le cas 4, alors

$$n = a_n p^2 + b_n p + c_n \text{ et } k = a_k p^2 + b_k p + c_k$$

On a $b_n = b_k$, donc

$$n - k = (a_n - a_k)p^2 + (b_n - b_k)p + c_n - c_k = (a_n - a_k)p^2 + c_n - c_k$$

Comme $c_n < c_k$ et $n - k \geq 0$, on a nécessairement $a_n - a_k \geq 1$. Ainsi,

$$n - k = (a_n - a_k)p^2 + c_n - c_k \geq p^2 + c_n - c_k \geq p^2 - p = p(p-1) \geq p$$

On n'a donc pas $n - k < p$. Ainsi, $\widehat{k} < \widehat{n}$, et donc

- Pour le cas 3 : $k' = \widehat{k} \leq \widehat{n} - 1 = n'$.
- Pour le cas 4 : $k' = \widehat{k} + 1 \leq \widehat{n} = n'$.

Les cas 3 et 4 sont donc aussi réglés.

La fonction `val_binom` est donc de complexité logarithmique en n, c'est à dire linéaire en le nombre de chiffres de n (en base p). On pouvait difficilement espérer mieux !

1.5.6 5.6 Factorisation des coefficients binomiaux

Nous voici maintenant capables de factoriser les coefficients binomiaux. Pour factoriser $\binom{n}{k}$, effectuer les opérations suivantes :

- Remarquer que si un nombre premier p divise $n!$, alors il divise un entier inférieur ou égal à n. Ainsi, $p \leq n$.
- Pour tout nombre premier $p \leq n$, déterminer $\nu_p\binom{n}{k}$.

Voici le code Python. La fonction `est_premier` teste naïvement si p est un nombre premier.

[54]:
```
def est_premier(p):
    k = 2
    while k * k <= p and p % k != 0: k = k + 1
    return k * k > p
```

[55]:
```
print([p for p in range(2, 100) if est_premier(p)])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]
```

La fonction `factor_binomial` renvoie la factorisation de $\binom{n}{k}$.

[56]:
```
def factor_binomial(n, k):
    ps = [p for p in range(2, n + 1) if est_premier(p)]
    s = []
    for p in ps:
        m = val_binom(n, k, p)
        if m != 0: s.append((p, m))
    return s
```

L'exemple ci-dessous est pris dans l'article donné en référence à la fin du notebook. Cet article date de 1987, l'auteur avait alors obtenu la factorisation en 1/2 seconde sur un PC équipé d'un processeur 8086 à 8 MHz avec un programme écrit en langage Pascal.

[57]:
```
print(factor_binomial(1000, 353))
```

```
[(2, 3), (3, 6), (5, 3), (11, 1), (19, 1), (29, 1), (31, 2), (37, 1), (41, 1),
(47, 1), (59, 1), (61, 1), (71, 1), (73, 1), (83, 1), (89, 1), (97, 1), (109,
1), (131, 1), (137, 1), (139, 1), (163, 1), (179, 1), (181, 1), (191, 1), (193,
1), (197, 1), (199, 1), (223, 1), (227, 1), (229, 1), (233, 1), (239, 1), (241,
1), (331, 1), (359, 1), (367, 1), (373, 1), (379, 1), (383, 1), (389, 1), (397,
1), (401, 1), (409, 1), (419, 1), (421, 1), (431, 1), (433, 1), (439, 1), (443,
1), (449, 1), (457, 1), (461, 1), (463, 1), (467, 1), (479, 1), (487, 1), (491,
1), (499, 1), (653, 1), (659, 1), (661, 1), (673, 1), (677, 1), (683, 1), (691,
```

```
1), (701, 1), (709, 1), (719, 1), (727, 1), (733, 1), (739, 1), (743, 1), (751,
1), (757, 1), (761, 1), (769, 1), (773, 1), (787, 1), (797, 1), (809, 1), (811,
1), (821, 1), (823, 1), (827, 1), (829, 1), (839, 1), (853, 1), (857, 1), (859,
1), (863, 1), (877, 1), (881, 1), (883, 1), (887, 1), (907, 1), (911, 1), (919,
1), (929, 1), (937, 1), (941, 1), (947, 1), (953, 1), (967, 1), (971, 1), (977,
1), (983, 1), (991, 1), (997, 1)]
```

Notre machine donne le résultat en zéro seconde, mais nous n'avons aucun mérite. C'est juste qu'elle est équipée d'un processeur à 4 coeurs tournant à 2.93 GHz .

1.5.7 5.7 Une fonction de calcul rapide des coefficients binomiaux

Nous voici en possession d'un nouvel algorithme permettant de calculer les coefficients binomiaux. En effet,

$$\binom{n}{k} = \prod_{p\in\mathcal{P}, p\leq n} p^{\nu_p\binom{n}{k}}$$

[58]:
```
def binomial4(n, k):
    ps = factor_binomial(n, k)
    b = 1
    for p, m in ps:
        b = b * p ** m
    return b
```

[59]:
```
N = 11
for n in range(N):
    print(fmt(n + 1, 4) % tuple([binomial4(n, k) for k in range(n + 1)]))
```

```
   1
   1   1
   1   2   1
   1   3   3   1
   1   4   6   4   1
   1   5  10  10   5   1
   1   6  15  20  15   6   1
   1   7  21  35  35  21   7   1
   1   8  28  56  70  56  28   8   1
   1   9  36  84 126 126  84  36   9   1
   1  10  45 120 210 252 210 120  45  10   1
```

[60]:
```
print(binomial4(1000, 353))
```

```
252294456330659742351440802520557737356130435153119568936355943885445596891848033301801495281415129453596585561663993923461189184397715091949204595205562522956838053320988825023746367692580376666922328125927686787505911718832270161158914674304910679826394724366531380353822141070000
```

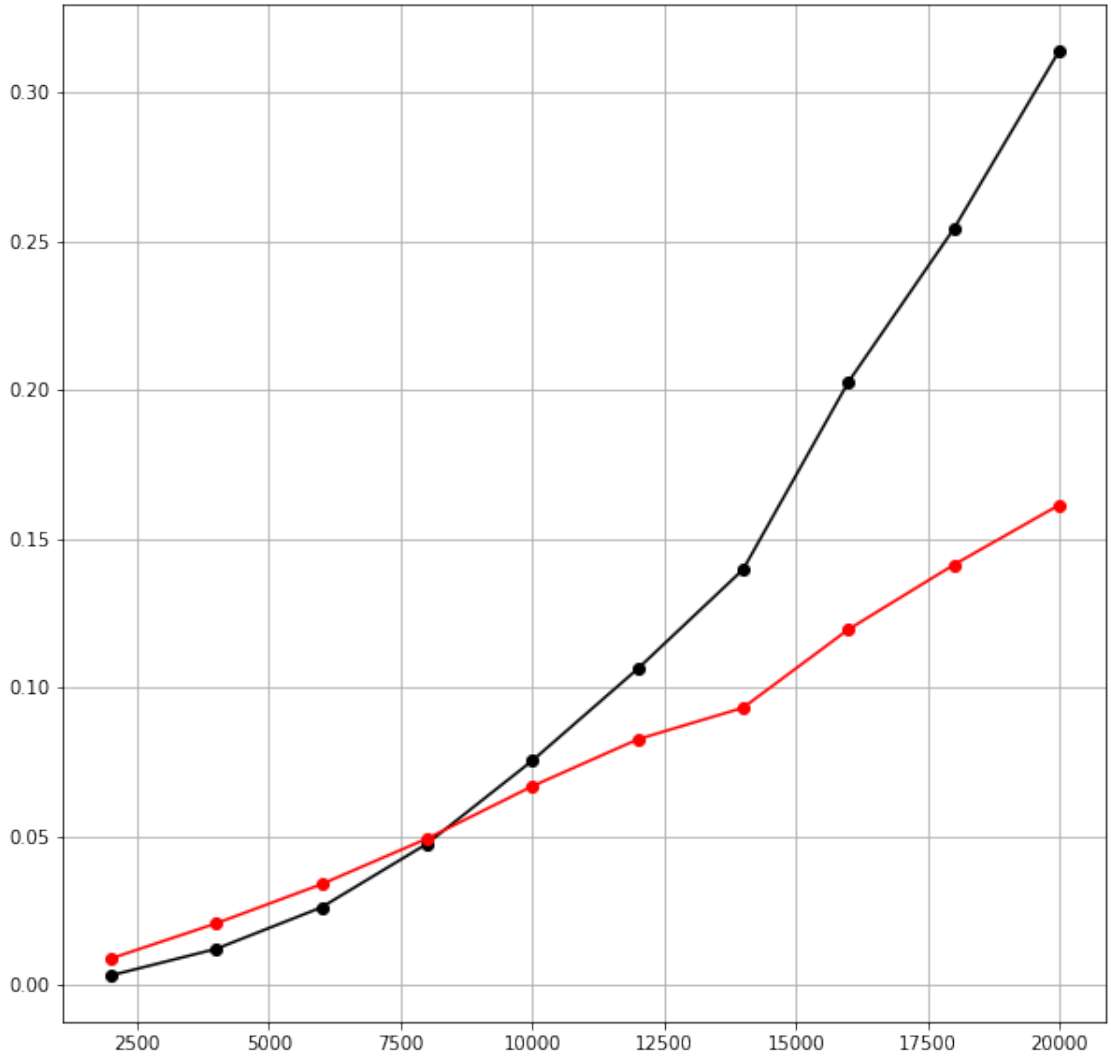
Ici encore, résultat en zéro seconde. L'auteur de l'article avait dû patienter 8 secondes pour effectuer le produit.

1.5.8 5.8 Complexité

Comparons les temps d'exécution de `binomial2` (en noir) et `binomial4` (en rouge).

[61]:
```
def test_binomial4(N):
    s2 = []
    s4 = []
    ns = list(range(N, 11 * N, N))
    for n in ns:
        t1 = time.time()
        x = binomial4(2 * n, n)
        t2 = time.time()
        s4.append(t2 - t1)
        t1 = time.time()
        x = binomial2(2 * n, n)
        t2 = time.time()
        s2.append(t2 - t1)
    plt.plot(ns, s2, '-ok')
    plt.plot(ns, s4, '-or')
    plt.grid()
```

[62]:
```
test_binomial4(2000)
```

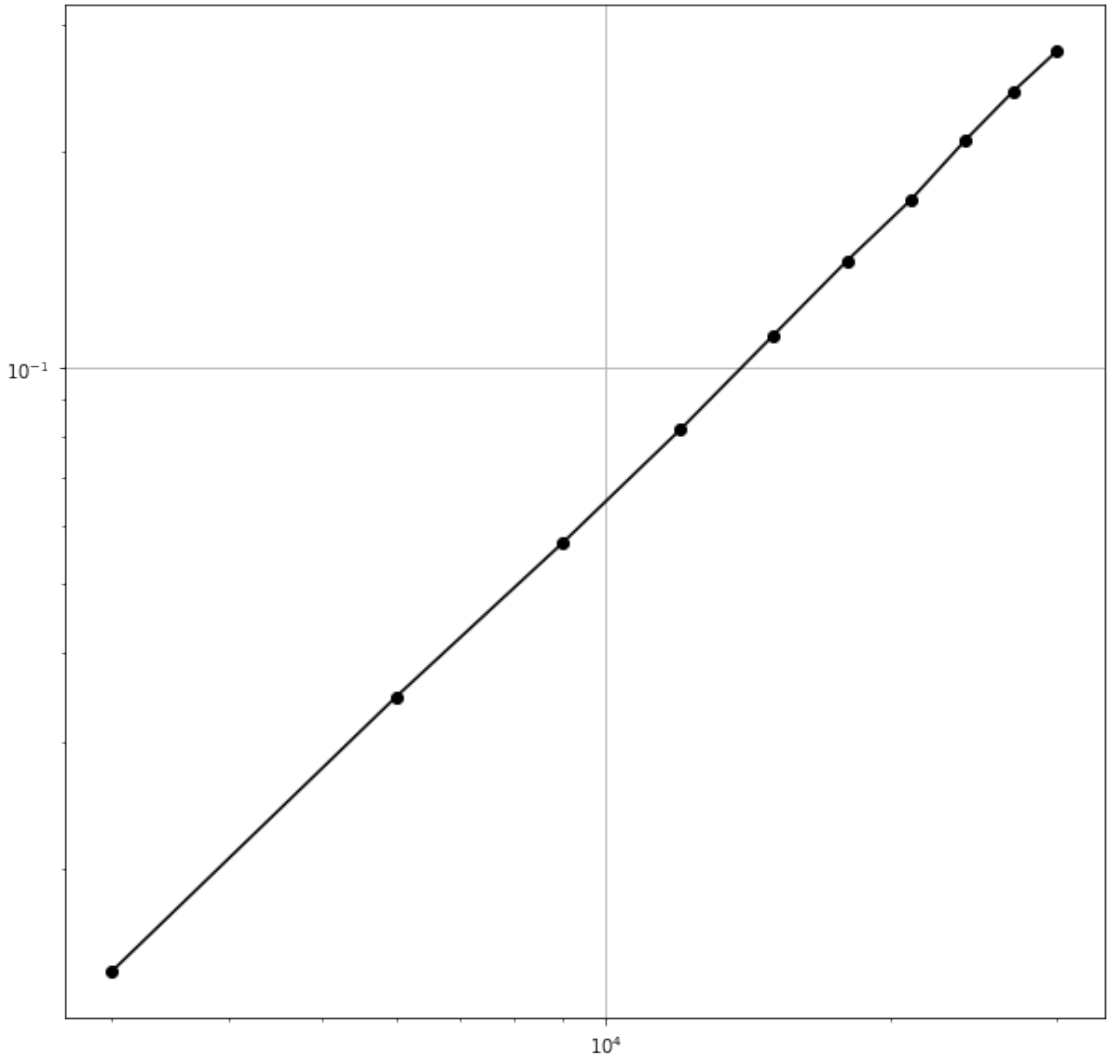


Pour de petites valeurs de n, `binomial2`est le vainqueur. Mais en gros pour $n \geq 8000$, c'est `binomial4`qui est le meilleur.

Reprenons maintenant pour `binomial4` les tests que nous avions faits pour `binomial2`.

[63]:
```
def test_binomial4bis(N):
    s = []
    ns = list(range(N, 11 * N, N))
    for n in ns:
        t1 = time.time()
        x = binomial4(2 * n, n)
        t2 = time.time()
        s.append(t2 - t1)
    plt.loglog(ns, s, '-ok')
    plt.grid()
```

[64]:
```
test_binomial4bis(3000)
```



La courbe est quasiment une droite. Si nous appelons T_n le temps mis pour calculer $\binom{2n}{n}$ on a donc à peu près

$$\ln T_n = d + \beta \log n$$

où d et β sont deux constantes. β est la pente de la droite.

[65]:
```
t1 = time.time()
na = 50000
x = binomial4(2 * na, na)
t2 = time.time()
ta = t2 - t1
```

```
t1 = time.time()
nb = 100000
x = binomial4(2 * nb, nb)
t2 = time.time()

tb = t2 - t1

beta = (math.log(tb) - math.log(ta)) / (math.log(nb) - math.log(na))
print(beta)
```

```
1.4137822940948956
```

Rappelons-nous que pour la fonction `binomial2` nous avions une pente supérieure à 2. Obtenir la constante d est maintenant facile.

[66]:
```
d = math.log(tb) - beta * math.log(nb)
print(d)
```

```
-15.885628510340448
```

Maintenant, $T_n = e^d n^\beta$.

[67]:
```
def temps_binomial4(n):
    k = math.exp(d)
    return k * n ** beta
```

[68]:
```
temps_binomial4(1000000)
```

[68]:
```
38.34007525279209
```

Cette estimation nous rend confiants. Nous devrions pouvoir enfin calculer $\binom{2000000}{1000000}$.

[69]:
```
t1 = time.time()
x = binomial4(2000000, 1000000)
t2 = time.time()
print('temps : %fs'% (t2 - t1))
```

```
temps : 41.748097s
```

C'est un peu plus que le temps prévu, mais nous avons gagné notre pari.

1.6 Références

- P. Goetgheluck, Computing Binomial Coefficients - The American Mathematical Monthly, Vol. 94, No. 4 (Apr. 1987), pp. 360-365.