Partitions d'un entier

Marc Lorenzi - 21 août 2018

In [1]:
```
from math import *
import sys
import matplotlib.pyplot as plt
```

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

1. Introduction

Dans le film *The Man Who Knew Infinity*, le mathématicien S. Ramanujan fait le pari avec P.A. Macmahon qu'il sera capable en quelques mois de calculer $P(200)$, le nombre de partitions de l'entier 200. La scène est probablement inventée. Ce qui est vrai, en revanche, c'est que Percy Alexander Macmahon utilisa une formule due à Euler (nous la verrons dans ce notebook) pour calculer les valeurs de $P(n)$ jusqu'à $n = 200$. Cela se passait au tout début du XXème siècle. Et nous, pouvons-nous, avec Python, faire ce même calcul en moins de 1 seconde ? Pouvons-nous calculer $P(100000)$ en moins d'une minute ? Et au fait, c'est quoi une partition ?

1.1 C'est quoi une partition ?

Définition : Soit n un entier naturel non nul. Une **partition** de n est une suite (finie) d'entiers naturels non nuls $(k_1, k_2, \dots, k_p)$ (où $p \geq 1$) telle que $\sum_{i=1}^{p} k_i = n$. Toute permutation d'une telle suite étant encore une partition de n, nous supposerons sans perte de généralité que $k_1 \geq k_2 \geq \dots \geq k_p$.

Conventions :

- Nous conviendrons que l'entier 0 possède une unique partition : la suite vide.
- Nous conviendrons aussi que les entiers négatifs n'ont pas de partition.

Notation : Pour tout $n \in \mathbb{Z}$, on note $P(n)$ le nombre de partitions de l'entier n.

Cas particuliers : On a donc $P(0) = 1$ et pour tout $n < 0$, $P(n) = 0$.

Exemples :

- 1 n'a qu'une partition qui est (1). On a $P(1) = 1$.
- Les partitions de 2 sont (2) et $(1, 1)$. On a $P(2) = 2$.
- Les partitions de 3 sont (3), $(2, 1)$ et $(1, 1, 1)$. On a $P(3) = 3$.
- Les partitions de 4 sont (4), $(3, 1)$, $(2, 2)$, $(2, 1, 1)$ et $(1, 1, 1, 1)$. On a $P(4) = 5$.
- Les partitions de 5 sont (5), $(4, 1)$, $(3, 1, 1)$, $(2, 1, 1, 1)$, $(1, 1, 1, 1, 1)$, $(2, 2, 1)$ et $(3, 2)$. On a $P(5) = 7$.

Exercice : Déterminer les partitions de 6.

1.2 Qu'allons-nous faire ?

La théorie des partitions est une théorie très riche. Il y a énormément de choses à dire sur le sujet. Nous allons rester très modestes : nous allons uniquement dans ce qui suit décrire trois algorithmes de calcul de $P(n)$.

- Le premier est naïf et terriblement inefficace, il permet d'envisager le calcul de $P(50)$, et pas beaucoup plus.
- Le second est une adaptation du premier. Il permet d'aller jusqu'à environ $P(1000)$.
- Le troisième utilise une formule due à Euler. Il nous donnera sereinement $P(100000)$.

Nous n'atteindrons pas $P(1000000)$, mais nous donnerons à la fin du notebook un équivalent de $P(n)$ lorsque n tend vers l'infini. Cela nous permettra d'avoir une idée de la *taille* de ce nombre.

2. Calculer toutes les partitions

Comment déterminer toutes les partitions de l'entier n ? Étant donnée une partition $(k_1, k_2, \dots, k_p)$ de n, la suite $(k_1, k_2, \dots, k_{p-1})$ est une partition de l'entier $n - k_p$. Mais, attention, cette partition de $n - k_p$ a une propriété supplémentaire : les nombres de cette partition sont supérieurs à k_p, puisque nous avons convenu d'écrire une partition dans l'ordre décroissant.

Nous allons donc écrire une fonction `part(n, k)` qui renvoie la liste des partitions de n dont tous les éléments sont supérieurs ou égaux à k. Bien entendu, le calcul des partitions de n se fera en évaluant `part(n, 1)`.

- Si $k > n$ c'est évident : il n'y a aucune telle partition.
- Sinon, pour tous les entiers j compris entre k et $n - k$, on calcule les partitions de $n - j$ dont les éléments sont supérieurs ou égaux à j (hypothèse de décroissance des suites) et à k (uniquement des entiers supérieurs à k). On rajoute j à la fin de ces partitions.

In [3]:
```
def part(n, k=1):
    if k > n: return []
    else:
        s = [[n]]
        for j in range(k, n - k + 1):
            s1 = part(n - j, max(j, k))
            s1 = [t + [j] for t in s1]
            s = s + s1
        return s
```

In [4]:
```
part(5)
```

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

In [5]:
```
print(part(10))
```

```
[[10], [9, 1], [8, 1, 1], [7, 1, 1, 1], [6, 1, 1, 1, 1], [5, 1, 1, 1
, 1, 1], [4, 1, 1, 1, 1, 1, 1], [3, 1, 1, 1, 1, 1, 1, 1], [2, 1, 1,
1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 2, 1, 1, 1, 1
, 1, 1], [3, 2, 1, 1, 1, 1, 1], [4, 2, 1, 1, 1, 1], [2, 2, 2, 1, 1,
1, 1], [3, 3, 1, 1, 1, 1], [5, 2, 1, 1, 1], [3, 2, 2, 1, 1, 1], [4,
3, 1, 1, 1], [6, 2, 1, 1], [4, 2, 2, 1, 1], [2, 2, 2, 2, 1, 1], [3,
3, 2, 1, 1], [5, 3, 1, 1], [4, 4, 1, 1], [7, 2, 1], [5, 2, 2, 1], [3
, 2, 2, 2, 1], [4, 3, 2, 1], [6, 3, 1], [3, 3, 3, 1], [5, 4, 1], [8,
2], [6, 2, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2], [3, 3, 2, 2], [5, 3, 2
], [4, 4, 2], [7, 3], [4, 3, 3], [6, 4], [5, 5]]
```

In [6]:
```
len(part(10))
```

Out[6]:
```
42
```

Notre fonction `part` a évidemment une complexité épouvantable. Nous verrons un peu plus loin dans ce notebook que le nombre de partitions de n croît "exponentiellement" avec n. Toute fonction renvoyant les partitions de n a donc une complexité exponentielle. Cessons donc de nous intéresser au calcul des partitions, et intéressons-nous uniquement au **nombre** de partitions.

3. Nombre de partitions

3.1 Première tentative

Comment calculer le nombre de partitions d'un entier n ? On peut évidemment calculer la liste de toutes les partitions, puis déterminer la longueur de cette liste comme nous l'avons fait pour $n = 10$ un peu plus haut. Cet algorithme gaspille un espace énorme : on n'a pas besoin de stocker les partitions, on veut juste les compter. Adaptons donc la fonction `part` ci-dessus pour qu'elle compte au lieu de stocker ... la fonction `npart1` ci-dessous prend deux paramètres n et k. Elle renvoie le nombre de partitions de n dont tous les éléments sont supérieurs ou égaux à k.

In [7]:
```
def npart1(n, k=1):
    if k > n: return 0
    else:
        s = 1
        for j in range(k, n - k + 1):
            s = s + npart1(n - j, max(j, k))
        return s
```

In [8]:
```
npart1(10)
```

Out[8]:
```
42
```

Voici les premières valeurs de $P(n)$.

In [9]:
```
for k in range(21):
    print('%5d%5d' % (k, npart1(k)))
```

```
    0    0
    1    1
    2    2
    3    3
    4    5
    5    7
    6   11
    7   15
    8   22
    9   30
   10   42
   11   56
   12   77
   13  101
   14  135
   15  176
   16  231
   17  297
   18  385
   19  490
   20  627
```

In [12]:
```
npart1(50)
```

Out[12]:
```
204226
```

La complexité de `npart1` est exponentielle en n. Inutile, donc, d'essayer avec celle-ci d'obtenir le nombre de partitions de, mettons, 100. Peut-on faire mieux ? Oui, heureusement.

Exercice : quel est le plus grand entier n pour lequel (sur votre machine) la fonction `npart1` renvoie le nombre de partitions en moins d'une minute ?

3.2 Une fonction de complexité polynomiale cubique en n

Un inconvénient majeur de la fonction `npart` est qu'elle fait des appels récursifs un grand nombre de fois avec les mêmes valeurs du paramètre. Bref, elle calcule, recalcule et rerecalcule tout le temps la même chose. Pour nous en convaincre, la fonction `testnpart1` ci-dessous prend en paramètres deux entiers n et k et une matrice n. Après exécution de `testnpart1(n, m, k)`, m_{ij} est égal au nombre de fois que `npart1(i, j)` a été calculé lors de l'appel `npart1(n, k)`.

In [13]:
```
def testnpart1(n, m, k=1):
    if k > n: return 0
    else:
        m[n][k] += 1
        s = 1
        for j in range(k, n - k + 1):
            s = s + testnpart1(n - j, m, max(j, k))
        return s
```

In [14]:
```
n = 40
m = (n + 1) * [None]
for k in range(n + 1):
    m[k] = (n + 1) * [0]
testnpart1(n, m)

# Afficher joliment la partie intéressante de m
sys.stdout.write('    ')
for j in range(n // 2 + 1):
    sys.stdout.write('%4d' % (j))
print()
for i in range(n + 1):
    sys.stdout.write('%4d' % (i))
    for j in range(n // 2 + 1):
        if m[i][j] == 0: sys.stdout.write('%4s' % u'\u2022')
        else: sys.stdout.write('%4d' % (m[i][j]))
    print()
```

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

```
 0   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 1   •   1   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 2   •   1  19   •   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 3   •   1  18 114   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 4   •   1  18 108 351   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 5   •   1  17 102 321 674   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 6   •   1  17  96 297 603 931   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 7   •   1  16  91 270 540 8111009   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
 8   •   1  16  85 249 480 709 860 919   •   •   •   •   •   •   •   •   •

•   •   •   •   •
 9   •   1  15  80 225 427 612 733 764 732   •   •   •   •   •   •   •   •
•   •   •   •   •
10   •   1  15  75 206 377 532 618 638 598 530   •   •   •   •   •   •   •
•   •   •   •   •
11   •   1  14  70 185 333 454 522 525 488 423 355   •   •   •   •   •   •
•   •   •   •   •
12   •   1  14  65 169 291 391 436 434 393 340 278 224   •   •   •   •   •
•   •   •   •   •
13   •   1  13  61 150 255 331 364 352 318 267 219 172 134   •   •   •   •
•   •   •   •   •
14   •   1  13  56 136 221 282 300 288 252 212 169 133 101  77   •   •   •
•   •   •   •   •
15   •   1  12  52 120 192 235 248 230 201 164 131 100  77  56  42   •   •
•   •   •   •   •
16   •   1  12  48 108 164 199 201 186 157 128  99  77  56  42  30  22   •
•   •   •   •
17   •   1  11  44  94 141 163 164 146 123  97  76  56  42  30  22  15  11
•   •   •   •
18   •   1  11  40  84 119 136 131 116  94  75  56  42  30  22  15  11   7
5   •   •
19   •   1  10  37  72 101 110 105  89  73  55  42  30  22  15  11   7   5
3   2   •
20   •   1  10  33  64  84  90  82  70  54  42  30  22  15  11   7   5   3
2   1   1
21   •   1   9  30  54  70  71  65  52  41  30  22  15  11   7   5   3   2
1   1   •
22   •   1   9  27  47  57  58  49  40  30  22  15  11   7   5   3   2   1
1   •   •
23   •   1   8  24  39  47  44  38  29  22  15  11   7   5   3   2   1   1
•   •   •
24   •   1   8  21  34  37  35  28  22  15  11   7   5   3   2   1   1   •
•   •   •
25   •   1   7  19  27  30  26  21  15  11   7   5   3   2   1   1   •   •
•   •   •
26   •   1   7  16  23  23  20  15  11   7   5   3   2   1   1   •   •   •
```

```
  27   •   1   6  14  18  18  14  11   7   5   3   2   1   1   •   •
•   •   •   •   •
  28   •   1   6  12  15  13  11   7   5   3   2   1   1   •   •   •
•   •   •   •   •
  29   •   1   5  10  11  10   7   5   3   2   1   1   •   •   •   •
•   •   •   •   •
  30   •   1   5   8   9   7   5   3   2   1   1   •   •   •   •   •
•   •   •   •   •
  31   •   1   4   7   6   5   3   2   1   1   •   •   •   •   •   •
•   •   •   •   •
  32   •   1   4   5   5   3   2   1   1   •   •   •   •   •   •   •
•   •   •   •   •
  33   •   1   3   4   3   2   1   1   •   •   •   •   •   •   •   •
•   •   •   •   •
  34   •   1   3   3   2   1   1   •   •   •   •   •   •   •   •   •
•   •   •   •   •

  35   •   1   2   2   1   1   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
  36   •   1   2   1   1   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
  37   •   1   1   1   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
  38   •   1   1   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
  39   •   1   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
  40   •   1   •   •   •   •   •   •   •   •   •   •   •   •   •   •
•   •   •   •   •
```

Par exemple, lors de l'appel `npart1(40)`, `npart1(10, 6)` est calculé 532 fois. Essayez avec $n = 50$, certains calculs sont effectués ... des milliers de fois !

Pour éviter cela il suffit de stocker tous les résultats dans une matrice P. Pour $1 \leq i \leq n$ et $1 \leq j \leq n$, P_{ij} est le nombre de partitions de l'entier i ne contenant que des entiers supérieurs ou égaux à j. On peut même éliminer tout recours à la récursivité.

In [16]:
```
def npart2(n, mat=False):
    P = (n + 1) * [None]
    for k in range(n + 1):
        P[k] = (n + 1) * [0]
    for j in range(n + 1): P[0][j] = 1
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            P[i][j] = 1
            for k in range(j, i - j + 1):
                P[i][j] = P[i][j] + P[i - k][max(k, j)]
    if mat: return P
    else: return P[n][1]
```

In [17]:
```
npart2(10, mat=True)
```

Out[17]:
```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 3, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 5, 2, 1, 1, 0, 0, 0, 0, 0, 0],
 [0, 7, 2, 1, 1, 1, 0, 0, 0, 0, 0],
 [0, 11, 4, 2, 1, 1, 1, 0, 0, 0, 0],
 [0, 15, 4, 2, 1, 1, 1, 1, 0, 0, 0],
 [0, 22, 7, 3, 2, 1, 1, 1, 1, 0, 0],
 [0, 30, 8, 4, 2, 1, 1, 1, 1, 1, 0],
 [0, 42, 12, 5, 3, 2, 1, 1, 1, 1, 1]]
```

In [18]:
```
npart2(10)
```

Out[18]:
```
42
```

Quelle est la complexité de notre fonction `npart2` ? Sans entrer dans les détails, la triple boucle `for` en i, j, k annonce une complexité en $\mathcal{O}(n^3)$. Nous avons donc maintenant une complexité polynomiale en n. Bien entendu l'exposant n^3 nous dit qu'il faudra rester raisonnables. Mais le calcul de $P(100)$ est maintenant immédiat.

In [19]:
```
print(npart2(100))
```

```
190569292
```

Et $P(1000)$? Eh bien, $1000^3 = 10^9$, c'est donc faisable en étant un tout petit peu patient (une quarantaine de secondes sur ma machine).

In [20]:
```
print(npart2(1000))
```

```
24061467864032622473692149727991
```

Le calcul de $P(10000)$ prendrait $10^3 = 1000$ fois plus de temps. Il faut trouver mieux que notre algorithme cubique.

3.3 Peut-on faire mieux ?

Oui, on peut faire mieux. La formule ci-dessous est non triviale, elle a été énoncée par Euler.

Théorème : Pour tout $n \geq 1$, on a
$P(n) = \sum_{k=1}^{\infty}(-1)^{k-1}[P(n - \frac{1}{2}k(3k-1)) + P(n - \frac{1}{2}k(3k+1)]$

Remarque : La somme ci-dessus est en réalité finie puisque son terme général est nul dès que $\frac{1}{2}k(3k-1) > n$. Cela nous donne environ $2\sqrt{\frac{2n}{3}}$ termes, ce qui est donc bien mieux que les sommes que nous avons considérées jusqu'à présent. Gardez cette valeur dans un coin de votre esprit, en mathématiques tout fait partie d'un grand plan :-).

Démonstration : Je ne ferai pas la démonstration de la formule d'Euler (encore une !!!) mais je ne résiste pas à la tentation d'évoquer sa provenance. On peut montrer que pour tout réel x tel que $|x| < 1$, on a la merveilleuse égalité

$$\sum_{n=0}^{\infty} P(n)x^n = \frac{1}{(1-x)(1-x^2)(1-x^3)\dots} = \prod_{k=1}^{\infty}\frac{1}{1-x^k}$$

La série du membre de gauche est la **série génératrice** de la suite $(P(n))_{n\geq 0}$. Le membre de droite est une fonction de x, écrite sous forme d'un **produit infini**. Cette fonction de x contient en essence les valeurs de $P(n)$ pour **tous** les entiers n. En développant le produit infini $\prod_{k=1}^{\infty}(1-x^k)$, puis en multipliant le résultat obtenu par $\sum_{n=0}^{\infty} P(n)x^n$, on obtient, en vertu de la merveilleuse égalité ... 1. La formule qui nous intéresse en découle.

Voici maintenant la fonction `npart3` . Elle prend un entier n en paramètre. Puis elle calcule $P(i)$ pour $i = 1, 2, \dots, n$ en utilisant la formule d'Euler. Nous suivons donc la méthode employée par Macmahon pour calculer $P(200)$ à la main. Les valeurs des $P(i)$ sont stockées au fur et à mesure dans une liste P.

Je n'entrerai pas dans les détails, mais une écriture soigneuse de l'algorithme permet le calcul des $P(k)$, $1 \leq k \leq n$ en faisant $O(n^2)$ opérations élémentaires. On peut vérifier que la fonction `npart3` effectue $O(n^{\frac{3}{2}})$ additions sur des entiers. Bien évidemment, les entiers en question peuvent être très grands et une simple addition peut devenir coûteuse. Une estimation plus fine de la taille des entiers mis en jeu donnerait (ou pas :-)) la complexité $O(n^2)$ espérée.

In [21]:
```
def npart3(n, mat=False):
    P = (n+1) * [0]
    P[0] = 1
    for i in range(1, n + 1):
        k = 1
        s = 0
        while True:
            u = i - k * (3 * k - 1) // 2
            v = i - k * (3 * k + 1) // 2
            if u < 0 and v < 0: break
            if u >= 0: s = s + (-1) ** (k - 1) * P[u]
            if v >= 0: s = s + (-1) ** (k - 1) * P[v]
            k = k + 1
        P[i] = s
    if mat: return P
    else: return P[n]
```

Maintenant, le calcul de $P(1000)$ est immédiat.

In [22]:
```
npart3(1000)
```

Out[22]:
```
24061467864032622473692149727991
```

Le calcul de $P(10000)$ est quasi-immédiat.

In [23]:
```
npart3(10000)
```

Out[23]:
```
36167251325636293988820471890953695495016030339315650422081868605887
95256875406642059231055605290691643514 4
```

Et $P(100000)$? Aussi, mais soyez un petit peu patients si vous voulez avoir la réponse (environ 50 secondes sur ma machine).

In [24]:
```
npart3(100000)
```

Out[24]:
```
27493510569775696512677516320986352688173429315980054758203125984302
14732811496417305505074166073662159015784477429624894049306307020046
17927644930335101160793424571901557189435097253124661084520063695589
34464248716828789832182345009262853831404597021307130674510624419227
31123899970228440860937093553162969785156956989219610848015860056942
1098519
```

Exercice : Faites comme Macmahon et calculez *à la main* $P(k)$ pour $k = 1, 2, 3, \ldots$ avec l'algorithme utilisé par `npart3` . Où en êtes-vous au bout d'une heure ? Dans le film cité plus haut, il est dit que Macmahon a obtenu $P(100)$ en un mois :-).

3.4 La "complexité expérimentale" de `npart3`

Nous avons dit un peu plus haut que la fonction `npart3` effectuait $O(n^{\frac{3}{2}})$ additions pour calculer les valeurs de $P(k)$, $k = 1, \dots, n$. Nous avons également remarqué que, ces additions opérant sur de grands entiers, la complexité réelle de notre fonction était plutôt en $O(n^2)$. Qu'en est-il en réalité ?

Calculons le temps mis par notre machine pour effectuer ces calculs, ceci pour $n = 1000, 2000, \dots, 30000$.

In [25]:
```
import timeit
```

Attention, l'évaluation de la cellule ci-dessous prend du temps. Allez prendre un café.

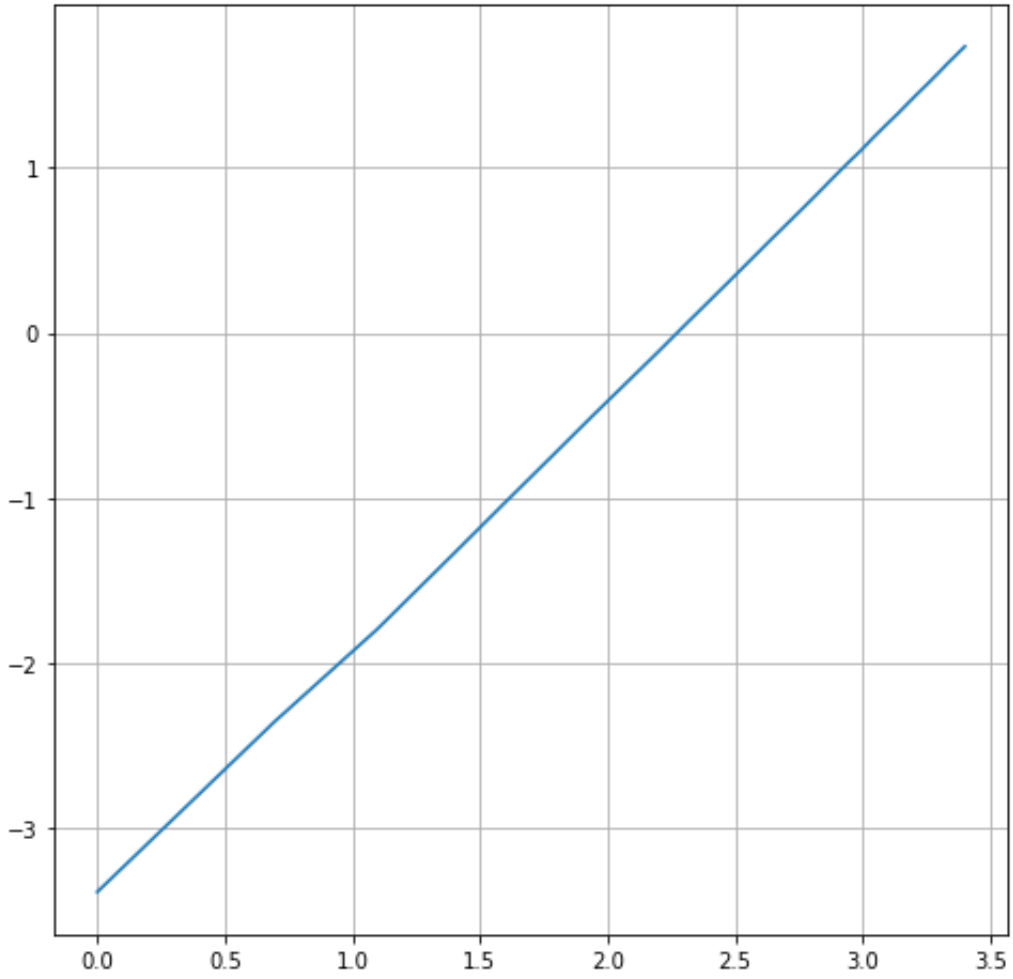
In [26]:
```
tms = [timeit.timeit('npart3(1000*%d)' % (n), setup='from __main__ imp
```

In [27]:
```
print(tms)
```

```
[0.033891445986228064, 0.09473188300034963, 0.16651452200312633, 0.2
581732029939303, 0.3638154990039766, 0.48053225698822644, 0.60876505
2005765, 0.7440559219976421, 0.8885975029988913, 1.0456503290042747,
1.2093359540012898, 1.3810918789968127, 1.5620597589877434, 1.751769
5459973766, 1.9443195380008547, 2.145373692997964, 2.35529177999706,
2.5778610589914024, 2.8010758870077552, 3.022614474000875, 3.2685118
970111944, 3.5098232549935346, 3.7490725929965265, 4.009198890998959
5, 4.270834879003814, 4.530098777991952, 4.80402107500413, 5.0922321
45994785, 5.370752525996068, 5.655364550999366]
```

Imaginons maintenant, en appelant t_n le temps mis pour calculer $P(1), \dots, P(n)$, que $t_n \sim Cn^\alpha$ où C et α sont deux réels strictement positifs ? C dépend évidemment de la machine choisie, mais que vaut α ? Remarquons que $\ln t_n \sim \ln C + \alpha \ln n \sim \alpha \ln n$, le graphe de t_n en fonction de n ressemblera donc fort à une droite.

In [28]:
```
plt.plot([log(k) for k in range(1, 31)], [log(t) for t in tms])
plt.grid()
```



Effectivement, cela ressemble fort à une droite. Quelle est sa pente, α ?

In [29]:
```
(log(tms[29])-log(tms[10])) / (log(30000) - log(11000))
```

Out[29]:
```
1.5374563140023387
```

On obtient $\alpha \simeq \frac{3}{2}$. Surprenant ... que peut-on en conclure ? Que les effets "grands entiers" ne se sont pas encore fait sentir ? Si nous étions courageux, nous calculerions les temps mis par notre fonction pour des entiers n plus grands. Il y a fort à parier que notre "constante" α serait plus grande que $\frac{3}{2}$ ...

Laissons tout cela au conditionnel, c'est l'heure de l'apéro :-).

3.5 Peut-on faire encore mieux ?

Oui, par exemple en utilisant des transformées de Fourier, mais ceci est une autre histoire que je n'aborderai pas ici. On peut calculer les $P(k)$, $1 \leq k \leq n$ avec une complexité en $O(n^{\frac{3}{2}} \log^2 n)$ opérations élémentaires. Pour $n = 1000000$ et le logarithme décimal, la quantité dans le grand O est égale à 36 milliards, un nombre d'opérations grand mais pas inatteignable. On peut également paralléliser les calculs, c'est à dire utiliser plusieurs machines qui effectuent chacune une partie des calculs, puis recombiner le tout. De telles techniques permettent d'atteindre la valeur de $P(n)$ pour des valeurs de n de l'ordre ... du milliard.

4. Un équivalent de $P(n)$

4.1 L'équivalent de Hardy et Ramanujan

J'ai annoncé au début de ce notebook que $P(n)$ croît "exponentiellement" avec n. Qu'en est-il exactement ? On possède un équivalent de $P(n)$ lorsque n tend vers l'infini. Ce résultat est dû à G.H. Hardy et S. Ramanujan. Le voici sans preuve :

Théorème : $P(n) \sim \frac{1}{4n\sqrt{3}} \exp(\pi \sqrt{\frac{2n}{3}})$

Remarque : Ce n'est pas la première fois que nous rencontrons $\sqrt{\frac{2n}{3}}$ ...

Remarque : Cet équivalent est le premier terme d'un développement asymptotique de $P(n)$ (Hardy, Ramanujan, Rademacher) dont le reste tend vers 0 lorsque le nombre de termes du développement tend vers l'infini. On dispose de plus d'un majorant de l'erreur commise. En utilisant ce développement asymptotique on peut donc calculer $P(n)$ de façon exacte puisque ce nombre est ... un entier ! Des algorithmes efficaces de calcul de $P(n)$ ont été développés par cette technique.

Exercice : Que signifient le G.H. et le S. de G.H. Hardy et S. Ramanujan ?

La fonction `approx_rama` calcule cet équivalent.

In [30]:
```
def approx_rama(n):
    return 1 / (4 * n * sqrt(3)) * exp(pi * sqrt(2 * n / 3))
```

Prenons par exemple $n = 200$.

In [31]:
```
approx_rama(200)
```

Out[31]:
```
4100251432187.85
```

In [32]:
```
float(npart3(200))
```

Out[32]:
```
3972999029388.0
```

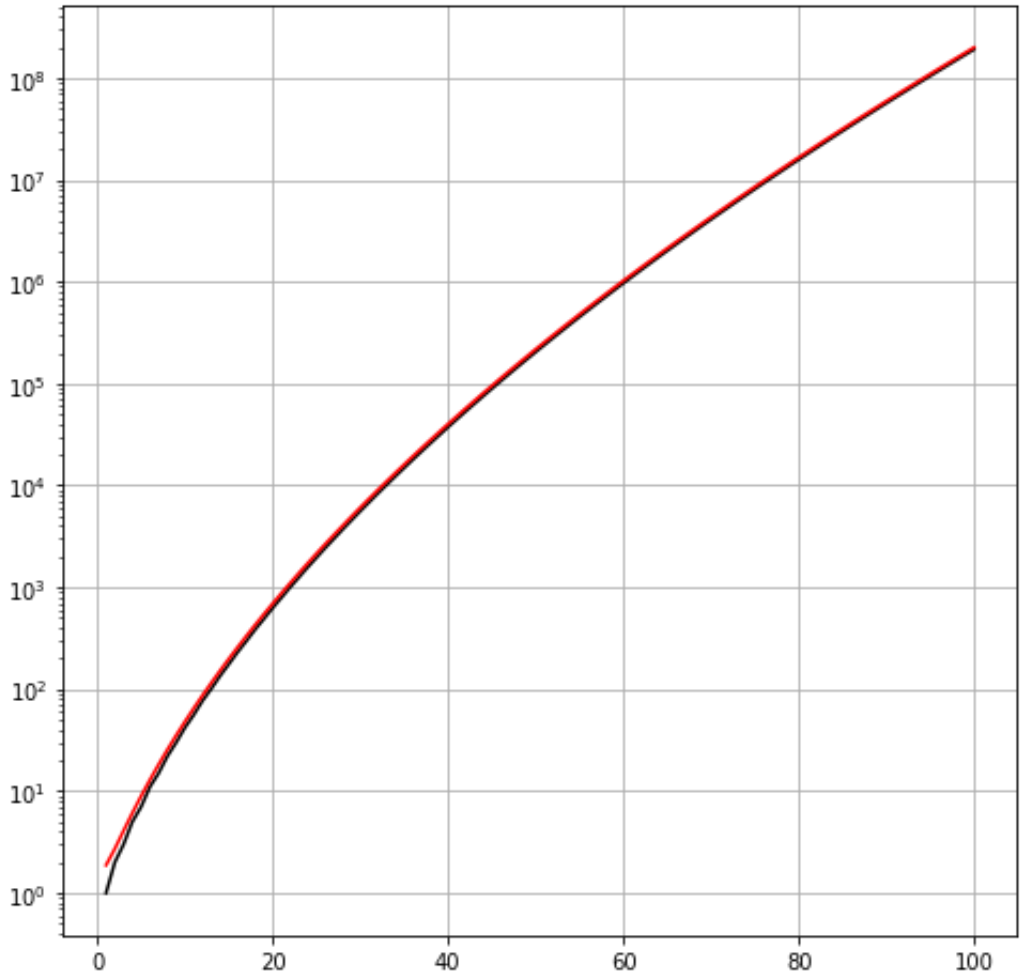
In [33]:
```
approx_rama(200) / npart3(200)
```

Out[33]:
```
1.0320293062894235
```

L'approximation de Hardy et Ramanujan nous donne donc $P(200)$ avec une erreur de l'ordre de 3%, ce qui est remarquable. Si vous ne trouvez pas cela remarquable, refaite l'exercice "sur les pas de Macmahon".

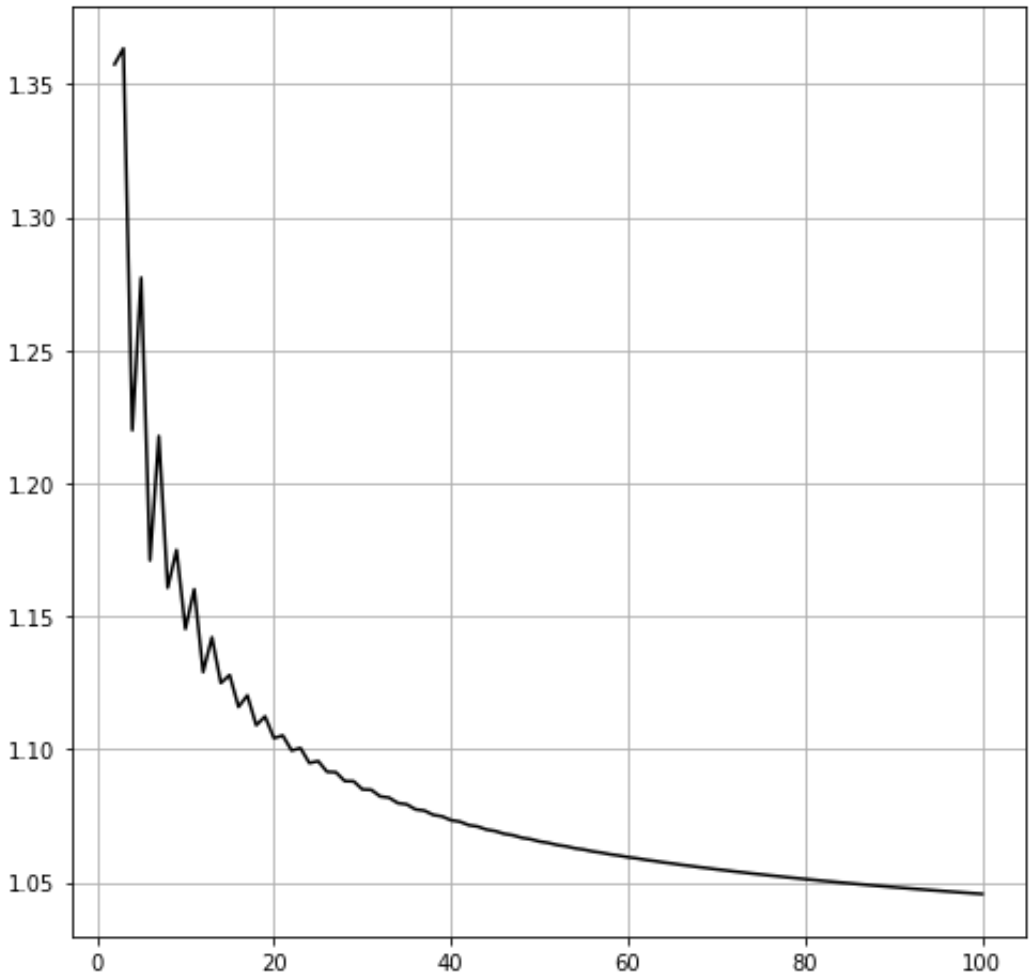
Traçons sur un même graphe $P(n)$ et son approximation.

In [34]:
```
n = 100
rg = range(1, n + 1)
P = npart3(n, mat=True)
Q = [approx_rama(k) for k in rg]
plt.semilogy(rg, P[1:],'k')
plt.semilogy(rg, Q, 'r')
plt.grid()
```



Les deux courbes sont quasi-superposées. Autre vision : le tracé de l'erreur relative.

In [35]:
```
n = 100
rg = range(2, n + 1)
P = npart3(n, mat=True)
Q = [approx_rama(k) / P[k] for k in rg]
plt.plot(rg, Q, 'k')
plt.grid()
```



4.2 La taille de $P(n)$

Soit N un entier non nul. Le nombre de chiffres en base 10 de l'entier n est $\lfloor \log n \rfloor + 1$, où $\log$ désigne le logarithme en base 10.

In [36]:
```
def taille(n): return floor(log(n) / log(10)) + 1
```

In [37]:
```
taille(12345)
```

Out[37]: 5

Il est donc immédiat d'avoir le nombre de chiffres de $P(n)$ lorsqu'on peut calculer $P(n)$.

In [38]:
```
taille(npart3(10000))
```

Out[38]: 107

In [39]:
```
def approx_taille(n):
    return floor((pi * sqrt(2 * n / 3) - log(4 * n * sqrt(3))) / log(1(
```

In [40]:
```
approx_taille(10000)
```

Out[40]: 107

Quel est (environ, ou peut-être exactement, qui sait ?) le nombre de chiffres de $P(1000000)$? De $P(1000000000)$?

In [41]:
```
approx_taille(10 ** 6)
```

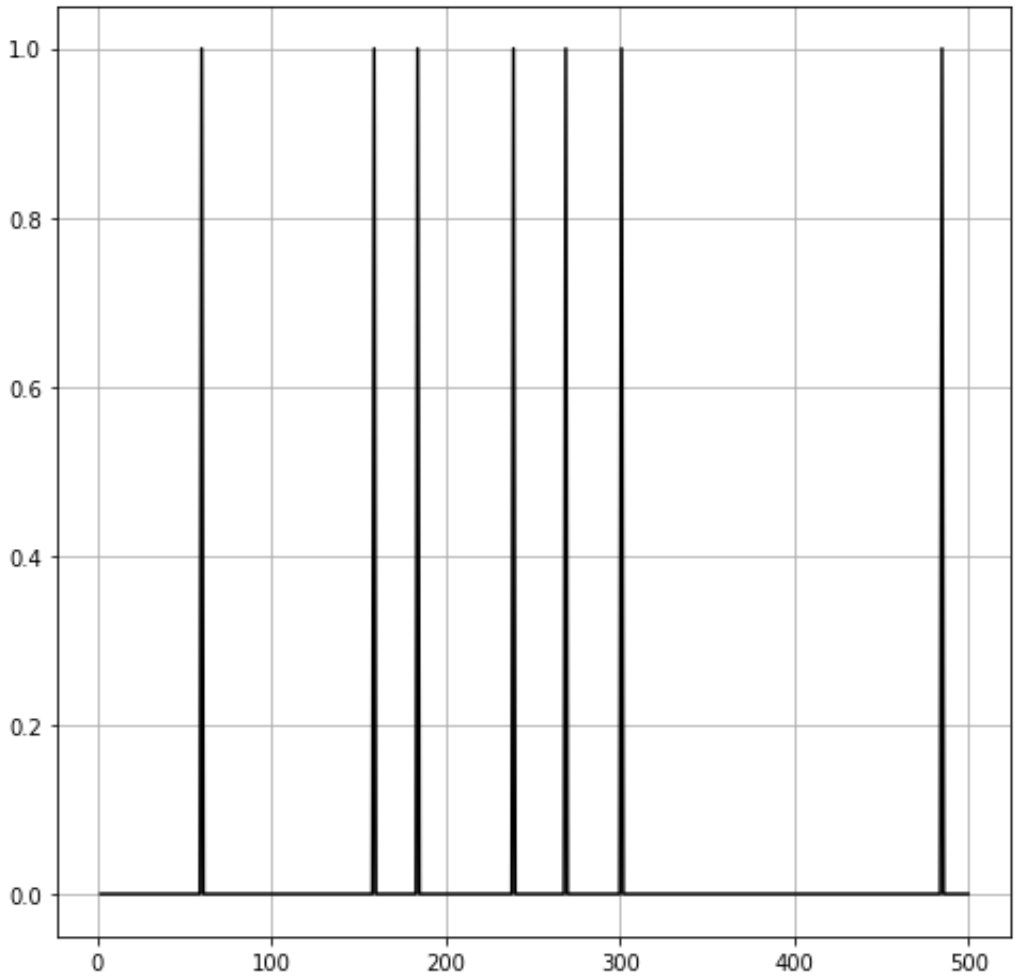
Out[41]: 1108

In [42]:
```
approx_taille(10 ** 9)
```

Out[42]: 35219

Notons $\varphi(n) = \frac{1}{4n\sqrt{3}}\exp(\pi\sqrt{\frac{2n}{3}})$. On a $\frac{P(n)}{\varphi(n)} \to 1$ lorsque n tend vers l'infini. En passant aux logarithmes, il vient donc que $\log P(n) - \log \varphi(n) \to 0$ lorsque n tend vers l'infini. La fonction `approx_taille` renvoie donc la valeur à peu près exacte (à 1 près) du nombre de chiffres de $P(n)$, pour tout n assez grand.

In [43]:
```
n = 500
rg = range(2, n + 1)
P = npart3(n, mat=True)
Q = [taille(approx_rama(k)) - taille(P[k]) for k in rg]
plt.plot(rg, Q, 'k')
plt.grid()
```



La taille de `approx_rama(n)` serait-elle la même que celle de $P(n)$ à 1 près, pour **tout** n ? Pourquoi ces pics par ci par là ? Il y en a un, par exemple, pour $n = 60$.

In [44]:
```
npart3(60)
```

Out[44]: 966467

In [45]:
```
approx_rama(60)
```

Out[45]: 1024004.4844484156

Ah ben oui. $P(60)$ est tout juste inférieur à une puissance de 10. Je n'irai pas plus loin, bien conscient de n'avoir fait qu'effleurer un sujet dont l'étendue est monumentale.

Exercice : il y a 7 pics sur le graphe ci-dessous. Le premier d'entre eux se situe à $n = 60$. Trouvez la position exacte des 6 autres.

5. Références

- The Man who Knew Infinity (2015). Réalisateur : Matt Brown. Acteurs : Dev Patel, Jeremy Irons.
- An Introduction to the Theory of Numbers, G.H. Hardy et E.M. Wright (6ème édition, 2008). Chapitre XIX : Partitions.

Et, bien entendu, une recherche sur Internet avec les mots clés "integer partition".

In [ ]: