Calcul de $\sum_{i,j\leqslant k} \mathrm{PPCM}(i,j)$

Jean-François Burnol

Jeudi 12 novembre 2020

Je remercie Hubert L., brillant élève de lycée, car ce texte est directement issu de discussions avec lui début novembre 2020.

Ce texte se veut introductif ; idéalement le lecteur aura déjà un peu travaillé avec les fonctions arithmétiques (on dit certaines choses simples mais importantes sans preuves), mais on partira presque de rien et on évitera d'aller trop loin dans les approfondissements, par exemple je ne parlerai pas ou à peine des coefficients de Möbius.

Par convention, et sauf mention exprès du contraire, toute somme indicée par un entier, par exemple i, débute à $i = 1$. Souvent, on a des conditions du genre $i \leqslant x$, et il est commode d'autoriser à x d'être un nombre réel. Par exemple on écrira $\sum_{i\leqslant x/2}$ même si $x/2$ n'est pas entier, et c'est la même chose que $\sum_{i\leqslant \lfloor x/2\rfloor}$ où la limite supérieure est explicitement un nombre entier. J'utilise $\lfloor x\rfloor$ pour la fonction partie entière au lieu de la notation usuelle $[x]$ (par contre pour rester avec des caractères ascii, je peux utiliser $[x]$ dans des commentaires de code).

Dans les codes Python par contre, x sera entier, et le code utilisera $x//3$ par exemple là où un commentaire de code écrira peut-être $x/3$, ou $[x/3]$. À propos pour $x \in \mathbb{R}$ et j, k deux entiers strictement positifs alors $\lfloor\lfloor x/j\rfloor/k\rfloor = \lfloor x/jk\rfloor$, ou encore en se limitant à $x \in \mathbb{Z}$ et avec les notation de Python $(x//j)//k = x//(j*k)$ (on rappelle qu'en Python $x//j$ est le quotient euclidien, c'est-à-dire $[x/j]$, même pour $x < 0$, lorsque $j > 0$).

Démonstration : pour $n \in \mathbb{Z}$, $n \leqslant \lfloor x/jk\rfloor \iff n \leqslant \frac{x}{jk} \iff nk \leqslant \frac{x}{j} \iff nk \leqslant \lfloor\frac{x}{j}\rfloor \iff n \leqslant k^{-1}\lfloor\frac{x}{j}\rfloor \iff n \leqslant \lfloor\frac{1}{k}\lfloor\frac{x}{j}\rfloor\rfloor$. Donc les deux entiers $\lfloor x/jk\rfloor$ et $\lfloor\lfloor x/j\rfloor/k\rfloor$ sont identiques.

1 Fonctions multiplicatives, convolution

Une fonction arithmétique $f : \mathbb{N}^* \to \mathbb{C}$ est dite multiplicative si $f(ab) = f(a)f(b)$ lorsque $(a,b) = 1$. On utilisera (a,b) comme notation alternative à $\mathrm{PGCD}(a,b)$. Alors $f(1)$ vaut 0 ou 1 et si $f(1) = 0$, f est la fonction nulle.

Si f et g sont deux fonctions sur $\mathbb{N}^*$ à valeurs complexes, leur convolution multiplicative est définie par $(f*g)(n) = \sum_{d|n} f(d)g(\frac{n}{d})$. Par convention une somme sur les diviseurs $(d|n)$ est restreinte aux diviseurs positifs (strictement, car de toute façon on a $n \geqslant 1$).

La convolution est associative. Si f et g sont multiplicatives alors $f*g$ aussi : en effet si $(a,b) = 1$ alors les diviseurs (positifs) de $n = ab$ sont exactement les produits d_1d_2 où d_1 et d_2 parcourent les diviseurs respectivement de a et de b (comme on peut voir par exemple avec la factorisation en produit de nombres premiers). La propriété de $f*g$ en découle.

Parmi les fonctions multiplicatives célèbres, il y a la fonction constante $\mathbf{1}$, la fonction « nombre de diviseurs » $\tau = \mathbf{1}*\mathbf{1}$, la fonction $\mathrm{Id} : n \mapsto n$, la fonction « somme des diviseurs » $\sigma = \mathbf{1}*\mathrm{Id}$. Notons aussi que si f est multiplicative alors la fonction produit $\mathrm{Id}\cdot f$ l'est également.

Enfin, citons l'indicatrice d'Euler φ comme exemple important de fonction multiplicative (par le théorème des restes chinois par exemple).

Un éclairage est apporté par l'idée des séries de Dirichlet $\sum_n f(n)n^{-s}$, car la convolution multiplicative correspond à la simple multiplication des séries de Dirichlet (formelle... mais si f a croissance au plus polynomiale, alors ce sont aussi des fonctions de la variable complexe de partie réelle suffisamment grande). Par exemple l'indicatrice d'Euler vérifie les identités $\sum_{d|n} \varphi(d) = n$, ce qui se traduit par $\mathbf{1} * \varphi = \mathrm{Id}$, puis par $\zeta(s)\mathrm{L}_\varphi(s) = \zeta(s-1)$ mais nous n'utiliserons pas ces notions ici.

2 Fonctions sommatoires, « principe de Dirichlet »

Souvent on s'intéresse à des fonctions sommatoires $\mathrm{F}(x) = \sum_{n\leqslant x} f(n)$. Si on peut écrire f sous la forme d'une convolution multiplicative $g * h$, alors, on peut rattacher F aux fonctions sommatoires respectives G et H de g et h.

En effet $\mathrm{F}(x) = \sum_{n\leqslant x}\sum_{d|n} g(d)h(n/d) = \sum_{i\cdot j\leqslant x} g(i)h(j)$. On va considérer i comme la coordonnée horizontale et j la coordonnées verticale. La somme porte sur des points entiers en-dessous d'une branche d'hyperbole.

Soit $q = \lfloor\sqrt{x}\rfloor$. Tout couple (i,j) d'entiers tels que $ij \leqslant x$ a soit $i \leqslant q$, soit $j \leqslant q$. Car sinon $ij \geqslant (q+1)^2 > x$, contradiction. Donc (idée de Dirichlet) :

$$\mathrm{F}(x) = \sum_{i\leqslant q, j\leqslant x/i} g(i)h(j) + \sum_{j\leqslant q, i\leqslant x/j} g(i)h(j) - \sum_{i,j\leqslant q} g(i)h(j) \tag{1}$$

La première somme compte la contribution des points sur les q premières verticales, la seconde celle des q premières horizontales, la dernière compense ce qui a été compté deux fois. Ainsi :

$$\mathrm{F}(x) = \sum_{i\leqslant q} g(i)\mathrm{H}(x/i) + \sum_{j\leqslant q} h(j)\mathrm{G}(x/j) - \mathrm{G}(q)\mathrm{H}(q) \tag{2}$$

Si G et H sont faciles à évaluer, ainsi bien sûr que g et h, alors on voit qu'on a ramené $\mathrm{F}(x)$ qui était une somme avec $\lfloor x\rfloor$ termes à une expression avec $2q+1$ termes. Naturellement ceci est utile pour des évaluations à la fois mathématiques et aussi numériques, puisqu'on passe de x à $\sqrt{x}$... attention cependant qu'il faut savoir évaluer les H et G en de grands nombres $x, x/2, \ldots$

Prenons comme exemple $f(n) = \tau(n)$, la fonction « nombre de diviseurs ». On a $\tau = \mathbf{1} * \mathbf{1}$, donc ici $\mathrm{G}(x) = \mathrm{H}(x) = \lfloor x\rfloor$, et on obtient

$$\mathrm{F}(x) = 2\sum_{i\leqslant q} \lfloor\frac{x}{i}\rfloor - q^2 \tag{3}$$

Ceci permet un équivalent asymptotique :

$$\mathrm{F}(x) = 2x\sum_{i\leqslant q}\frac{1}{i} + \mathrm{O}(q) + \mathrm{O}(q^2) = 2x\log q + \mathrm{O}(x) = x\log x + \mathrm{O}(x) \sim x\log x \tag{4}$$

Ce qu'on peut résumer en disant qu'un entier $\mathrm{N} \leqslant x$ a en moyenne $\log x$ diviseurs... (au sens où $\mathrm{F}(x)/x \sim \log x$).

Ici, on peut procéder un peu différemment, car $\mathrm{G}(x) = \mathrm{H}(x)$ est si simple. On sommant par les verticales on a d'une manière générale

$$\mathrm{F}(x) = g(1)\mathrm{H}(x) + g(2)\mathrm{H}(\frac{x}{2}) + g(3)\mathrm{H}(\frac{x}{3}) + \ldots \tag{5}$$

donc ici $F(x) = \sum_{i \leqslant x} \lfloor \frac{x}{i} \rfloor = x \sum_{i \leqslant x} \frac{1}{i} + O(x) = x \log x + O(x)$, confirmant notre résultat mathématique.

Mais si l'on veut la valeur numérique exacte, cette méthode a un coût $O(x)$ [1] tandis que la formule plus subtile $2 \sum_{i \leqslant q} \lfloor \frac{x}{i} \rfloor - q^2$ ne coûte que $O(q) = O(\sqrt{x})$ (ici on a les O() au sens des informaticiens). Ceci étant dit en postulant pour simplifier que les divisions entières ont un coût indépendant de la taille de i et de x.

```
>>> 2*sum(100//i for i in range(1, 11))-10**2
482
>>> sum(100//i for i in range(1, 101))
482
>>> 2*sum(10000//i for i in range(1, 101))-100**2
93668
>>> sum(10000//i for i in range(1, 10001))
93668
>>> 2*sum(100000000//i for i in range(1, 10001))-10000**2
1857511568
>>> import timeit
>>> timeit.timeit(lambda: print(sum(100000000//i for i in range(1, 100000001))), number = 1)
1857511568
7.8242023670172784
```

On voit que sur un ordinateur à 2.8Ghz, il faut déjà presque 8 secondes pour $x = 10^8$ par l'approche naïve. Pour $x = 10^{10}$ il faudra sûrement au moins 100 fois plus de temps, donc plus de 13 minutes. Par contre [2]

```
>>> timeit.timeit(\
... lambda: print(2*sum(10000000000//i for i in range(1, 100001)) - 10**10),\
... number = 1)
231802823220
0.014943361980840564
```

et donc le résultat 231802823220 est obtenu en moins de 2 centièmes de seconde par la méthode de Dirichlet. Et au passage on peut constater que l'équivalent asymptotique en $x \log x$ est déjà assez bien vérifié :

```
>>> import math
>>> 231802823220/(10**10*10*math.log(10))
1.0067068701404096
>>> timeit.timeit(\
... lambda: print(2*sum(1000000000000//i for i in range(1, 1000001)) - 10**12),\
... number = 1)
27785452449086
0.13982620899332687
>>> 27785452449086/(10**12*12*math.log(10))
1.0055890563186036
```

À propos, vu que les processeurs sont cadencés avec une fréquence de quelques giga-hertz, faire des sommes avec un milliard de termes est déjà à la limite de ce qui est raisonnable, car prendra de l'ordre de la seconde au bas mot, et de la seconde à la minute, puis de la minute à l'heure, puis de l'heure au jour, il n'y a pas une marge si immense. Dans le dernier exemple ci-dessus on

1. en considérant que toute opération arithmétique est $O(1)$.
2. On a utilisé dans l'interpréteur Python le caractère de continuation de ligne \ pour pouvoir imprimer ici la session interactive sans déborder dans la marge.

a fait 1 million d'opérations arithmétiques et une somme avec 1 million de termes et l'exécution du code a nécessité déjà plus de un dixième de seconde.

3 Application au calcul numérique de $\sum_{i,j\leqslant x} \mathrm{PPCM}(i,j)$

Notre objectif est de calculer numériquement

$$\mathrm{A}(x) = \sum_{i,j\leqslant x} \mathrm{PPCM}(i,j) \tag{6}$$

pour des x assez grands, disons avec 14 chiffres, et bien que l'on puisse chercher à vous rendre fou en vous demandant d'y arriver en moins de 5 secondes, on est prêt finalement à laisser tourner l'ordinateur pendant une nuit si besoin.

Inutile de dire que l'approche naïve qui consiste à calculer environ 10^{28} plus petits communs multiples et à les additionner est totalement irréalisable sur ordinateur personnel. Nous travaillons un peu sur la double-somme pour faire apparaître une quantité auxiliaire $\mathrm{S}(x)$ qui sera notre point focal.

$$\mathrm{A}(x) = \sum_{i,j\leqslant x} \frac{i\cdot j}{\mathrm{PGCD}(i,j)} = \sum_{d} \frac{1}{d} \sum_{\substack{i,j\leqslant x\\(i,j)=d}} ij \tag{7}$$

Maintenant rappelons que $\mathrm{PGCD}(i,j) = d$ si et seulement si $i = di', j = dj', \mathrm{PGCD}(i',j') = 1$. Donc

$$\mathrm{A}(x) = \mathrm{S}(x) + 2\mathrm{S}(\frac{x}{2}) + 3\mathrm{S}(\frac{x}{3}) + 4\mathrm{S}(\frac{x}{4}) + \ldots\ldots = \sum_{d\leqslant x} d\mathrm{S}(\frac{x}{d}) \tag{8}$$

avec

$$\mathrm{S}(x) = \sum_{\substack{i,j\leqslant x\\(i,j)=1}} ij \tag{9}$$

Notons $w(m) = \sum_{\substack{\max(i,j)=m\\(i,j)=1}} ij$ de sorte que

$$\mathrm{S}(x) = \sum_{m\leqslant x} w(m) \tag{10}$$

La formule (8) devient

$$\mathrm{A}(x) = \sum_{d\leqslant x} \sum_{m\leqslant x d^{-1}} dw(m) = \sum_{m\cdot d\leqslant x} dw(m) \tag{11}$$

On s'est ramené à sommer une fonction $g(m)h(d)$ pour les points entiers (m,d) en dessous de l'hyperbole $m\cdot d = x$!

Pour l'instant la suite $(w(m))$ est non explicite, mais en tout cas en notant $\mathrm{U}(x)$ la fonction sommatoire de la fonction Id, qui est bien connue :

$$\mathrm{U}(x) = \sum_{d\leqslant x} d = \frac{\lfloor x\rfloor(\lfloor x\rfloor+1)}{2}\ , \tag{12}$$

la formule générale de Dirichlet (2) nous donne :

$$\mathrm{A}(x) = \sum_{d\leqslant q} d\mathrm{S}(\frac{x}{d}) + \sum_{m\leqslant q} w(m)\mathrm{U}(\frac{x}{m}) - \mathrm{S}(q)\mathrm{U}(q) \tag{13}$$

Pour l'instant on navigue un peu à vue, rendons les choses plus concrètes en donnant une formule explicite pour $w(m)$. Calculons pour $m > 1$:

$$\begin{aligned} w(m) &= \sum_{\substack{\max(i,j)=m\\(i,j)=1}} ij \\ &= \sum_{j<m,(j,m)=1} mj + \sum_{i<m,(i,m)=1} im \\ &= 2\sum_{i<m,(i,m)=1} im \\ &= \sum_{i<m,(i,m)=1} im + \sum_{i<m,(i,m)=1} (m-i)m \\ &= m^2 \sum_{i<m,(i,m)=1} 1 = m^2\varphi(m) \end{aligned} \tag{14}$$

On a utilisé que $(i,m) = 1 \iff (m-i,m) = 1$ et $0 < i < m \iff 0 < m-i < m$. Le résultat vaut aussi pour $m = 1$. Donc, parfait, notre w est une fonction arithmétique assez banale, c'est $w = \mathrm{Id}^2 \cdot \varphi$.

Et en particulier w est une fonction arithmétique multiplicative, mais nous n'allons pas du tout exploiter cette information ! Et on note au passage qu'en posant $a(n) = \mathrm{A}(n) - \mathrm{A}(n-1) = \sum_{\max(i,j)=n} \mathrm{PPCM}(i,j)$ on a la formule exacte :

$$a(n) = \mathrm{A}(n) - \mathrm{A}(n-1) = \sum_{m\cdot d=n} dm^2\varphi(m) = \sum_{m|n} \frac{n}{m} m^2\varphi(m) = n\sum_{m|n} m\varphi(m) \tag{15}$$

Donc avec les notations de convolution multiplicative : $a = \mathrm{Id}\cdot(\mathbf{1} * (\mathrm{Id}\cdot\varphi))$, d'où :

Théorème. *$a(n) = \sum_{\max(i,j)=n} \mathrm{PPCM}(i,j)$ est une fonction multiplicative de n.*

Pour information, $a(p^e) = p^e \frac{p^{2e+1}+1}{p+1}$, pour p premier (simple calcul à partir de (15)).

Mais comme indiqué précédemment, nous n'utiliserons en fait pas la propriété de multiplicativité de $w(m) = m^2\varphi(m)$ ni celle de $a(n) = n\sum_{m|n} m\varphi(m)$.

Signalons juste rapidement au passage que la série de Dirichlet $\sum a(n)n^{-s}$ est $\frac{\zeta(s-1)\zeta(s-3)}{\zeta(s-2)}$ et que son résidu au pôle le plus à droite en $s = 4$ est $\zeta(3)/\zeta(2)$ que l'on retrouvera dans la section sur la formule asymptotique $\zeta(3)\zeta(2)^{-1}x^4/4$ de Bordellès pour $\mathrm{A}(x) = \sum_{n\leqslant x} a(n)$.

Revenant à (13), il ne nous reste plus pour évaluer $\mathrm{A}(x)$ qu'à :

1. évaluer (efficacement) tous les $w(m) = m^2\varphi(m)$, pour $m \leqslant q$,
2. évaluer (aussi bien que possible) $\mathrm{S}(x)$, $\mathrm{S}(x/2)$, …, $\mathrm{S}(x/q)$ et $\mathrm{S}(q)$, avec S la fonction sommatoire des coefficients $w(m)$.

Le premier point s'attaque par une méthode élégante à la Eratosthène. Mais le problème maintenant c'est qu'on voit pas comment on va faire le deuxième point !

4 Une identité récursive pour les $S(x) = \sum_{i,j \leqslant x,(i,j)=1} ij$

L'idée est très simple : on fait apparaître les $S(x), S(x/2), \ldots$, dans le calcul de $\sum_{i,j \leqslant x} ij = U(x)^2$. Je formule sous la forme d'un théorème :

Théorème (preuve laissée en exercice)**.**

$$U(x)^2 = S(x) + 4S(\frac{x}{2}) + 9S(\frac{x}{3}) + \cdots = \sum_d d^2 S(\frac{x}{d})$$

Donc, si on connaît les $S(y)$ pour $y < x$ on connaît $S(x)$. En fait il suffit de connaître $S(y)$ pour $y = \lfloor x/2 \rfloor, \lfloor x/3 \rfloor, \ldots, 1$. Or pour $j > q$, on a $j \geqslant q+1$ donc $x/j < q+1$, donc $y = x_j = \lfloor x/j \rfloor \leqslant q$. Il y a donc au plus $2q$ entiers $x_j \geqslant 1$ de cette forme (en fait on peut montrer qu'il y en a exactement $2q - 1$ ou $2q$, et q, $q - 1$, ..., 1 y sont tous inclus automatiquement).

Et, comme il a déjà été signalé que $\lfloor \lfloor x/j \rfloor / k \rfloor = \lfloor x/jk \rfloor$, cela ouvre la voie à une approche récursive pour l'évaluation de au plus $2q$ quantités : $S((x_j)_k) = S(x_{jk})$.

Mais, à y regarder de plus près, lorsque j est grand, on peut aussi bien utiliser directement la définition $S(x_j) = \sum_{m \leqslant x_j} m^2 \varphi(m)$ puisque x_j sera petit. Et mieux encore, ce ne sont pas les $S(x_j)$ per se qui nous intéressent mais uniquement leur contribution totale à la somme $\sum_d d^2 S(\frac{x}{d})$. Et on va à nouveau donc aborder ce problème via le Principe de l'Hyperbole.

On fait comme précédemment :

$$U(x)^2 = \sum_d d^2 S(\frac{x}{d}) = \sum_d d^2 \sum_{m \leqslant x d^{-1}} w(m) = \sum_{md \leqslant x} d^2 w(m) \tag{16}$$

$$U(x)^2 = \sum_{d \leqslant q} d^2 S(\frac{x}{d}) + \sum_{m \leqslant q} w(m) V(\frac{x}{m}) - S(q)V(q) \tag{17}$$

Dans cette formule, la fonction sommaire $V(x)$ remplace le $U(x)$ de (13)

$$V(x) = \sum_{d \leqslant x} d^2 = \frac{\lfloor x \rfloor (\lfloor x \rfloor + 1)(2\lfloor x \rfloor + 1)}{6} \,. \tag{18}$$

Conclusion : si nous connaissons les $S(\lfloor x/j \rfloor)$ pour $2 \leqslant j \leqslant q$ et $S(q)$ alors par (17) nous connaissons $S(x)$. Ceci au prix d'avoir calculé tous les $w(m)$, $m \leqslant q$, et de faire ensuite une soustraction à $U(x)^2$ de $2q - 1$ quantités et l'addition de $S(q)V(q)$.

Comme une fois qu'on a tous les $w(m)$, $m \leqslant q$, ça ne coûte que $O(q)$ d'obtenir tous les $S(m)$, $m \leqslant q$ qui sont leurs sommes successives, j'ai décidé de faire ce calcul, même si peut-être on calcule trop de $S(m)$ par rapport ce qui est nécessaire.

5 Algorithme

Comme dans la section suivante je donne une implémentation en Python, je vais juste décrire informellement la structure sous-jacente de l'algorithme pour la calcul de $A(x)$, comme une succession d'étapes.

1. On calcule $q = \lfloor \sqrt{x} \rfloor$.
2. On calcule tous les $w(m) = m^2\varphi(m)$ pour $m \leqslant q$ ainsi que leurs sommes $\mathrm{S}(m)$, $m \leqslant q$.
3. Si $\lfloor x/q \rfloor > q$, c'est-à-dire si $q(q+1) \leqslant x$, on pose $j = q$. Sinon alors en fait $\lfloor x/q \rfloor = q$. Dans ce cas on pose $j = q - 1$.
4. Si j est nul on s'arrête. Sinon on pose $y = \lfloor x/j \rfloor$ et $q_y = \lfloor \sqrt{y} \rfloor$. On utilise la formule (17) avec x remplacé par y et q par q_y. On connaît toutes les valeurs de S sauf $\mathrm{S}(y)$ que l'on peut donc déduire. Ceci coûte de l'ordre de $\mathrm{O}(q_y)$.
5. On décrémente j de une unité et on boucle vers l'étape précédente.
6. À ce stade on connaît les $\mathrm{S}(\lfloor x/j \rfloor)$ pour $j \leqslant q$ ainsi que $\mathrm{S}(q)$ (dès la deuxième étape on a calculé tous les $\mathrm{S}(m)$, $m \leqslant q$). On utilise la formule (13) pour obtenir finalement $\mathrm{A}(x)$.

On peut estimer le coût en temps de cet algorithme comme étant $\mathrm{O}(\sum_{1 \leqslant j \leqslant q} \sqrt{\frac{x}{j}}) = \mathrm{O}(\sqrt{x}\sqrt{q}) = \mathrm{O}(x^{0,75})$ (les autres opérations que la boucle sur j coûtant plutôt de l'ordre de $\mathrm{O}(q \log\log q)$). Son coût en mémoire est $\mathrm{O}(x^{0,5})$.

Pour $x = 10^{14}$, $x^{0,75} \approx 31{,}6 \cdot 10^9$. Pour « un milliard » d'opérations arithmétiques comme des multiplications, des divisions etc..., on s'attend à une durée de l'ordre de la minute au minimum :

```
>>> def foo(x):
...     for y in range(x):
...         z = x*y
...
>>> timeit.timeit(lambda: foo(10**6), number = 1)
0.0719125850009732
>>> timeit.timeit(lambda: foo(10**7), number = 1)
0.7026002480124589
>>> timeit.timeit(lambda: foo(10**8), number = 1)
7.033876487024827
>>> timeit.timeit(lambda: foo(10**9), number = 1)
69.75429132900899
```

Ici on a environ 32 milliards d'opérations arithmétiques. Comme les choses sont plus compliquées que cela, avec des structures de dictionnaire ou de listes, et qu'on ne s'obstine pas à optimiser le code privilégiant la lisibilité, il faut encore ajouter un facteur 5 ou 10 au minimum, donc on peut penser que pour $\mathrm{O}(31{,}6 \cdot 10^9)$ on pourra en quelques heures obtenir le résultat pour $x = 10^{14}$. Et c'est effectivement ce que l'on constate : avec une implémentation un peu plus optimisée (en particulier en faisant un peu d'inclusion-exclusion pour ne travailler qu'avec les j premiers avec 30), j'ai obtenu

$$\mathrm{A}(10^{14}) = 18\,269\,074\,235\,036\,192\,615\,433\,508\,173\,581\,108\,768\,530\,043\,596\,378\,896\,920$$

en environ six heures (avec un ordinateur équipé d'une puce à 2.8Ghz, je sais, mes moyens logistiques sont limités). Et avec le même code

$$\mathrm{A}(10^{10}) = 1\,826\,907\,423\,626\,873\,186\,205\,519\,018\,089\,134\,255\,148$$

est obtenu en un peu plus de 20 secondes, ce qui est effectivement environ 1000 fois plus rapide confirmant le comportement à peu près en $\mathrm{O}(x^{0,75})$ (en réalité, l'exposant semble plutôt être 0,76 sur mes observations de temps d'exécutions de cette implémentation particulière, mais peut-être faut-il y voir l'impact de faire des multiplications, divisions et sommes sur des entiers plus grands). À propos le résultat numérique a à peu près été multiplié par 10^{16} pour x ayant été multiplié par 10^4 ce qui suggère un comportement de $\mathrm{A}(x)$ en x^4, voir plus loin.

J'indique également que

$$A(10^{11}) = 18\ 269\ 074\ 235\ 277\ 524\ 758\ 737\ 296\ 596\ 165\ 569\ 110\ 162\ 456$$

est obtenu en un peu moins de deux minutes (comme on pouvait s'y attendre au vu des données précédentes) et que la valeur confirme une loi approchée en Cx^4.

Ces résultats ont été obtenu avec un code (un peu alambiqué) seulement 20% à 25% plus rapide que celui présenté dans la section suivante. En fait, il était une évolution d'un code antérieur moins bon que celui de la section suivante. Mais le fait d'avoir rédigé ce texte m'a donc amené à coder plus efficacement (ça va déjà beaucoup mieux lorsqu'on peut coder sans se préoccuper de commenter, puisque les commentaires ont déjà été faits...)

6 Implémentation en Python

On suit de près la description de la section précédente. On utilise des listes pour stocker des valeurs dépendant d'un $1 \leqslant m \leqslant q$ et des dictionnaires lorsqu'il s'agit de quantités (par exemple la somme des carrés des entiers inférieurs) dépendant des $\lfloor x/j \rfloor$, $1 \leqslant j \leqslant q$. On pourrait utiliser des dictionnaires pour tout, ce qui éviterait dans le code à certains endroits de se demander si on doit utiliser la liste ou le dictionnaire pour accéder aux données dont on a besoin, mais sans doute ça serait plus lent. On peut utiliser aussi des listes pour tout...

```
"""

:author: Jean-François Burnol
:date: 13 novembre 2020

Calcul la somme des PPCM(i,j) pour 1<= i, j <= x

Pour les explications voir sommesPPCM.pdf, document du même auteur, 12
novembre 2020 et la docstring de doublesommePPCM()

>>> doublesommePPCM(14)
8119

"""


def racinecarree(n):
    """Calcule la partie entière de la racine carrée.

    Fonctionne avec des entiers arbitrairement grands.
    L'entier n est supposé positif.

    >>> [racinecarree(x) for x in range(10)]
    [0, 1, 1, 1, 2, 2, 2, 2, 2, 3]
    >>> racinecarree(10000000)
    3162
    >>> racinecarree(10000000000000)
    3162277
    """
    if n < 2:
        return n
    # Comme valeur de départ de l'algorithme on prend
    # la plus petite puissance t de 2 avec t * t > n.
```

```
    # Accepte même n = 0 (alors t = 1) ou n = 1 (alors t = 2).
    t = 1 << ((n.bit_length() + 1) >> 1)
    # variante entière de l'algorithme babylonien
    q = n // t
    while q < t:
        t = (t + q) >> 1
        q = n // t
    return t


def listEulerPhi(N):
    """Calcule [0, phi(1), ...., phi(N)]

    phi(n) est la fonction indicatrice d'Euler (totient function)

    Idée du code : la fonction phi vérifie la formule

        phi(n) = n produit des (1 - 1/p) pour p diviseur premier de n

    En parcourant à la modeÉratosthène une liste d'entiers 2, 3, 4, ...
    on va repérer les nombres premiers les uns après les autres
    comme étant ceux qui ont conservé la valeur initiale sans modification,
    ensuite  on multiplie par (p-1) et divise par p pour les entiers n
    multiples de ce p.

    Il est important que cette procédure est en fait exacte, c'est-à-dire
    les divisions par p tombent toujours juste.

    Comme on trouve ainsi tous les p, on a donc ajusté phi(n) correctement
    pour tous les n.

    >>> listEulerPhi(19)
    [0, 1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]

    """
    L = [n for n in range(N+1)]
    # attention ici, il faut inclure la valeur maximale N dans cette boucle
    # (uniquement important si N est premier)
    for p in range(2, N + 1):
        if L[p] == p:
            # c'est donc que p est premier !
            # on ajuste pour p, 2p, etc..., jusqu'à N éventuellement
            for n in range(p, N + 1, p):
                L[n] = (L[n] * (p - 1)) // p
    return L


def doublesommePPCM(x):
    """Calcule A(x) = la somme des PPCM(i, j) pour 1<= i, j <= x

    C'est

        (*) S(x) + 2 * S(x/2) + 3 * S(x/3) + ...

    avec S(x) = somme des i * j pour i, j <= x premiers entre eux.

    On obtient la formule (*) en partitionnant la somme des PPCM(i,j) suivant
```

```
les valeurs des PGCD(i,j).

On calcule S(x) ainsi :

S(x) = 1 + somme des i*j pour i > j + somme des i*j pour j > i
     = 1 + 2 somme des i * j pour 1 <= j < i <= x, PGCD(i,j)=1
     = 1 + somme des (i * j + i * (j-i)) pour 1 <= j < i <= x, PGCD(i,j)=1
     = somme des i**2 * Phi(i) pour i <= x

On peut vérifier que

    (**) ([x]*([x]+1)//2)**2  = S(x) + 4 * S(x/4) + 9 * S(x/9) + ...

À gauche on a le carré de la (somme des i pour i <= x) qui est aussi la
(somme des cubes i**3 pour i<= x).

On prouve (**) par exemple en partitionnant la somme des produits i*j pour
1<= i, j<= x suivant les PGCD(i,j) = d, d = 1, ...., [x].

La formule (**) permet un calcul par récurrence de S(x) en fonction des
S(x/j) pour j = 2, ...

Dans cette récurrence on séparera les q premiers termes des suivants, avec
q la racine carrée entière de x. Après S(x//2), ...., S(x//q) on a des
S(m) pour m<=q. Le même "S(m)" est utilisé plusieurs fois dans (**), on
calcule la multiplicité qui est donc une somme de carrés sur une
progression arithmétique. Et les "S(m)", 1 <= m <= q ont été pré-calculés
via

    (***) S(m) = somme des (w(i) = i**2 * Phi(i)) pour i <= m

comme expliqué plus haut.

Pour les formules à la « Principe de l'hyperbole de Dirichlet » en
lesquelles (*) et (**) sont transformées et utilisées voir le document
sommesPPCM.pdf de J.-F. Burnol, 12 novembre 2020
"""
q = racinecarree(x)
# tout d'abord on va stocker certains quantités numériques
# utilisées plusieurs fois
listcarres = []
listU = []
listV = []
d2, s1, s2 = 1, 0, 0
for d in range(q + 1):
    d2 += (d << 1) - 1
    s1 += d
    s2 += d2
    listcarres.append(d2)
    listU.append(s1)
    listV.append(s2)
#
lesxj = [0]
lesxj.extend(x // j for j in range(1, q + 1))
# somme des i <= N est U(N) = N(N+1)/2
# on va stocker les U(N) pour N "grand" dans un dict
dictU = { xj: (xj * (xj + 1)) >> 1 for xj in lesxj}
```

```
# somme des i**2 pour i<=N est V(N) = N(N+1)(2N+1)/6
# on va stocker les V(N) pour N "grand" dans un dict
dictV = { xj: xj * (xj + 1) * ((xj << 1) + 1) // 6 for xj in lesxj}
# on calcule les coefficients w(m) = m**2 * phi(m) pour 1 <= m <= q
# ainsi que leur fonction sommatoire S(m)
listphi = listEulerPhi(q+1)
listS = []
listW = []
S = 0
for m in range(q + 1):
    w = listcarres[m] * listphi[m]
    listW.append(w)
    S += w
    listS.append(S)
#
# on aura besoin d'un dictionnaire de clés les xj pour stocker
# les valeurs des S(xj)
dictS = {}
#
if q*(q+1) <= x:
    J = q
else:
    J = q-1
    dictS[q] = listS[q]
# on s'arrête en j = 2 en partie pour ne pas calculer deux fois racinecarree(x)
# on fera j = 1 en dernier après la boucle.
# pour suivre, cf sommesPPCM.pdf, formules (17) et (13)
for j in range(J, 1, -1):
    y = lesxj[j]
    # valeur initiale pour S(y) = S(x//j)
    Sy = dictU[y]**2
    qy = racinecarree(y)
    My = q // j     # valeur maximale de m avec m*j <= q
    Sy -= dictV[y]  # contribution m=1 dans (17)
    jfoism = j
    for m in range(2, My + 1):
        jfoism += j
        Sy -= listcarres[m] * dictS[lesxj[jfoism]]
        Sy -= listW[m] * dictV[lesxj[jfoism]]
    for m in range(My + 1, qy + 1):
        ym = y//m  # on a fusionné les boucles pour ne le calculer qu'une fois...
        Sy -= listcarres[m] * listS[ym]
        Sy -= listW[m] * listV[ym]
    Sy += listS[qy] * listV[qy]
    # ENFIN ! on a donc la valeur de S(x//j) et on la stocke
    dictS[y] = Sy
# Faut pas oublier j = 1 maintenant
S = dictU[x]**2
S -= dictV[x]  # contribution m=1 dans (17)
for m in range(2, q + 1):
    S -= listW[m] * dictV[lesxj[m]]
    S -= listcarres[m] * dictS[lesxj[m]]
S += listS[q] * listV[q]
# On a donc la valeur de S(x) et on la stocke
dictS[x] = S
# Maintenant il s'agit de calculer S(x) + 2 S(x/2) + ...
# On suit la formule (13) du fichier sommesPPCM.pdf
```

```
    S = 0
    for m in range(1, q + 1):
        S += m * dictS[lesxj[m]]
        S += listW[m] * dictU[lesxj[m]]
    S -= listS[q] * listU[q]
    return S


# python doublesommePPCM.py exécute les tests
if __name__ == "__main__":
    import doctest
    doctest.testmod(optionflags=doctest.NORMALIZE_WHITESPACE |
                    doctest.ELLIPSIS,
                    # verbose=True
    )
```

7 Théorème de Bordellès

Théorème (Bordellès, 2007)**.** *On a l'équivalent asymptotique* $\mathrm{A}(x) \sim \mathrm{C}x^4$ *avec* $\mathrm{C} = \frac{\zeta(3)}{4\zeta(2)}$.

Numériquement C = 0,182 690 742 350 359 624 681 509 182 826 928 659 882 . . . .

Ce théorème (Journal of Integer Sequences, Vol. 10 (2007), Article 07.9.2) nécessite des techniques un peu plus avancées que celles du présent texte introductif.
http://www.cs.uwaterloo.ca/journals/JIS/VOL10/Bordelles2/bordelles61.html

Notre résultat numérique exact

$$\begin{aligned}\mathrm{A}(10^{14}) &= 18\ 269\ 074\ 235\ 036\ 192\ 615\ 433\ 508\ 173\ 581\ 108\ 768\ 530\ 043\ 596\ 378\ 896\ 920\\ &= 0{,}182\ 690\ 742\ 350\ 361\ 926\ 15\ldots\cdot 10^{56}\end{aligned}$$

lui est agréablement compatible. L'erreur constatée ici est de l'ordre de x^3, la formule de Bordellès donnant 13 chiffres exacts (quasiment 14) pour x ayant ici 15 (à peine plus que 14) chiffres.

8 Deux identités

Sans preuve j'indique deux formules récursives pour les $\mathrm{A}(x)$.

Théorème.

$$\sum_d d^2\mathrm{A}(\frac{x}{d}) = \sum_{m\leqslant x} m\sigma_2(m) = \sum_{u\cdot v\leqslant x} uv^3$$

Théorème.

$$\sum_d d\varphi(d)\mathrm{A}(\frac{x}{d}) = \sum_{m\leqslant x} m^3 = \sum_{i,j\leqslant x} ij = \mathrm{U}(x)^2$$

Dans la première formule, σ_2 est la fonction « somme des carrés des diviseurs (positifs) ». La première formule permet un algorithme relativement efficace, n'utilisant pas du tout l'indicatrice d'Euler, mais ma tentative a donné un résultat moins bon que celui présenté dans ce texte. La deuxième formule ne semble pas susceptible de donner un algorithme efficace, car il faut calculer des sommes longues avec l'indicatrice d'Euler. Mais justement les $\mathrm{S}(x)$ ont ce genre de structure et ce n'était pas évident de les calculer.