RETOUR TP 1 – PRISE EN MAIN DE Sage

On donne ici quelques détails mathématiques ainsi que quelques précisions ou commentaires concernant la fiche de TP 1 à lire en parallèle du Notebook Jupyter de correction du TP. Noter que `Sage`, malgré le fait que les objets aient des types différents, lorsqu'il manipule `a` dans $\mathbf{Z}$ et `b` dans $\mathbf{Z}/n\mathbf{Z}$, voit automatiquement `a` dans $\mathbf{Z}/n\mathbf{Z}$ si bien que si `a=2` et `b=2`, `Sage` renverra bien `True` pour `a==b` et sera capable de calculer `a+b`. De même, si `a` dans $\mathbf{Z}/n_1\mathbf{Z}$ et `b` dans $\mathbf{Z}/n_2\mathbf{Z}$, `Sage` voit automatiquement `a` et `b` dans $\mathbf{Z}/\text{pgcd}(n_1,n_2)\mathbf{Z}$ si bien que si `a=2` et `b=2`, `Sage` renverra bien `True` pour `a==b` et sera capable de calculer `a+b`. Un message d'erreur sera renvoyé en revanche dans le cas où $\text{pgcd}(n_1,n_2)=1$.

2 Boucles et fonctions

- Définir deux fonctions calculant la factorielle d'un entier n, d'une part en définissant une fonction récursive [1] et d'autre part en faisant une boucle.
  ▶ **CORRECTION.–** On rappelle qu'une fonction **itérative** est une fonction qui utilise une boucle tandis qu'une fonction **récursive** est une fonction faisant appel à elle-même. Ainsi, on peut coder simplement une fonction factorielle itérativement en utilisant la formule
  $$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$
  qui dit simplement qu'il faut prendre 1 puis le multiplier par 2, puis multiplier le résultat par 3 et ainsi de suite jusqu'à mutliplier le résultat par n. Cela donne lieu au code suivant :

```
def factorielle(n):
    x=1
    for i in range(2,n+1):
        x=x*i
    return x
```

  Mais l'on pouvait aussi utiliser le fait que
  $$n! = n \times (n-1)!$$
  et ainsi calculer factorielle n en disant qu'il s'agit de n fois factorielle de $n-1$. On calcule alors $(n-1)!$ en disant qu'il s'agit de $(n-1) \times (n-2)!$ et ainsi de suite. La seule chose **essentielle** ici est de préciser des conditions initiales pour que la descente de l'ordinateur s'arrête ! Par exemple, ici le procédé de descente ci-dessus s'arrête si l'on rajoute l'information que $0! = 1$. Cela conduit au code suivant :

```
def factorielle2(n):
    if n==0:
         return 1
    else:
        return n*factorielle2(n-1)
```

  Alors par exemple pour calculer `factorielle2(3)`, l'ordinateur procède comme suit

```
factorielle2(3)=3*factorielle2(2)
factorielle2(3)=3*(2*factorielle2(1))
factorielle2(3)=3*(2*(1*factorielle2(0)))
factorielle2(3)=3*(2*(1*1))=6
```

  Bien noter que sans la condition initiale `factorielle2(0)=1`, l'algorithme continuerait sans fin la suite d'instructions ci-dessus avec

```
factorielle2(0)=0*factorielle2(-1), factorielle2(-1)=-1*factorielle2(-2),...
```

  On **commencera donc toujours** une fonction récursive par **les conditions initiales**.
- On rappelle que la suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour tout $n \geqslant 2$. Définir une fonction `fibonacci` à nouveau de deux manières, la première à l'aide d'une boucle et la seconde à l'aide d'une fonction récursive. Que se passe-t-il si vous essayez de calculer F_{40} ? Pourquoi ? Recommencer en utilisant la commande `@cached_function`.
  ▶ **CORRECTION.–** On constate qu'avec la version récursive, `Sage` ne parvient pas à calculer F_{40} alors qu'avec la version itérative il y parvient sans peine. Cela est dû au fait que la version récursive de la fonction effectue un nombre exponentielle d'appels récursifs à elle-même (au cours desquels de nombreux calculs sont effectués plusieurs fois) et qu'à partir de $n = 40$, `Sage` n'arrive plus à gérer cela. L'option `@cached_function` permet à l'ordinateur de garder en mémoire cache (une mémoire temporaire plus facilement et rapidement accessible) certains calculs effectués, évitant donc tout un tas de calculs redondants lors des appels récursifs et permettant d'accélérer le processus [2]. On reviendra sur l'exemple de la suite de Fibonacci en section 4.

1. On rappelle qu'une fonction récursive est une fonction qui fait appel à elle-même.
2. Vous pouvez par exemple constater que maintenant la fonction récursive va plus vite que la version itérative !

3 Exponentiation rapide

- Afficher la liste des 3^n modulo 42 pour $n \in \{1, \dots, 50\}$. Que constatez-vous ? En déduire une autre manière de calculer $3^{2^{2^{2^2}}}$ et l'implémenter.

  ▶ **Correction.–** On conjecture que pour tout $k \in \mathbf{N}^*$, $x^{6+k} \equiv x^k \pmod{42}$ [3] si bien qu'il suffit de connaître l'exposant $2^{2^{2^2}}$ modulo 6 . On obtient grâce à `Sage` que $2^{2^{2^2}} \equiv 4 \pmod 6$ de sorte qu'en itérant la relation ci-dessus il vient

  $$x^{2^{2^{2^{2^2}}}} \equiv x^4 \equiv 39 \pmod{42}.$$

  Établissons alors notre conjecture. Soit k un entier naturel supérieur ou égal à 1. On cherche à établir que $x^{6+k} \equiv x^k \pmod{42}$, ce qui est équivalent à

  $$x^k \left(x^6 - 1\right) \equiv 0 \pmod{42} \quad \iff \quad x^{k-1} \left(x^6 - 1\right) \equiv 0 \pmod{14}$$

  car $42 = 3 \times 14$ et $k \geqslant 1$. Mais alors $x = 3$ et 14 sont premiers entre eux et donc le petit théorème de Fermat implique que $x^{\varphi(14)} \equiv 1 \pmod{14}$ et un calcul ou `Sage` fournit que $\varphi(14) = 6$ de sorte que $x^6 - 1 \equiv 0 \pmod{14}$ et on a le résultat !

- Définir de nouveau une fonction `fibonacci` en utilisant votre fonction d'exponentiation rapide appliquée à une matrice carrée de taille 2 et directement en utilisant l'expression exacte de F_n pour tout $n \in \mathbf{N}$. Que constatez-vous ?

  ▶ **Correction.–** Ici, on utilise le fait que si l'on pose

  $$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

  alors

  $$\forall n \in \mathbb{N}, \quad \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

  si bien qu'une récurrence fournit immédiatement que [4]

  $$\forall n \geqslant 0, \quad \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

  C'est cela qu'on implémente dans `fibonacci4`.
  On pouvait également bien sûr utiliser les formules de Binet (découlant du fait qu'on sait résoudre explicitement des récurrences linéaires d'ordre 2), à savoir que

  $$\forall n \in \mathbb{N}, \quad F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right).$$

  C'est ce qu'on utilise dans `fibonacci5`. On constate du fait que les nombres réels ont seulement une représentation approchée dans `Sage` et que prendre des puissances de ces représentations approchées peut accroître les erreurs commises, que les résultats obtenus par cette méthode ne sont pas entiers !

4 Comparaison des différentes versions de Fibonacci

- La version itérative

```
def fibonacci(n):
    a=0
    b=1
    for i in range(n):
        c=a
        a=b
        b=b+c
    return a
```

  conduit à n opérations sommes et donc à une complexité en $O(n)$ comme cela est confirmé expérimentalement ;

- La version récursive

3. Attention ici que cela ne découle directement pas du petit théorème de Fermat qui stipule que si a et n sont premiers entre eux, alors $a^{\varphi(n)} \equiv 1 \pmod n$ où φ est l'indicatrice d'Euler, `euler_phi()` en `Sage`. En effet, 3 n'est pas premier à 42 !

4. Autrement dit, on veut le coefficient $(1, 1)$ de A^n.

```
def fibonacci2(n):
    if (n==0):
        return 0
    elif (n==1):
        return 1
    else:
          return fibonacci2(n-1)+fibonacci2(n-2)
```

donne lieu à une complexité exponentielle. Notons C_n le nombre d'opérations nécessaires à l'exécution de `fibonaci2`. On a alors clairement que

$$\forall n \geqslant 2, \quad C_n = C_{n-1} + C_{n-2} + 1$$

avec $C_0 = C_1 = 1$ et on montre alors par une récurrence immédiate que $C_n = 2F_{n+1} - 1$ et les formules de Binet impliquent alors que

$$C_n = O\left(\varphi^n\right), \quad \text{avec} \quad \varphi = \frac{1+\sqrt{5}}{2}.$$

Cette méthode est donc bien plus longue! On peut la raccourcir en évitant le double appel récursif avec

```
def fibonacci2aux(n):
    if (n==0):
        return (0,1)
    else:
        (a,b)=fibonacci2aux(n-1)
        return (b,a+b)
def fibonacci2bis(n):
    return fibonacci2aux(n)[0]
```

Dans ce cas le calcul de F_{40} fonctionne parfaitement. Étudions alors la complexité de cet algorithme. On a cette fois avec les mêmes notations $C_{n+1} = C_n + 1$ soit $O(n)$, ce qui est là encore confirmé expérimentalement, cependant on voit qu'on est limité en espace à F_{900} environ tandis qu'en itératif, on peut aller (au moins) jusqu'à F_{7000}. On constate par ailleurs que la version récursive avec `@cached_function` est de loin la plus rapide mais on ne donnera pas sa complexité du fait que l'utilisation de cette fonction rend un peu plus obscur le détail de ce qu'il se passe lors de l'éxécution de l'algorithme.

- On peut alors utiliser la version avec la matrice A en section 3 et une exponentiation rapide.

```
def fibonacci4(n):
    A=Matrix([[1,1],[1,0]])
    F=vector([1,0])
    if (n==0):
        return 0
    elif (n==1):
        return 1
    else:
        A=exponentiation_rapide2(A,n-1)
        r=A*F
        return r[0]
```

Cette méthode semble plus rapide que l'itératif linéaire. En effet, en considérant que les affectations et extraire un coefficient d'un vecteur de taille 2 sont des opérations en temps constant, l'exponentiation rapide de A^{n-1} requiert $O(\log_2(n))$ multiplications matricielles et chacune de ces multiplications requiert (sous forme naïve) 4 additions et 8 multiplications ce qui donne toujours $O(\log_2(n))$, ce qui est sensiblement mieux que l'itératif, la différence étant expérimentalement significative à partir de $n \approx 2000$ d'après l'expérience.

- On donne ensuite une dernière version basée sur la formule de Binet et une exponentiation rapide dans $K = \mathbf{Q}(\sqrt{5})$.

```
def fibonacci6(n):
    r = 1/u*(exponentiation_rapide2((1+u)/2,n)-exponentiation_rapide2((1-u)/2,n))
    return r
```

On obtient de même une complexité en $O(\log_2(n))$ qui semble significativement plus rapide que l'itératif à partir de $n \approx 2000$. Il semble aussi que cette méthode soit (très légèrement) plus rapide que la précédente alors que l'on effectue deux exponentiations rapides en $O(\log_2(n))$ multiplications et qu'une multiplication dans K requière (naïvement là encore) 4 multiplications et 2 additions. Je n'ai pas trouvé cela explicitement dans la documentation de `Sage` mais le fait que la seconde est (un tout petit peu) plus rapide

provient probablement la raison suivante[5] et en particulier du fait que `Sage` ne multiplie pas naïvement dans K ou dans l'anneau des matrices. Dans K on effectue ici en fait 2 fois $O(\log_2(n))$ multiplications dans K. Tout d'abord, la formule de Binet nous indique que le nombre de chiffres de F_n est proche de n. Par ailleurs, en mimant la preuve de Karatsuba du poly, on voit que pour calculer $(a+b\sqrt{5})(c+d\sqrt{5})$, il suffit de calculer trois produits, à savoir ac, bd et $(a+b)(c+d)$ de sorte qu'une multiplication dans K coûte en réalité $O(n^{\log_2(3)})$ plutôt que $O(n^2)$ si bien qu'on a une complexité C_n qui vérifie

$$C_n \leqslant 6C\log_2(n)n^{\log_2(3)} \quad \text{avec} \quad C>0 \qquad (*).$$

Concernant la première version avec la matrice, cette version requiert $O(\log_2(n))$ produit matriciels, que l'on peut effectuer en 8 multiplications d'entiers avec Karatsuba et obtenir[6] une complexité C'_n

$$C'_n \leqslant 8C\log_2(n)n^{\log_2(3)}$$

avec la même constante C que pour $(*)$. On peut améliorer un peu le résultat en se rendant compte qu'on a en fait besoin uniquement de 7 produits (c'est probablement ce que fait `Sage` en utilisant un algorithme appelé algorithme de Strassen pour calculer des produits matriciels). On voit alors bien que la version dans K devrait être très légèrement plus rapide.

6 Un peu d'arithmétique et de théorie des nombres

Un petit avant-goût du TP de la semaine prochaine avec un peu d'arithmétique.

- Comparer en utilisant `Sage` les fonctions suivantes :

$$x\mapsto \pi(x)=\#\{p\leqslant x \ : \ p \text{ premier }\}, \quad x\mapsto \frac{x}{\ln(x)} \quad \text{et} \quad x\mapsto \mathrm{Li}(x)=\int_2^x \frac{\mathrm{d}t}{\ln(t)}.$$

  Que constatez-vous ?

  ▶ **Correction.–** On semble constater que

$$\pi(x) \underset{x\to+\infty}{\sim} \frac{x}{\ln(x)} \underset{x\to+\infty}{\sim} \mathrm{Li}(x)$$

  et que l'approximation fournie par $\mathrm{Li}(x)$ semble meilleure[7]. Cet équivalent (et notamment le fait que la version Li est meilleure) a été conjecturé par Gauß en 1792 (à la main et à l'âge de 15 ans!) et par Legendre quelques années plus tard avant d'être démontré indépendamment en 1896 par Hadamard et La Vallée Poussin à l'aide d'outils issus de l'analyse complexe et des propriétés de la célèbre fonction ζ de Riemann.
  On semble également conjecturer que pour x assez grand

$$\frac{x}{\ln(x)} \leqslant \pi(x) \leqslant \mathrm{Li(x)}.$$

  Si la première inégalité ci-dessus peut être établie, la seconde est en revanche fausse et l'on peut montrer que la fonction $x\mapsto \mathrm{Li}(x)-\pi(x)$ change de signe infiniment souvent. En revanche, il faut attendre la valeur astronomique[8] de $x\geqslant 1,39822\times 10^{316}$ pour assister au premier changement de signe !
  On pourrait constater de même qu'on a

$$\pi_2(x) \underset{x\to+\infty}{\sim} 2\prod_{p\geqslant 3}\frac{p(p-2)}{(p-1)^2}\frac{x}{\log(x)^2} \underset{x\to+\infty}{\sim} 2\prod_{p\geqslant 3}\frac{p(p-2)}{(p-1)^2}\int_2^x \frac{\mathrm{d}t}{\log(t)^2}$$

  si $\pi_2(x)=\#\{p\leqslant x \ : \ \text{tel que} p \text{ et } p+2 \text{ sont premiers}\}$. Cet énoncé est bien sûr toujours un problème ouvert et complètement hors de portée ! Je vous renvoie à la littérature sur le sujet et aux conjectures de Hardy-Littlewood, de Bateman-Horn ou à l'hypothèse de Schinzel pour des compléments et des généralisations ! Une forme faible de cette conjecture concerne les écarts entre nombres premiers qui est un domaine de recherche très actif et qui a connu des avancées spectaculaires ces dernières années !

- Comparez $x\mapsto \displaystyle\sum_{p\leqslant x}\frac{1}{p}$ où la somme porte sur tous les nombres premiers inférieurs à x avec la fonction $x\mapsto \ln(\ln(x))$. Que conjecturez-vous concernant la série $\displaystyle\sum_p \frac{1}{p}$? Et concernant $\displaystyle\sum_{p\leqslant x}\frac{1}{p}-\ln(\ln(x))$?

  ▶ **Correction.–** On semble constater que

$$\sum_{p\leqslant x}\frac{1}{p} \underset{x\to+\infty}{\sim} \ln(\ln(x)) \quad \text{et que} \quad \lim_{x\to+\infty}\left(\sum_{p\leqslant x}\frac{1}{p}-\ln(\ln(x))\right)=M$$

5. Où tout n'est pas complètement rigoureux.
6. À peu de choses près parce que je n'ai pas pris en compte les additions et les multiplications par 5 par exemple.
7. L'équivalent $\frac{x}{\ln(x)} \underset{x\to+\infty}{\sim} \mathrm{Li}(x)$ est d'ailleurs un exercice intéressant à base d'intégrations par parties.
8. Je rappelle qu'on estime que le nombre d'atomes de l'univers est de l'ordre de 10^{80}.

pour une certaine constante $M \in \mathbf{R}$ qui a l'air de valoir entre 2,5 et 3. On peut en réalité démontrer la divergence de la série $\sum_p \frac{1}{p}$ et ainsi obtenir une preuve du fait qu'on a une infinité de nombres premiers ! Sinon l'équivalent ci-dessus découle du théorème des nombres premiers par exemple. On peut alors pousser la preuve plus loin et obtenir qu'en effet, il existe une constante

$$M = \gamma + \sum_p \left(\ln\left(1 - \frac{1}{p}\right) + \frac{1}{p} \right) \approx 0,261497$$

appelée constante de Meissel-Mertens et où γ désigne la constante classique d'Euler-Mascheroni telle que

$$\lim_{x \to +\infty} \left(\sum_{p \leqslant x} \frac{1}{p} - \ln(\ln(x)) \right) = M.$$

- Reprendre la question précédente pour $\sum_{\substack{p \\ p \equiv 1 (\mathrm{mod}\ 4)}} \frac{1}{p}$.

  ▶ **Correction.–** On semble constater que

  $$\sum_{\substack{p \leqslant x \\ p \equiv 1 (\mathrm{mod}\ 4)}} \frac{1}{p} \underset{x \to +\infty}{\sim} \frac{1}{2} \ln(\ln(x)) \quad \text{et que} \quad \lim_{x \to +\infty} \left(\sum_{p \leqslant x} \frac{1}{p} - \frac{1}{2} \ln(\ln(x)) \right) = M_1$$

  pour une certaine constante $M_1 \in \mathbb{R}$. On constaterait de même que

  $$\sum_{\substack{p \leqslant x \\ p \equiv 3 (\mathrm{mod}\ 4)}} \frac{1}{p} \underset{x \to +\infty}{\sim} \frac{1}{2} \ln(\ln(x)) \quad \text{et que} \quad \lim_{x \to +\infty} \left(\sum_{p \leqslant x} \frac{1}{p} - \frac{1}{2} \ln(\ln(x)) \right) = M_2$$

  pour une certaine constante $M_2 \in \mathbf{R}$ et telle que $M_1 + M_2 = M$ avec M la constante de Meissel-Mertens. Ces résultats sont en fait des théorèmes et indiquent [9] que les nombres premiers se répartissent équitablement entre les entiers qui sont congrus à 1 et à 3 modulo 4. Ces séries interviennent dans la démonstration du théorème de la progression arithmétique de Dirichlet et permettent dans ce cas précis d'établir qu'il existe une infinité de nombres premiers congrus à 1 modulo 4 et à 3 modulo 4.

- Que parvenez-vous à conjecturer en utilisant `Sage` concernant $\sum_{p \text{ jumeaux}} \left(\frac{1}{p} + \frac{1}{p+2} \right)$ où la série porte sur tous les nombres premiers jumeaux ?

  ▶ **Correction.–** On semble conjecturer que la série converge vers une valeur un peu en-dessous de 2. Il s'agit en fait d'un théorème établi par le mathématicien norvégien Viggo Brun utilisant des méthodes dites de crible [10]. Cette série converge donc mais très lentement et la valeur de sa somme est environ 1,9. On dispose d'heuristiques nous poussant à croire qu'il existe une infinité de tels nombres premiers jumeaux et nous donnant même un équivalent du théorème des nombres premiers dans ce cadre mais malheureusement le problème reste toujours ouvert à l'heure actuelle ! L'étude de cette série aurait pu fournir une démonstration de ce fait si la série avait été divergente, malheureusement elle converge !

7 Un exercice supplémentaire

J'avais donné en fin de TP pour ceux qui avaient terminé en avance, l'exercice suivant (qui est corrigé dans le Notebook Jupyter).

- Soit $u_0 \in \mathbf{N}*$ donné et $A \geqslant 1$. On définit alors une suite $(u_n)_{n \in \mathbf{N}}$ par récurrence de la façon suivante

  $$u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ si } n \equiv 0 \,(\mathrm{mod}\, 2) \\ Au_n + 1 \text{ sinon.} \end{cases}.$$

  On note alors $f(u_0, A)$ la valeur minimale de $n \in \mathbb{N}$ telle que $u_n = 1$ (éventuellement $+\infty$ si la suite n'atteint jamais cette valeur). Calculer avec `Sage` les valeurs de $f(u_0, 3)$ pour $u_0 \in \{2, \ldots, 100\}$. Que pouvez-vous conjecturer ? Que se passe-t-il lorsque $A = 5$?

  ▶ **Correction.–** On conjecture que pour $A = 3$, $f(u_0, 3)$ semble être fini pour toute valeur de $u_0 \in \mathbb{N}^*$. Il s'agit dans le cas $A = 3$ de la célèbre suite de **Syracuse** et on conjecture en effet que $f(u_0, 3)$ semble être fini pour toute valeur de $u_0 \in \mathbb{N}^*$ mais malgré

9. Noter que $2 = \varphi(4)$
10. Inspirées du crible d'Erathostène.

l'apparente simplicité de la définition de la suite $(u_n)_{n\in\mathbb{N}}$, le problème est encore largement ouvert à l'heure actuelle [11] !
Au contraire, lorsque $A = 5$, il semble que pour de nombreuses valeurs de u_0, $f(u_0, 5) = +\infty$. Dans ce cas, on observe plusieurs types de comportements : les suites qui repassent par 1 et se retrouvent dans un cycle de longueur 7, celles qui atteignent 5 et qui font alors un cycle [12] de longueur 2 et encore d'autres cycles commençant par 13 et 17 et celles qui ne font pas de cycles et semblent diverger vers $+\infty$ mais à ma connaissance, on ne sait pas encore démontrer ce dernier point ni expliquer exactement ce qui fait la différence avec le cas $A = 3$ ou encore si ces cycles sont les seuls. On conjecture cependant que *la plupart* des valeurs de u_0 donnent [13] lieu à une suite divergente même si l'on ne sait pas en exhiber une seule !

11. Quoi qu'en dise I. Aberkane...
12. Noter que si $u_n = 5$, alors $u_{n+1} = 10$, $u_{n+2} = 5$, $u_{n+3} = 10$, etc...
13. Vous pouvez faire l'essai et afficher la suite avec $u_0 = 7$.