PROJET EULER
Problème 119

Table des matières

I Résolution du problème **2**

1 Présentation **2**

2 Méthodes de réflexion **2**
2.1 Méthode itérative . . . 2
2.1.1 Algorithme de principe . . . 2
2.1.2 Développement . . . 3
2.2 Seconde méthode . . . 3
2.2.1 Algorithme de principe . . . 3
2.2.2 Développement . . . 3

3 Le problème "J'ai de la chance" **4**

II Optimisation **5**

4 Rappels de nos objectifs **5**

5 Comment trouver les bornes de recherche ? **5**
5.1 Comment borner a ? . . . 5
5.2 Comment borner b ? . . . 5

6 Optimisation de la fonction qui calcule la somme des chiffres d'un nombre ? **6**

7 Comment déterminer vMax ? **6**

III Comparaison des deux méthodes **7**

IV Annexes **9**

Première partie
Résolution du problème

1 Présentation

Le problème 119 du Projet Euler consiste à trouver le 30ème nombre tel que la somme de ses chiffres élevé à une certaine puissance soit égale à lui-même. Par exemple, le 1er nombre respectant ces règles est 81 car la somme des chiffres de 81 est : $8 + 1 = 9$ et $9^2 = 81$. De même, le deuxième est 521 : $5 + 2 + 1 = 8$ et $8^3 = 521$. Dans l'énoncé du problème, on nous dit que le dixième est 614 656. On rappelle qu'un nombre doit contenir au moins deux chiffres pour avoir une somme.

2 Méthodes de réflexion

2.1 Méthode itérative

Voici les algorithmes de principe d'une méthode itérative permettant de trouver le n-ème nombre respectant les règles du problème précédemment expliquées, ainsi que de la fonction $sommeChiffreNombre$ qui permet de calculer la somme des chiffres d'un nombre. Cette méthode est dite "iérative" car elle itère sur tous les entiers tant qu'on n'a pas trouvé le nombre que l'on recherche :

2.1.1 Algorithme de principe

```
1  Algorithme : Projet Euler, Problème 119, Première Méthode
2  Type de Retour : Nombre, n-ème nombre dont la somme de ses chiffres élevé à une
       certaine puissance est égale à lui-même
3  Arguments : Nombre n
4
5  index := 0
6  nombreAct := 10
7  tant que index < n:
8    decomposition := sommeChiffreNombre(nombreAct)
9    puiss := decomposition ^ k, k = [2, ..., à définir]
10   si puiss == nombreAct:
11     index := index + 1
12 retourner puiss
```

```
1  Algorithme : Projet Euler, Problème 119, Fonction SommeChiffreNombre
2  Type de retour : Nombre, somme des chiffres de n
3  Argument : Nombre n
4
5  somme := 0
6  tant que n != 0:
7      somme := somme + nombre % 10
8      nombre := nombre / 10 arrondi à la valeur inferieure
9  retourner somme
```

2.1.2 Développement

Une première méthode pour calculer la somme des chiffres d'un nombre consiste à récupérer le reste de la division du nombre par 10, de l'ajouter à la somme puis faire la division entière de notre nombre par 10 et répéter ceci jusqu'à ce qu'il soit égal à 0. Par exemple, pour 614 656, le reste de 614656/10 est 6 et la division entière de 614 656 par 10 donne 61 465. Ensuite le reste de 61465/10 est 5. Ainsi de suite, on obtient $6+5+6+4+1+6$. Ici, on divise 6 par 10, ce qui donne 0 et qui cause la sortie de la boucle. On a donc bien obtenu la somme des chiffres de 614 656.

Pour trouver le 30ème nombre respectant les règles citées précédemment, il existe une méthode itérative, vérifiant pour chaque nombre, en partant de 10, s'il fait partie des nombres que l'on recherche. Ainsi, pour chaque nombre, on calcule la somme de ses chiffres, on l'élève au carré (on appelera ce nombre n), on vérifie si le résultat est égal au nombre de départ, auquel cas on ajoute 1 à un compteur qui doit atteindre 30. Si n est inférieur au nombre de départ, on augmente la puissance et on recommence, s'il est supérieur on s'arrête car il ne pourra plus jamais être égal au nombre de départ, et on passe au nombre suivant. Lorsque le compteur atteint 30 c'est qu'on a trouvé le nombre que l'on cherche. Or, cette méthode est beaucoup trop longue car l'ordre de grandeur du 30ème est de 10^{15}, sachant que l'on calcule la somme des chiffres de tous les nombres précédent.

2.2 Seconde méthode

Une autre méthode ne consiste pas à itérer sur les entiers pour vérifier s'ils font partie des nombres recherchés, mais plutôt d'itérer sur de plus petits nombres a et b afin de vérifier si la somme des chiffres de a^b est égale à a.

2.2.1 Algorithme de principe

```
1  Algorithme : Projet Euler, Problème 119, Seconde Méthode
2  Type de Retour : Nombre, n-ème nombre dont la somme de ses chiffres élevé à une
       certaine puissance est égale à lui-même
3  Arguments : Nombre n
4
5  pour a allant de 2 à 400:
6      pour b allant de 2 à 50:
7          value := a ^ b
8          si sommeChiffreNombre(value) = a:
9              ajouter value dans liste
10 trier liste
11 retourner liste[n - 1]
```

2.2.2 Développement

Cette seconde méthode ne consiste pas à réduire la complexité de notre fonction qui calcule la somme des chiffres d'un nombre mais plutot d'atteindre plus rapidement le 30ème sans avoir à calculer la somme des chiffres de tous les nombres. Pour cela, nous pouvons prendre des petits nombres a (par exemple jusqu'à 400 pour commencer) que l'on éléve à des petites puissances b (par exemple jusqu'à 50 pour l'instant) et calculer la somme des chiffres de chacun de ces nombres a^b. Lorsque $a^b = a$, on ajoute ce dernier à une liste. Après avoir testé tous les nombres, on trie la liste pour en récupérer le 30ème. De cette façon, on calcule la somme des chiffres de beaucoup moins de nombre puisqu'il y en a une multitude qui ne sont pas une puissance d'un autre. Par

exemple, aucun nombre élévé à une certaine puissance est égal à 26 donc on ne calcule jamais la somme des chiffres de 26. Ceci réduit clairement le temps d'éxécution de notre programme et nous avons réussi à trouver le nombre recherché qui est 248 155 780 267 521.

3 Le problème "J'ai de la chance"

Malheureusement, cette méthode a posé un autre problème. Si l'on veut développer un programme générique qui pourrait retourner un nombre quelconque de la liste de ceux qui respectent les règles, il faut compter sur la chance. En effet, dans notre cas, nous avons décider d'aller jusqu'a 400^{50} et heureusement nous avons trouvé le bon résultat. Mais rien ne nous disait auparavant que le nombre que l'on cherchait n'était pas égal à 500^{10} ou 5^{75} (sachant que ces deux nombres sont plus petits que 400^{50} mais que nous ne les testons pas). Si c'était le cas, notre programme n'aurait pas renvoyé le bon résultat puisqu'il ne l'aurait même pas étudié donc pas ajouter à la liste. C'est ainsi qu'on s'est demandé s'il n'y avait pas un moyen de connaître à l'avance le a et le b maximum pour lesquels le programme donne le bon résultat.

Deuxième partie
Optimisation

4 Rappels de nos objectifs

Suite à notre dernier entretien, nos objectifs sont les suivants :
- Etudier la piste du logarithme afin de déterminer à l'avance nos bornes de recherche
- Déterminer les limites de notre fontion qui calcule la somme des chiffres d'un nombre
- Essayer de supprimer ce phénomène de chance dans notre programme

5 Comment trouver les bornes de recherche ?

Ici, notre objectif est de borner a et b de manière à ce que a^b soit inférieur à un certain nombre défini à l'avance. Ainsi, on considère que lorsque l'on cherche le n-ième terme de notre liste de nombres, on connait approximativement son ordre de grandeur et il existe une valeur connue, supérieure à cette dernière. Si on appelle $vMax$ cette valeur maximum, le but est donc de trouver pour quelles valeurs de a et de b, $a^b < vMax$.

5.1 Comment borner a ?

Nous devons trouver la valeur maximum de a telle que $a^b < vMax$. Pour cela, il faut prendre la valeur minimum de b, car $\forall a1, a2, vMax, b1, b2 \in \mathbb{R}, b1 > b2 \Rightarrow \exists a1 > a2, a2^{b1} > a1^{b2}$.

Ainsi, on obtient l'inéquation suivante :

$$a^2 < vMax$$
$$\Leftrightarrow \ln a^2 < \ln vMax$$
$$\Leftrightarrow 2 * \ln(a) < \ln(vMax)$$
$$\Leftrightarrow \ln(a) < \frac{\ln(vMax)}{2}$$
$$\Leftrightarrow a < e^{\frac{\ln(vMax)}{2}}$$

Ainsi, si l'on connaît $vMax$, on sait à l'avance que a ne devra pas dépasser $e^{\frac{\ln(vMax)}{2}}$

5.2 Comment borner b ?

Une fois que l'on connaît la valeur maximum de a, on peut exprimer la valeur limite de b en fonction de a et de $vMax$. On a donc l'inéquation suivante :

$$a^b < vMax$$
$$\Leftrightarrow \ln a^b < \ln vMax$$
$$\Leftrightarrow b * \ln(a) < \ln(vMax)$$
$$\Leftrightarrow b < \frac{\ln(vMax)}{\ln a}$$

Ainsi, si l'on connaît $vMax$ et a , on sait à l'avance que b ne devra pas dépasser $\frac{\ln(vMax)}{\ln a}$

6 Optimisation de la fonction qui calcule la somme des chiffres d'un nombre ?

Lorsque l'on borne a et b, le temps de calcul augmente largement car en supprimant la chance de notre programme on augmente largement le nombre d'appels à la fonction sommeChiffresNombre afin d'être sûr de trouver le bon résultat. Nous avons donc essayé de réduire le temps de calcul de cette fonction, d'autant plus que celle-ci ne fonctionne plus à partir d'une certaine valeur. En effet, nous n'avons pas réussi à en connaître la raison mais à partir de 10^{17} notre fonction ne retourne pas le bon résultat. Nous avons donc essayé une différente méthode pour calculer la somme des chiffres d'un nombre et cette dernière s'est retrouvé beaucoup plus efficace, pas tellement au niveau du temps de calcul mais plus au niveau de la complexité et du fait qu'elle fonctionne pour toutes les valeurs. Elle consiste à transformer le nombre en chaîne de caractères et calculer la somme des caractères de la chaîne obtenue. En python, la fonction correspond donc à :

```
1 def sommeChiffreNombre(nombre):
2     return sum([ int(c) for c in str(nombre) ])
```

7 Comment déterminer vMax ?

Pour pouvoir borner a et b, il faut d'abord déterminer $vMax$. Le temps d'exécution dépendra de $vMax$ car si il est trop éloigné de la valeur recherchée, le temps d'exécution du programme sera trop élevé. Cependant, on ne peut connaître à l'avance la valeur recherchée. C'est pourquoi nous avons fait en sorte d'augmenter progressivement $vMax$: si $vMax$ est atteint sans que l'on ait trouvé le n^{eme} terme (où le n^{eme} terme est celui que l'on recherche), on augmente $vMax$. La formule que nous avons choisit pour augmenter $vMax$ est :

$$vMax = vMax * 5^{(n-\text{la quantité déjà trouvé})/2}$$

Cette formule n'est pas admise mais résulte de nos expériences. Par exemple : on cherche le 20ème terme, et on en a déja trouvé 10 avec le $vMax$ actuel. Ici, $n -$ la quantité déjà trouvé $= 10$. On multiplie donc $vMax$ par5^5 ($5^{10/2}$). Ainsi, peu importe le $vMax$ d'origine, le programme finira toujours pas trouvé la solution. Néanmoins il existe des $vMax$ plus efficaces pour certaines valeurs. Par exemple : la valeur recherchée est 1000000 et le $vMax$ d'origine est 2. Admettons que $vMax$ atteigne 999999, le programme n'aura pas trouvé la bonne valeur mais $vMax$ augmentera de beaucoup trop pour rien. Alors que si le $vMax$ d'origine avait été égal à 3, on aurait atteint plus rapidement et simplement 1000000. On remarque donc que le programme est plus rapide pour $vMax = 2$ que pour $vMax = 3$, même s'il fonctionne dans les deux cas. Ce n'est ici qu'un exemple, et en suivant les contraintes de ce problème, on peut déterminer un palier : si $n < 10$, il faut choisir $vmax = 10$, sinon il faut commencer avec $vMax = 700000$.

Troisième partie

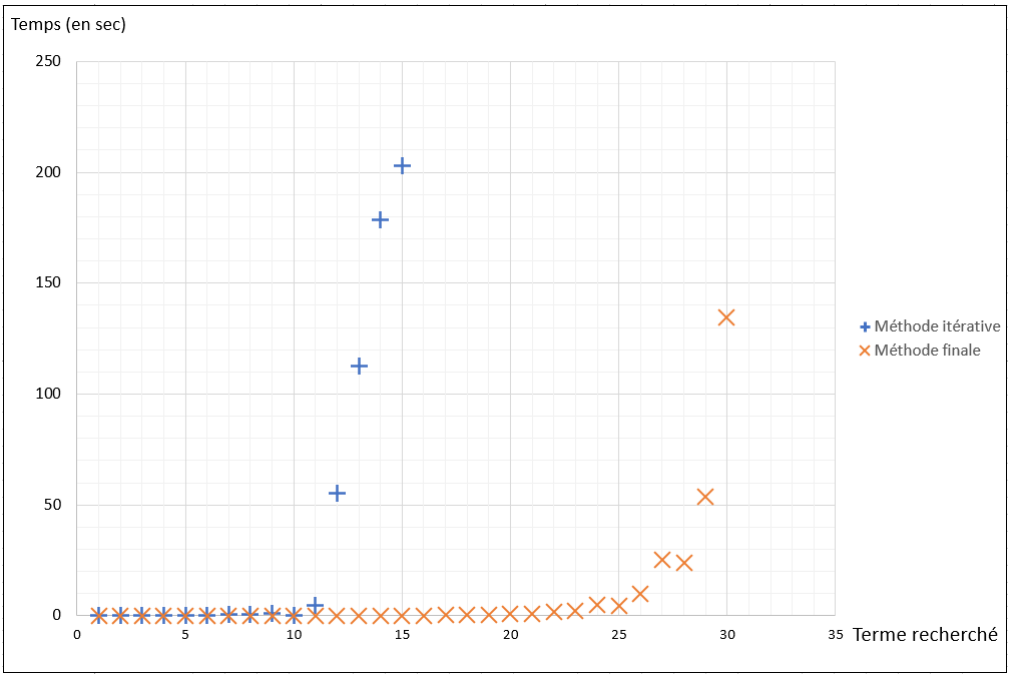
Comparaison des deux méthodes

Afin de comparer l'efficacité de nos deux méthodes nous avons réaliser un tableau représentant le temps d'exécution de nos deux programmes en fonction de la valeur recherchée avec le même $vMax$ à chaque fois :

	Méthode itérative		Méthode finale	
	Solution	Temps (en secondes)	Solution	Temps (en secondes)
1	81	0,0000000000	81	0,0000000000
2	512	0,0009372234	512	0,0000000000
3	2 401	0,0060114861	2 401	0,0009980202
4	4 913	0,0089755058	4 913	0,0000000000
5	5 832	0,0109701157	5 832	0,0000000000
6	17 576	0,0398950577	17 576	0,0009968281
7	19 683	0,4189038277	19 683	0,0009970665
8	234 256	0,5984334946	234 256	0,0039896965
9	390 625	0,9993245602	390 625	0,0029911995
10	614 656	0,1580773115	614 656	0,0039815903
11	1 679 616	4,7004325390	1 679 616	0,0540158749
12	17 210 368	55,1016736031	17 210 368	0,0329442024
13	34 012 224	112,5149283409	34 012 224	0,0309162140
14	52 521 875	178,7131516933	52 521 875	0,0688104630
15	60 466 176	203,1438820362	60 466 176	0,0289285183
16	Trop long		205 962 976	0,0718114376
17	Trop long		612 220 032	0,1695432663
18	Trop long		8 303 765 625	0,4039208889
19	Trop long		10 460 353 203	0,3839831352
20	Trop long		24 794 911 296	0,8347561359
21	Trop long		27 512 614 111	0,8766241074
22	Trop long		52 523 350 144	1,9189016819
23	Trop long		68 719 476 736	2,0086295605
24	Trop long		271 818 611 107	4,7732377052
25	Trop long		1 174 711 139 837	4,6356072426
26	Trop long		2 207 984 167 552	10,0242002010
27	Trop long		6 722 988 818 432	25,2295100689
28	Trop long		20 047 612 231 936	23,8607327938
29	Trop long		72 301 961 339 136	53,7165598869
30	Trop long		248 155 780 267 521	134,3962495327

Ainsi, grâce à ce tableau, nous remarquons qu'à partir du 12ème terme, le temps d'exécution de la première méthode devient plus de 1000 fois supérieur à celui de la seconde. De plus, à partir du 16ème terme, la première méthode devient beaucoup trop long pour être mesuré. L'efficacité de la seconde méthode par rapport à la première est donc avéré puisqu'elle met autant de temps à trouver le 30ème terme que la première pour trouver le 13ème.

Nous avons aussi réaliser un graphique représentant l'évolution de ces temps d'exécution en fonction du terme recherché :



Ici, nous voyons bien que pour les deux méthodes, le temps d'exécution devient exponentiel à partir d'un certain point. Ce dernier est beaucoup plus grand pour la seconde méthode ce qui explique sa meilleure efficacité. Nous pouvons donc en conclure avec ce graphique que la complexité en temps des deux programmes est de l'odre de $O(e^n)$ où n est l'index du terme que l'on recherche. Nous n'avons pas réussi à montrer ceci grâce au calcul car la complexité en temps de nos programmes dépendent de plusieurs éléments à la fois : n, $vMax$, le nombre de chiffres du $n^{\text{ème}}$ terme.

Quatrième partie

Annexes

Méthode itérative

```python
from math import floor
import time

def sommeChiffreNombre(nombre):
    """
    Renvoie la somme des chiffres du nombre passé en paramètres
    """
    somme = 0
    while (nombre != 0):
        somme += nombre%10
        nombre=floor(nombre/10)
    return somme


def A(n):
    """
    Retourne le n-ème terme répondant aux critères du problème
    """
    index = 0
    nombreAct = 10
    valeurRetournee = 0
    while index != n:
        decomposition = sommeChiffreNombre(nombreAct)
        k = 2
        if decomposition != 1:
            while 1:
                puiss = decomposition**k
                if puiss == nombreAct:
                    index += 1
                    valeurRetournee = nombreAct
                    break
                if puiss > nombreAct:
                    break
                k += 1
        nombreAct += 1
    return valeurRetournee


nbTeste = int(input("Rentrez le n-ème terme recherché : "))
begin = time.time()
print("Résultat au problème 119 avec", nbTeste, "comme entrée :", A(nbTeste))
print(f"Duration = {time.time() - begin} seconds to complete.")
```

Méthode finale

```
from math import floor, log, exp, ceil
import time

def sommeChiffreNombre(nombre):
    '''
    :param nombre: nombre à décomposer
    :return: somme des chiffres de nombre
    '''
    return sum([ int(c) for c in str(nombre) ])


def A(n, vMax):
    '''
    :param n: terme dont on cherche la valeur
    :param vMax: variable permettant de borner a et b
    :return: n-ème terme répondant aux critères du problème
    '''
    listeNombre = []
    bMaximum = 2
    aMaximum = 2
    listebMaximum = {}
    while len(listeNombre) < n:
        a = 2
        aPrevious = aMaximum
        aMaximum = exp(log(vMax)/2)
        while a < aMaximum:
            if a >= aPrevious:
                bMinimum = 2
            else:
                bMinimum = listebMaximum[a]
            bMaximum = floor(log(vMax) / log(a))+1
            listebMaximum[a] = bMaximum
            for b in range(bMinimum, bMaximum):
                value = a**b
                if sommeChiffreNombre(value) == a:
                    listeNombre.append(value)
            a += 1
        vMax *= 5**ceil((n - len(listeNombre))/2)
    listeNombre.sort()
    return listeNombre[n-1]


nbTeste = int(input("Rentrez le n-ème terme que vous souhaitez obtenir : "))
vTeste = int(input("Rentrez votre 'vMax' : "))
```