Persistance des nombres

Année 2024 – 2025

Jeremy Fuso, Alix Oger élèves de classe terminale générale spécialité mathématique et physique chimie

Établissement(s) : Lycée Raynouard (Brignoles - Var)

Enseignant·e(s) : Mourau Nelly, Berger Antoine, Guicheteau Denis

Chercheur·Chercheuse(s) : Thierry Champion laboratoire imath. Frédéric Havet - INRIA

1. Introduction

1.1. Présentation du sujet

On prend un nombre et on multiplie l'ensemble de ses chiffres entre eux afin d'obtenir un nouveau nombre, on prend donc ce nouveau nombre et on recommence jusqu'à ce que le résultat du produit soit un chiffre.
On appelle donc persistance d'un nombre, le nombre de multiplications nécessaires pour arriver au résultat avec un seul chiffre.
Par exemple 89 c'est $8 \times 9 = 72$, puis 72 c'est $7 \times 2 = 14$ et 14 c'est $1 \times 4 = 4$; il y a donc 3 étapes pour arriver à 4, la persistance de 89 est donc 3.

1.2. Résultats

- Le résultat de la multiplication des chiffres du nombre est inférieur au nombre.
- La plus grande persistance trouvée avec nos algorithmes est 11 pour 277777788888899 pour les nombres jusqu'à 10^{15}.

2. Premières recherches

Nous avons calculé à la main les premières persistances.
On remarque alors :

- ❖ Lorsqu'un nombre comporte un 0, son produit vaut 0 et sa persistance 1.

Démonstration :
Soit P un entier naturel de n chiffres strictement plus grand que 9 et a_i des entiers naturels compris entre 0 et 9 tel que: P=$\sum_{i=0}^{n} a_i \times 10^i$.
Si on a a_k = 0 avec k∈ □ et 0≤ $k <$ □ alors pour le produit des chiffres de P on a :
$\prod_{i=0}^{n} a_i = a_0 \times a_1 \times \ldots \times a_k \times \ldots \times a_n$
$= \prod_{i=0}^{k-1} a_i \times \prod_{i=k+1}^{n} a_i \times a_k$
$= \prod_{i=0}^{k-1} a_i \times \prod_{i=k+1}^{n} a_i \times 0$
= 0

- ❖ Si un nombre est composé uniquement de 1, alors on obtient 1 et une persistance de 1.

Démonstration :
Soit P un entier naturel strictement plus grand que 9 composé que de n chiffres tous égaux à 1 alors le produit de ces chiffre est:

$$\prod_{i=0}^{n} a_i = 1\times 1 \times \ldots \times 1 \times 1 = 1^{n+1} = 1$$

- ❖ Tout nombre qui contient un chiffre pair et un 5, a une persistance d'au plus 2.

Démonstration :
Soit P un entier naturel strictement plus grand que 9 et $a_i$un entier naturel compris entre 0 et 9 tel que : $P = \sum_{i=0}^{n} a_i \times 10^i$
On suppose qu'il existe k entier tel que $0 \leq k \leq n$ et a_k pair non nul et qu'il existe un j ∈ □ tel que $0 \leq j \leq n$ (évidemment $j \neq k$) et $a_j = 5$
alors $a_k = 2 \times t \; avec \; t = 1,2,3,4$ donc $\prod_{i=0}^{n} a_i = 2\times t \times 5 \times \frac{\prod_{i=0}^{n} a_i}{a_k a_j} \; \frac{\prod_{i=0}^{n} a_i}{a_k a_j} \in \mathbb{N}$

$$= 10\times t \times \frac{\prod_{i=0}^{n} a_i}{a_k a_j}$$

ainsi $\prod_{i=0}^{n} a_i$ est un multiple de 10 donc il comporte un 0 donc d'après la propriété précédente, sa persistance vaut 1 et donc P a pour persistance 2 (sauf s'il possède un chiffre égal à 0, dans ce cas, sa persistance vaut 1).

Nous avons ensuite calculé la persistance de nombres assez grands (plus de 5 chiffres) pris au hasard afin de déterminer si une persistance maximale ressortait. Nous avons ainsi abouti à une première conjecture, la persistance maximale semble être 4.

3. Les anagrammes

D'après le calcul des premières persistances nous avons pu remarquer que tous les anagrammes d'un même nombre ont une persistance identique. En effet, cela est dû au fait que le produit de tous les chiffres composant le nombre est le même peu importe l'ordre dans lequel on effectue ce produit.

Nous avons donc par la suite utilisé la formule des anagrammes afin de connaître le nombre d'anagrammes pour un nombre et ainsi réduire le champ des possibilités.

Soit k_j le nombre de fois que le chiffre j est dans le nombre alors le nombre d'anagrammes d'un nombre à n chiffres est : $\frac{n!}{\prod_{j=0}^{9} k_j!}$ avec évidemment $\sum_{j=0}^{9} k_j = n$

4. Création de programmes:

La recherche de persistance à la main étant longue et fastidieuse, nous avons réfléchi à la création de programme afin de rendre ce processus plus sûr et plus rapide.

Tout d'abord nous avons écrit les programmes de base, afin de pouvoir les utiliser par la suite dans des programmes plus complexes.

Nous avons ainsi commencé par le programme du produit des chiffres d'un nombre

```
def produit(n):
    q=n
    p=1
    while q>0 and p!=0:
        r=q%10
        q=q//10
        p=p*r
    return p
```

puis celui de la persistance:

```
def persistance(n):
    s=0
    while n>9:
        n=produit(n)
        s=s+1
    return s
```

sachant que nous cherchions à déterminer la persistance la plus grande possible ou encore à montrer qu'il existe une persistance infinie, la valeur de chaque persistance ne nous était donc pas nécessaire. Nous avons donc créé un programme qui nous donne uniquement les persistances ainsi que les valeurs pour lesquelles la persistance est supérieure à celle déterminée.

```
def final(n,m,M):
    L=[]
    chi=0
    A=[]
    for i in range (n,m):
        per_i=persistance(i)
        chi=i
        if per_i>=M:
            L.append(per_i)
            A.append(chi)
    print(len(L),A[0])
    return L,A
```

Dans cette fonction, n et m sont les bornes pour la recherche et M est la persistance que l'on veut atteindre ou dépasser.

On stocke alors les nombres et leur persistance dans les listes L et A.

puis, un programme qui nous donne la persistance maximum sur un intervalle donné

```
def maximum(n,m):
    maxi=0
    ind=0
    for k in range(m-n):
        if produit(n+k)!=0:
            p=persistance(n+k)

            if p>maxi:
                maxi=p
                ind=k+n
    return maxi,ind
```

Grâce à ces premiers programmes nous pouvons déjà dire que notre première conjecture d'une persistance maximum égale à 7 est fausse. En effet, sur l'intervalle $[0; 10000]$, la persistance maximale est 6; sur l'intervalle $[10000\ ;\ 100000]$, la persistance maximale est 7; sur l'intervalle 3 millions-5 millions la persistance maximale est 8.

Le problème est que ces programmes prennent du temps dès que l'on prend des grands nombres.

Les programmes nous permettant d'avancer nous avons donc continué avec la création de nouveaux programmes en cherchant à optimiser le temps de calcul. Nous nous sommes donc intéressés à la création d'une boucle qui calcule uniquement le produit du nombre, puis va chercher la persistance du résultat du produit dans la liste déjà existante et y ajoute 1. le programme calcule ensuite le produit de nombre suivant et ainsi de suite…

```
83  def Listpers(d,j):
84      D=[]
85      for i in range (j):
86          D.append((i,persistance(i)))
87      while j<=d:
88          f=produit(j)
89          t=1+D[f][1]
90          D.append((j,t))
91          j=j+1
92      return D[d]
93
```

Pour que cela fonctionne de manière optimisée il aurait fallu que la liste créée préalablement puisse être gardée pour ne pas tout recommencer à chaque fois. Nous ne sommes donc pas allés plus loin sur cette voie. Cependant, pour être sûrs que la boucle finissait, il a été nécessaire de montrer que le résultat de la multiplication des chiffres du nombre est inférieur au nombre, on a donc écrit la démonstration suivante:

- ❖ Le résultat de la multiplication des chiffres du nombre est inférieur au nombre.

Démonstration :

Soit k un nombre sous la forme $k = \sum_{i=0}^{n} a_i \times 10^i$ et $k > 9$. $\forall$ i $\in \mathbb{N}$, $\forall i, a_i \in \mathbb{N}$ et $a_i \in [\,1\,;\,9\,]$
On note $Pers_1(k)$ la multiplication de tous les chiffres de k donc $Pers_1(k) = \prod_{i=0}^{n} \quad a_i$

$\forall i \in [0; n], a_i \leq 9$ d'où par produit $\prod_{i=0}^{n} a_i \leq a_n \times 9^n$ et

$\forall i \in [0; n-1], a_i > 0$ donc $\sum_{i=0}^{n} a_i \times 10^i > a_n \times 10^n$

De plus, pour $n \in \mathbb{N}^*$:

9 < 10

⇔ ln 9 < ln 10 car la fonction logarithme népérien est strictement croissant sur $\mathbb{R}^+$

⇔ n × ln 9 < n × ln 10 car $n > 0$

⇔ $\ln 9^n < \ln 10^n$

⇔ $9^n < 10^n$ car la fonction exponentielle est strictement croissante sur $\mathbb{R}^+$

⇔ $a_n \times 9^n < a_n \times 10^n$ car $a_n > 0$

Donc vu que $\prod_{i=0}^{n} a_i \leq a_n \times 9^n$ et $\sum_{i=0}^{n} a_i \times 10^i \geq a_n \times 10^n$ et que $a_n \times 9^n < a_n \times 10^n$

On peut affirmer que $\sum_{i=0}^{n} a_i \times 10^i > \prod_{i=0}^{n} a_i$

Si a_i peut être égal à 0 alors $Pers_1(k) = 0$ donc $Pers_1(k) < k$ (car $k > 9$)

Donc le produit des chiffres d'un nombre est inférieur au nombre initial ce qui implique que nous pouvons trouver la persistance d'un nombre en un nombre fini d'étapes.

Enfin, comme remarqué précédemment, l'assemblage de certains chiffres conduisent à faire apparaître des 0 ou ne change rien (par exemple le 1). Nous avons donc créé un programme qui construit un nombre à partir des chiffres que l'on veut y faire intervenir et de la taille finale du nombre que l'on souhaite.

Ce programme calcule la persistance de tous les nombres utilisant uniquement les chiffres mentionnés et renvoie le premier nombre à avoir la plus grande persistance trouvée.

```
#une variable global pour limiter l'utilisation mémoire
#et mettre les valeurs des nombres persistants
global listp
listp=[]
#une fonction qui construit des nombres de taille+1 chiffres utilisant
#uniquement les nombres précisés
def nbpersi(nb,taille,nombre):
    global listp
    if taille>-1:

        for n in nb :
            nombre=nombre+n*10**taille
            nbpersi(nb, taille-1, nombre)
            nombre=nombre-n*10**taille
    else:
        if persistance(nombre)>=11:
            listp.append(nombre)
```

puis un deuxième plus efficace qui nous à permis de déterminer la plus grande persistance que l'on a, c'est-à-dire 11 pour 277777788888899.

Pour ce dernier programme, nous avons utilisé des chaînes de caractères pour faire des mots avec nos chiffres, rangés par ordre croissant. Cela permet de ne faire qu'une occurrence de nos anagrammes, et donc de pouvoir tester plus de nombres (mais nous n'avons pas dépasser les 10^{30})

```python
import itertools

def multiplicative_persistence(n):
    """Calcule la persistance multiplicative d'un nombre."""
    steps = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

def generate_ascending_numbers(max_length):
    """Génère des nombres avec les chiffres 2, 7, 8 et 9 dans une structure ascendante."""
    digits = ['2', '7', '8', '9']
    max_persistence = 0
    best_number = None

    for length in range(1, max_length + 1):
        for combo in itertools.combinations_with_replacement(digits, length):
            num_str = ''.join(combo)
            num = int(num_str)
            persistence = multiplicative_persistence(num)
            if persistence > max_persistence:
                max_persistence = persistence
                best_number = num
                print(f"Nombre: {num}, Persistance: {persistence}")

    return max_persistence, best_number


max_length = 15  # Limite de longueur pour les nombres

max_persistence, best_number = generate_ascending_numbers(max_length)

print(f"\nNombre avec la plus grande persistance: {best_number}")
print(f"Persistance maximale trouvée: {max_persistence}")
```

En parallèle des programmes nous avons également suivi une autre piste, celle de réaliser le processus dans le sens inverse. Dire que l'on fait le processus à l'envers signifie que l'on part d'un nombre k et on cherche un nombre dont le produit de ces chiffres donne k.

L'intérêt de cela est que si on note t la persistance de k alors ce nouveau nombre à une persistance de t+1 et par définition de ce processus si k ne comporte pas que des nombres premiers inférieur ou égal à 7 en décomposition de facteurs premiers alors on ne peut pas trouver un produit de chiffres donnant k.

Cette recherche est encore en cours.

5. Conclusion

Pour conclure, nous n'avons pas répondu à la question initiale qui demandait s'il existait une persistance maximale pour les entiers. Néanmoins, nous avons trouvé un certain nombre de petits théorèmes et programmer des algorithmes qui nous font conjecturer qu'il n'existe pas de persistance maximale mais la croissance de cette persistance maximale est très petite.