Travaux Pratiques de Python

Cours de Méthodologie

—Licence MPI L1 S2 - Info 121—

Traitement de donnée

Dans cette séance de TP, nous allons appliquer les méthodes de traitement de donnée à l'étude du temps de calcul de trois algorithmes d'arithmétique sur les grands nombres.

1 Arithmétique de la machine et des grands entiers

Dans beaucoup de langage de programmation, le type des entiers (par exemple `int` en `Java`, `C` ou `C++`) ne permet de représenter qu'un intervalle fini des entiers relatifs (typiquement $[2^{63}, 2^{63} - 1]$, sur beaucoup de machines modernes). Ceci est dû au fait que le processeur de la machine code les entiers en binaire avec un nombre fixe de chiffres (typiquement 64). On appellera les entiers de cet intervalle **petits entiers**. Chaque opération sur les petits entiers est programmée par une unique instruction du langage de la machine et est directement effectuée par un circuit électronique dont le temps de calcul ne dépend pas de la taille de l'entier. Par exemple, l'addition de deux nombres se fait, sur les machines de la famille `i86`, par l'instruction `ADD` dont le temps de calcul est (pour faire simple) un cycle d'horloge, c'est-à-dire 0.5 ns sur une machine à 2 Ghz. Pour cette raison, on appelle aussi les petits entiers : «**entiers machines**».

En Python, quand on sort de l'intervalle des petits entiers, on bascule automatiquement sur une bibliothèque de **grands entiers** ou **entiers multiprécision** nommée GMP. Pour noter le passage aux entiers GMP, Python ajoute le suffixe `L` à la fin de l'entier. Voici, par exemple, quelques puissances de 2 :

```
>>> 2**3
8
>>> 2**10
1024
>>> 2**62
4611686018427387904
>>> 2**63
9223372036854775808L
>>> 2**64
18446744073709551616L
>>> 2**100
1267650600228229401496703205376L
```

Quand on passe aux grands entiers, les opérations usuelles (addition, multiplication...) sont effectuées par un sous-programme dont le temps de calcul augmente avec la taille de l'entier. Dans ce travail, on veut mesurer et étudier **la variation de ce temps de calcul en fonction de la taille de l'entier**.

Après cette première partie introductive, la deuxième partie explicite la manipulation des grands entiers en Python à travers des fonctions (i) de représentation des grands entiers, (ii) de somme de grands entiers, (iii) de produits de grands entiers selon deux algorithmes différents, la multiplication

dite «naive» et la multiplication rapide de Karatsuba. Le troisième partie propose trois modèles pour les temps de calcul des trois opérations citées, la quatrième partie indique comment mesurer un temps de calcul en Python. Et pour finir la cinquième partie énumère les différentes étapes du travail à effectuer en séance de TP.

Note 1 *Pour la séance de TP, vous n'avez pas besoin de comprendre le fonctionnement interne des fonctions présentées dans la partie 2. Il vous suffit ce comprendre ce qu'elles calculent et comment les utiliser. Pour ceci, vous pouvez vous inspirer des exemples fournis dans les fonctions.*

2 Manipulation des grands entiers en Python

La bibliothèque GMP utilise des codes extrêmement optimisés qui sont difficiles à lire (c'est un mélange de `C` et de langage machine) et dont les temps de calcul sont très courts donc difficiles à mesurer. De plus, depuis Python, on n'a pas le choix de l'algorithme utilisé pour effectuer les opérations, il est choisi automatiquement. Pour les besoin de ce TP, on va utiliser un code écrit en Python qui a pour objectif d'être compréhensible et ne vise absolument pas à avoir de bonnes performances.

2.1 Représentation des entiers

Note 2 *On ne représentera que des entiers* positifs ou nuls. *Un entier n sera représenté par la liste de ses chiffres en base* 10 dans l'ordre du poids faible au poids fort*, c'est-à-dire l'ordre inverse.*

L'avantage est que le chiffre correspondant à la puissance i de 10 est stocké dans la case i de la liste. Ainsi, le nombre qui s'écrit 42348 en base 10, se décompose comme ceci

$$42348 = 8 * 10^0 + 4 * 10^1 + 3 * 10^2 + 2 * 10^3 + 4 * 10^4$$

et sera donc représenté par la liste `[8, 4, 3, 2, 4]`

Note 3 *Le nombre* 0 *est représenté par la liste vide `[]`.*

Voici les fonctions de conversion entier `int` vers liste qui sont appelées `nb` dans le code. Les parties de code entre `"""` sont des commentaires qui contiennent des exemples.

```
Base = 10

def intnb(nbint):
    """
    Convertis un nombre en liste

    >>> intnb(0)
    []
    >>> intnb(1)
    [1]
    >>> intnb(3245)
    [5, 4, 2, 3]
    """
    res = []
    while nbint:
```

```
        res.append(nbint % Base)
        nbint = nbint // Base
    return res

def nbint(nb):
    """
    Convertis une liste en nombre

    >>> nbint([])
    0
    >>> nbint([1])
    1
    >>> nbint([5, 4, 2, 3])
    3245

    On verifie que les deux conversions sont bien inverse:

    >>> for i in range(200):
    ....:     assert(nbint(intnb(i)) == i)
    """
    res = 0
    for i in nb[::-1]:
        res = res*10 + i
    return res
```

2.2 Somme

On n'a pas écrit directement la fonction qui ajoute deux entiers. On a écrit à la place la fonction plus générale `ajoute_decal` qui ajoute à la liste `n` le nombre `m` décalé de `d` (on en aura besoin pour la multiplication). Voici, une illustration de `ajoute_decal(123,13, 2)` :

```
  123
+13..
-----
 1423
```

```
def ajoute_decal(n, m, d):
    """
    Ajoute a n le nombre m decale de d
    On modifie n

    >>> n = [3,2,1]; ajoute_decal(n, [3,1], 2); n
    [3, 2, 4, 1]
    >>> n = []; ajoute_decal(n, [1], 1); n
    [0, 1]
    >>> n = [9, 9]; ajoute_decal(n, [1], 1); n
    [9, 0, 1]
    >>> n = [9, 9]; ajoute_decal(n, [1, 1], 1); n
    [9, 0, 2]
    """
```

```
    if m == []: return
    # On complete n avec des zero si besoin
    n.extend([0]*(len(m) + d - len(n)))
    retenue = 0
    for i in range(len(m)):
        n[d] += m[i]+retenue
        if n[d] >= Base:
            n[d] -= Base
            retenue = 1
        else:
            retenue = 0
        d += 1
    while d < len(n) and retenue:
        n[d] += retenue
        if n[d] >= Base:
            n[d] -= Base
            retenue = 1
        else:
            retenue = 0
        d += 1
    if retenue:
        n.append(retenue)
```

Maintenant pour faire la somme de `n` et `m` , il suffit de copier `n` et de lui ajouter `m` sans décaler :

```
def somme(n, m):
    """
    Retourne la somme des nombres n et m

    >>> somme([], [])
    []
    >>> somme([1], [])
    [1]
    >>> somme([], [1])
    [1]
    >>> somme([4,4,5,2],[2,8])
    [6, 2, 6, 2]

    >>> for i in range(1000):
    ....:     for j in range(1000):
    ....:         assert(nbint(somme(intnb(i), intnb(j))) == i + j)
    """
    res = n[:]
    ajoute_decal(res, m, 0)
    return res
```

On écrit de manière similaire la différence (voir le code fourni par l'enseignant, fichier mult.py).

2.3 Multiplication naive

Pour faire la multiplication de deux entiers, on procède comme à la petite école, en multipliant le premier nombre par les chiffres du deuxième nombre et en décalant, puis on ajoute le tout :

```
          36
         x 25
         ----
36*5 :    180
36*2 :   +72
         ====
          900
```

On commence par écrire une fonction qui multiplie un entier par un chiffre :

```
def produit_chiffre(n, c):
    """
    Multiplie le nombre n par un chiffre

    >>> produit_chiffre([4,4,5,2],9)
    [6, 9, 8, 2, 2]
    """
    if c == 0: return []
    if c == 1: return n
    retenue = 0
    res = [0]*len(n)
    for i in range(len(n)):
        resc = n[i]*c + retenue
        res[i] = resc % Base
        retenue = resc // Base
    if retenue:
        res.append(retenue)
    return res
```

Ensuite, il suffit d'ajouter en décalant :

```
def produit_naif(n, m):
    """
    >>> produit_naif([9,9],[9,9])
    [1, 0, 8, 9]
    >>> for i in range(1000):
    ....:     for j in range(1000):
    ....:         assert(nbint(produit_naif(intnb(i), intnb(j))) == i * j)
    >>> a = 129806570567313324234234
    >>> nbint(produit_naif(intnb(a), intnb(a)))
    16849745762446893790131992088457114322497566756
    """
    res = []
    for i in range(len(n)):
        ajoute_decal(res, produit_chiffre(m, n[i]), i)
    return res
```

Note 4 *Ainsi, en utilisant la méthode naive, pour faire le produit d'un nombre à n chiffres par un nombre à m chiffres, on a besoin de faire $n * m$ produit de chiffres.*

2.4 Multiplication rapide de Karatsuba

L'algorithme inventé en 1960 par le russe Karatsuba (Анатолий Алексеевич Карацуба) permet pour les grands nombres d'aller plus vite (voir `https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba`). L'idée est la suivante : le calcul de

$$(a \times 10^k + b)(c \times 10^k + d) \tag{1}$$

qui, sous forme développée s'écrit

$$ac \times 10^{2k} + (ad + bc) \times 10^k + bd \tag{2}$$

semble nécessiter le calcul des quatre produits ac, ad, bc et bd. Pour de grands nombres, en regroupant les calculs sous la forme suivante :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd \tag{3}$$

la méthode peut être appliquée de manière récursive pour les calculs de ac, bd et $(a-b)(c-d)$ en scindant à nouveau a, b, c et d en deux et ainsi de suite. C'est un algorithme de type diviser pour régner.

Exécutons l'algorithme pour calculer le produit 1237×2587.

- Pour calculer, 1237×2587, on écrit $1237 \times 2587 = a_0 10^4 + (a_0 + a_2 - a_1)10^2 + a_2$ où $a_0 = 12 \times 25$, $a_1 = (12-37) \times (25-87) = 25 \times 62$ et $a_2 = 37 \times 87$.
  - Pour calculer 12×25, on écrit $12 \times 25 = a'_0 10^2 + (a'_0 + a'_2 - a'_1)10 + a'_2$ où $a_0 = 1 \times 2$, $a_1 = (1-2) \times (2-5) = -1 \times -3$ et $a_2 = 2 \times 5$.
    - Les calculs $1 \times 2 = 2$, $2 \times 5 = 10$ et $-1 \times -3 = 3$ sont tous des multiplications de chiffres (on ne les redécompose pas). Ils sont faits par les opérations sur les petits entiers.
    - On obtient $12 \times 25 = 2 \times 100 + (2 + 10 - 3) \times 10 + 10 = 300$
- De la même façon, on obtient $a_1 = 25 \times 62 = 1550$.
- De la même façon, on obtient $a_2 = 37 \times 87 = 3219$.
- d'où

$$1237 \times 2587 = 300 \times 100^2 + (300 + 3219 - 1550) \times 100 + 3219 = 3000000 + 196900 + 3219 = 3200119\,.$$

Note 5 *Le calcul complet ne demande que* 9 *produits de deux chiffres au lieu de* $4 \times 4 = 16$ *par la méthode usuelle.*

Bien entendu, cette méthode, fastidieuse à la main, révèle toute sa puissance pour une machine devant effectuer le produit de grands nombres.

En Python, on peut écrire le code :

```
def Karatsuba(n, m, seuil = 10):
    """
    >>> Karatsuba([9,9],[9,9])
    [1, 0, 8, 9]
    >>> for i in range(100):
    ....:     for j in range(100):
    ....:         assert(nbint(Karatsuba(intnb(i), intnb(j))) == i * j)
    >>> a = 129806570567313324234234
    >>> nbint(Karatsuba(intnb(a), intnb(a)))
    16849745762446893790131992088457114322497566756
    """
```

```
# On suppose que n est le plus long.
if len(n) < len(m):
    n, m = m, n
if m == []: return []
if len(m) == 1: return produit_chiffre(n, m[0])
if len(m) <= seuil: return produit_naif(n, m)
cut = len(n) // 2
n1 = n[:cut]; n2 = n[cut:]
m1 = m[:cut]; m2 = m[cut:]
n1m1 = Karatsuba(n1, m1)
n2m2 = Karatsuba(n2, m2)
mid = Karatsuba(somme(n1, n2), somme(m1, m2))
retranche(mid, n1m1)
retranche(mid, n2m2)
res = n1m1
ajoute_decal(res, mid, cut)
ajoute_decal(res, n2m2, 2*cut)
return res
```

3 Notion de complexité

On s'intéresse au temps de calcul des trois fonctions `somme`, `produit_naif` et `Karatsuba`. L'objectif est d'estimer le temps de calcul nécessaire pour faire des opérations sur des nombres à un million de chiffres !

Pour faire la somme de deux nombres à n chiffres, on a n additions de chiffres à faire, il est donc raisonnable de penser que le temps de calcul de la somme va être de l'ordre de grandeur de n fois le temps de calcul de la somme de deux chiffres et va donc suivre une loi de la forme :

$$\text{temps}_{\texttt{somme}}(n) \approx C_{\texttt{somme}} \times n + T_0 \tag{4}$$

où C est une constante (qui dépend de la vitesse de la machine) à trouver. Le temps T_0 correspond au temps des initialisations et est tout petit. On va le négliger et donc ne garder que :

$$\text{temps}_{\texttt{somme}}(n) \approx C_{\texttt{somme}} \times n \tag{5}$$

Pour ce qui est de la multiplication naive, on a vu que pour multiplier deux nombres à n chiffres, il faut faire n^2 multiplications de chiffres. On va chercher une loi de la forme :

$$\text{temps}_{\texttt{naif}}(n) \approx C_{\texttt{naif}} \times n^2 \tag{6}$$

Le cas de la multiplication de Karatsuba est plus compliqué. On va chercher une loi de la forme

$$\text{temps}_{\texttt{Kara}}(n) \approx C_{\texttt{Kara}} \times n^{\alpha} \tag{7}$$

Le but de ce TP est de
- vérifier que les lois théoriques ci-dessus correspondent bien à ce que l'on observe sur une machine.
- de déterminer les valeurs de $C_{\texttt{somme}}$, $C_{\texttt{naif}}$, $C_{\texttt{Kara}}$ et α.
- d'extrapoler pour estimer le temps pris par l'addition et la multiplication de deux nombres à un million de chiffres.

4 Mesure d'un temps de calcul en Python

Pour mesurer le temps d'un calcul en Python, on va utiliser la fonction `time.time()`. Pour ceci, il faut charger le module `time` :

```
>>> import time
```

Ensuite, un appel à `time.time()` donne le nombre de secondes écoulées depuis la date référence `epoch` c'est à dire le 1er janvier 1970 :

```
>>> time.time()
1486144342.05344
```

Pour trouver le temps de calcul pris par la multiplication naive du nombre formé de 100 chiffre 9 par lui même il, il faut donc faire :

```
>>> nb = [9]*100
>>> avant=time.time(); res = produit_naif(nb, nb); apres=time.time()
>>> apres-avant
0.040435075759887695
```

Ici le calcul a pris environ 0.04 s

5 Travail demandé

1. Écrire une fonction `mesure(fun, nb_chiffres)` qui étant donnée une opération `fun` sur les grands nombres retourne le temps de calcul de `fun` sur deux entiers ayant `nb_chiffres` chiffres.
2. Créer une liste `tailles` contenant les nombres de chiffre des nombres que l'on va multiplier. On part de nombres à 10 chiffres jusqu'à des nombres à 2560 chiffres et en multipliant par deux à chaque fois.
3. Mesurer les temps de calcul des trois fonctions `somme`, `produit_naif` et `Karatsuba` pour les nombres de tailles précédentes.
4. Tracer les courbes correspondantes.
5. La seule courbe qu'un être humain est capable de reconnaître à l'œil est une droite. Après avoir commenté les trois tracées, on restreindra à l'étude des deux opérations de multiplication (naive et Karatsuba). Pour chacun des deux modèles correspondants, déterminer une fonction $y(x)$ qui sera une droite si le modèle est vérifié, i.e. dire ce qu'il faut prendre pour y et pour x pour que les données s'alignent sur une droite si le modèle est valide.
6. Pour chacune des deux fonctions, créer le vecteur y et le vecteur x correspondant à partir des données expérimentales.
7. Faire les ajustements de droite (utiliser le module `scipy.optimize` de Python) sur les mesures de temps de calcul obtenues précédemment et en déduire les valeurs de $C_{\texttt{naif}}$, $C_{\texttt{Kara}}$ et α.
8. Faire les tracés de comparaison modèle/données, commenter et extrapoler pour estimer le temps de calcul pour des nombres à 1 million de chiffres.