TP 13 - correction, complexité

1 Un premier exemple

Question 1. La fonction suivante prend en arguments deux naturels a et b. Que renvoit-elle ? Le justifier et en préciser la complexité selon a et/ou b :

```
def mystere(a, b):
    c = 0
    while b > 0:
        c = c + a
        b = b - 1
    return c
```

2 Recherche de minimum

Question 2. écrire une fonction d'entête `def minimum(L):` qui prend en argument une liste L de nombres et qui renvoie le plus petit de ses éléments.

Justifier la correction de votre algorithme, et en préciser la complexité.

3 Produit matriciel

Question 3. Ecrire une fonction `def produitMat(A, B):` qui prend en argument deux matrices A et B de n lignes et n colonnes représentées par des listes de listes de nombres (entiers, flottants, peu importe) et qui renvoie leur produit AB sous la forme encore d'une liste de n listes.

En préciser la complexité, selon l'entier n.

4 Produit de polynômes

Un polynôme à une indéterminée X, à coefficients dans $\mathbb{K}$ et de degré au plus n s'écrit $P = a_0 + a_1 X + \cdots + a_n X^n$. Si Q est le polynôme $b_0 + b_1 X + \cdots + b_n X^n$, alors la somme de P et de Q est le polynôme : $P + Q = a_0 + b_0 + (a_1 + b_1) X + \cdots + (a_n + b_n) X^n$.

Leur produit est le polynôme $PQ = \sum_{k=0}^{2n} c_k X^k$ où $c_0 = a_0\, b_0$, $c_1 = a_0\, b_1 + a_1\, b_0$, $\cdots$, $c_n = a_0\, b_n + a_1\, b_{n-1} + \cdots + a_n\, b_0$, $c_{n+1} = a_1\, b_n + \cdots + a_n\, b_1, \ldots, c_{2n} = a_n\, b_n$.

On représentera en python un polynôme tel que $P = a_0 + \cdots + a_n X^n$ sous la forme de la liste $[a_0, a_1, \cdots, a_n]$.

Question 4. Etant donnés deux polynômes $P = a_0 + a_1 X + \cdots + a_p X^p$ et $Q = b_0 + b_1 X + \cdots + b_q X^q$, alors le produit PQ s'écrit $c_0 + c_1 X + \cdots + c_{p+q} X^{p+q}$.

Etant donné $i \in [\![0, p+q]\!]$, alors c_i s'écrit sous la forme $\sum_{k=\alpha}^{\beta} a_k\, b_{i-k}$. Déterminer α et β selon p, q et i.

Question 5. Ecrire une fonction `def produitPoly(P, Q):` qui admet en arguments deux listes P et Q (représentant donc les coefficients de deux polynômes, du coefficient constant à celui de plus haut degré) et qui renvoie la liste des coefficients du polynôme produit PQ de P et de Q.

En supposant que P et Q soient deux listes de n valeurs, préciser la complexité, selon n, de la fonction que vous venez d'écrire dans le cas où les deux listes P et Q données en argument comportent l'une comme l'autre le même nombre n de termes (autrement dit où P et Q sont de même degré $n-1$)

4.1 Algorithme de Karatsuba

On a longtemps considéré que l'algorithme élémentaire utilisé ci-dessous était optimal, jusqu'à ce que Karatsuba fasse la remarque suivante :

Supposons P et Q de même degré $2\,n-1$ (chacun étant alors formé de $2\,n$ coefficients) et qu'on écrive P ainsi :

$$P = \sum_{k=0}^{2n-1} a_k X^k = \left(\sum_{k=0}^{n-1} a_k X^k\right) + X^n \left(\sum_{k=0}^{n-1} a_{n+k} X^k\right) = P_1 + X^n P_2$$

et qu'on fasse de même pour Q, alors

$$PQ = P_1\,Q_1 + X^n\,(P_1\,Q_2 + P_2\,Q_1) + X^{2n}\,P_2\,Q_2 = P_1\,Q_1 + X^n\,(P_1\,Q_1 + P_2\,Q_2 - (P_1 - P_2)\,(Q_1 - Q_2)) + X^{2n}\,P_2\,Q_2$$

Cette dernière expression semble tortueuse et peu économique, et pourtant elle présente un avantage : au lieu de nécessiter 4 produits de polynômes de degrés au plus $n-1$, 3 suffisent (quelques additions se rajoutent mais, on le devine, additionner deux polynômes est beaucoup moins coûteux que de les multiplier). Bien sûr, on ne s'arrête pas là, et pour calculer chacun de ces trois produits, on découpe chaque polynôme en deux et ainsi de suite.

Vous l'avez compris, on va faire appel à la récursivité. Pour simplifier, les polynômes que l'on multipliera seront toujours formés d'un nombre de coefficients égal à une puissance de 2.

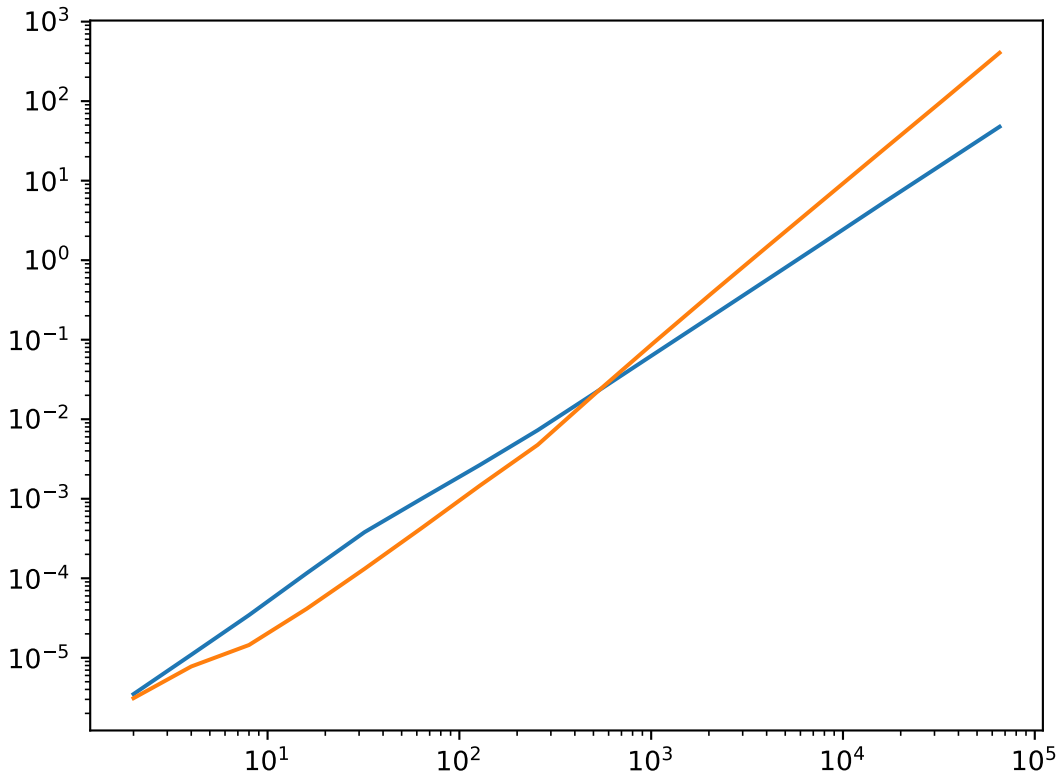
Question 6. Ecrire une fonction d'entête `def soustrait(P, Q):` qui suppose deux polynômes représentés par des listes P et Q de même longueur (on ne le testera pas) et qui renvoie le polynôme $P - Q$.

On n'oublie évidemment pas le cas de base, qui est celui de polynômes P et Q constants (listes de longueur 1) qu'il serait bien difficile de couper en deux ! Dans ce cas, nul besoin de s'invoquer soi-même. Pour tous les autres cas, on découpe P en P_1 et P_2, Q en Q_1 et Q_2 et on calcule les trois produits que sont $P_1\,Q_1$, $P_2\,Q_2$ et $(P_1 - P_2)\,(Q_1 - Q_2)$.

Question 7. Ecrire alors une fonction d'entête `def karatsuba(P, Q):` qui prend en argument deux polynômes P et Q représentés par deux listes de même longueur (supposée être une puissance de 2) et qui renvoie leur produit.

4.2 Etude empirique de la complexité

Voici sur une échelle doublement logarithmique ce que j'ai obtenu : en abscisse le nombre de coefficients des polynômes à multiplier, et en ordonnée le temps écoulé (en secondes) pour une multiplication : une courbe pour l'algorithme élémentaire de multiplication, et l'autre pour l'algorithme de Karatsuba (je vous laisse deviner qui est qui !)



(Les échelles logarithmiques écrasent un peu les différences, mais, comme on s'en doute, l'algorithme de Karatsuba est, asymptotiquement, significativement meilleur que l'algorithme élémentaire).

Question 8. Du graphe précédent, pouvez-vous indiquer une stratégie pour améliorer encore un peu la complexité temporelle de la multiplication de deux polynômes ? (Je suis parvenu à gagner un facteur 2 environ pour ma part. Comment ai-je fait ?)

4.3 Analyse de complexité de l'algorithme de Karatsuba

On note $T(n)$ le temps nécessaire à multiplier deux polynômes de même degré $2^n - 1$.

Question 9. Justifier la formule suivante :

$$T(n+1) = 3\,T(n) + c\,2^n$$

où c est une certaine valeur (qu'on ne cherchera pas à déterminer)

Question 10. On considèrera que $T(0) = 0$. Calculer alors $T(n)$ pour tout n. (Indication : on pourra commencer par chercher une suite de la forme $(\lambda\,2^n)$ solution de la récurrence précédente)

On montrera l'existence de $\alpha \in \mathbb{R}$ tel que pour tout n, $T(n) = \alpha\,3^n - c\,2^n$.

Remarque 1. L'expression obtenue ci-dessus montre que pour multiplier deux polynômes de même degré $N = 2^n - 1$, il faut donc un temps qui, asymptotiquement, est en $\alpha\,3^n$, c'est-à-dire proportionnel à $N^{\frac{\ln 3}{\ln 2}} = N^{\log_2(3)}$. Or $\log_2(3) = 1,585$ à 10^{-3} près par excès, à comparer à un temps d'exécution proportionnel à N^2 pour l'algorithme élémentaire.

5 A faire à la maison

On a vu dans un TP précédent l'algorithme de tri par insertion, dont voici une implémentation en python :

```
def tri_insertion(A):
    for i in range(1, len(A)):
        j, x = i, A[i]
        while j > 0 and A[j-1] > x:
            A[j] = A[j-1]
            j = j - 1
        A[j] = x
```

(Pas de return ici, mais vous aurez noté que la liste fournie en argument est modifiée dans la fonction. On pourrait bien sûr décider de terminer cette fonction par un `return A`)

Question 11. Justifier que si A est une liste d'objets comparables (des nombres entiers ou flottants, des chaînes de caractères par exemple, mais pas de mélange bien sûr) alors après exécution de `tri_insertion(A)` la liste A est contituée des mêmes éléments qu'initialement mais ordonnés dans le sens croissant.

Question 12. Pour une liste A de n termes, combien de comparaisons et d'affectations sont effectuées dans l'exécution de `tri_insertion(A)` dans le meilleur cas ? dans le pire des cas ? Comment qualifier la complexité temporelle de l'algorithme de tri par insertion ?

Remarque 2. Pour rappel : on a vu que le tri fusion a une complexité (en moyenne, dans le meilleur comme dans le pire des cas) logarithmique en $n\ln(n)$ (on parle de complexité quasi-linéaire) et il en va de même, **en moyenne**, pour le tri rapide (mais ce dernier a un talon d'achille : dans le pire des cas, on obtient une complexité quadratique, c'est-à-dire en n^2)

6 Rappels de syntaxe

- Etant donnée une liste `L`, `L.append(v)` modifie `L` pour y adjoindre, à la fin, l'élément `v`.
- Etant donnée une liste `L` de n termes, `L[:a]` est une nouvelle liste formée des termes de `L` jusqu'à celui, non compris, d'indice `a`, `L[a:]` est quant à elle formée des termes de `L` de l'indice `a` jusqu'à la fin. A noter qu'on n'obtient jamais ici d'erreur de type `out of range`, mais que par exemple si $a > n$, alors `L[n:]` sera la liste vide.

  (Bien sûr, on n'en aura pas besoin dans ce TP mais on peut spécifier à la fois un indice de départ (compris) et d'arrivée (non compris) avec `L[a:b]`, voire un pas avec `L[a:b:pas]`)