

Devoir surveillé n°2 - Jeudi 8 octobre 2026

Rédiger sur une copie double. La calculatrice n'est pas autorisée.

Consignes de présentation :

- Nom Prénom lisibles en haut à gauche, classe en haut à droite, DS n°2 au milieu et encadré à la règle.
- Vous laissez ensuite 5 ou 6 centimètres pour mes éventuels commentaires, délimités par deux traits tracés à la règle.
- S'il n'y a pas de marge déjà tracée sur votre feuille, vous en tracez une à 4 cm du bord de la feuille.
- Les codes Python demandés doivent être écrit très lisiblement, avec une indentation très claire.

Exercice 1 : Pour calculer la puissance a^n d'un nombre réel $a \in \mathbb{R}$ par un entier naturel $n \in \mathbb{N}$, l'approche élémentaire consiste à effectuer $n - 1$ multiplications successives par a . Pour de grandes valeurs de n , cette méthode devient inefficace.

L'algorithme d'**exponentiation rapide** repose sur le principe « diviser pour régner » en exploitant les propriétés suivantes de la puissance :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(a^{n/2}\right) \times \left(a^{n/2}\right) & \text{si } n \text{ est pair et } n > 0 \\ a \times a^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

1. Écrire en Python une fonction réursive `puissance(a,n)` qui implémente cet algorithme.
2. Dérouler à la main la suite des appels récursifs lors de l'exécution de l'appel `puissance(2,10)`. Indiquer précisément le nombre de multiplications effectuées.

Exercice 2 : On s'intéresse aux matrices parcimonieuses, c'est-à-dire aux matrices dont la plupart des coefficients sont nuls. Une telle matrice M de taille (n, p) pourra être codée par un dictionnaire ayant pour couples clefs/valeurs :

- `'dim': (n,p)`
- `(i, j): Mij` pour chaque couple (i, j) tel que $M_{ij} \neq 0$.

Par exemple la matrice $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 \end{pmatrix}$ sera codée par le dictionnaire `{ 'dim': (2,4), (1,2):15 }`

1. Proposer une fonction d'addition de deux matrices parcimonieuses (que l'on suppose de même dimension). Les deux matrices données en argument seront de type dictionnaire comme expliqué ci-dessus, et le résultat renvoyé devra aussi être un dictionnaire de cette forme.
2. Si la première matrice contient c_1 coefficients non nuls, et la seconde c_2 , quelle est la complexité de cet algorithme en fonction de c_1 et c_2 ?

Exercice 3 : Le but de cet exercice est d'écrire une fonction qui prend en argument une liste L composée de 0 et de 1 et qui renvoie l'entier correspondant à la longueur de la plus grande sous-liste de L composée uniquement de 1. Par exemple, avec la liste $[0,1,0,1,1,1,0,1,0,1,1,1,1]$, la fonction devra renvoyer 4. Dans cet exercice, la seule fonction préprogrammée autorisée est la fonction `len`, pour connaître la longueur d'une liste.

1. Première méthode

- (a) Ecrire une fonction `test_longueur`, qui prend en argument une liste **supposée non vide et commençant par 1**, qui retourne le nombre de 1 consécutifs dans la liste en partant du début. Par exemple, `test_longueur([1,1,1,0,0,1,1,1,1])` doit renvoyer 3.
- (b) Déterminer, en fonction de la taille n de la liste passée en argument, la complexité de cette fonction dans le pire des cas.
- (c) En utilisant la fonction `test_longueur` précédemment créée, écrire une fonction `max_longueur` répondant au problème exposé au début de l'exercice.
On rappelle à cet effet le principe du slicing : La syntaxe $L[i:j]$ extrait la sous-liste composée des éléments d'indices allant de i inclus à j exclu : $[L[i], L[i+1], \dots, L[j-1]]$.
- (d) Déterminer, en fonction de la taille n de la liste passée en argument, la complexité de la fonction `max_longueur` dans le pire des cas.

2. Deuxième méthode

Ecrire une fonction `max_longueurV2`, qui soit **de complexité linéaire**, qui réponde au problème posé au début de l'énoncé.

Exercice 4 (*Bonus pour ceux qui auraient tout fini*) :

On reprend l'exercice 1 et le programme d'exponentiation rapide, les questions suivantes font suite aux deux premières questions traitées dans l'exercice 1 :

- 3. Déterminer le nombre de multiplications nécessaires pour calculer a^{1000} avec cette méthode. Comparer au nombre de multiplications de la méthode naïve.
- 4. Déterminer la complexité temporelle $C(n)$ de cet algorithme en notation $\mathcal{O}$.
- 5. Un étudiant propose le code suivant pour le cas pair :

```
1 | return puissance(a, n // 2) * puissance(a, n // 2)
2 |
```

Expliquer la maladresse commise et préciser l'impact de cette modification sur la complexité temporelle globale.