

Directives

- Décomposer le code en fonctions autonomes pour que votre travail puisse être évalué même si votre réponse est incomplète.
- L'utilisation d'un notebook est indispensable. Séparer les cellules de déclaration de procédure ou fonction avec une cellule de test du code sur un exemple.
- Votre notebook "prenom_nom.ipynb" est à envoyer à "mohammed.sedki@gustaveroussy.fr".

I. Cryptage d'un mot

Écrire un programme en PYTHON permettant de crypter un MOT comme suit :

- Saisir un MOT de 25 caractère maximum et non vide, formé uniquement par des lettres alphabétiques majuscules.
- Remplir puis afficher une matrice M carrée par des entiers aléatoire entre 1 et 25. (nombre de lignes = nombre de colonnes = nombre des caractères du MOT).
- Former la chaîne cryptée en remplaçant chaque lettre du MOT par celle qui la suit de p positions dans l'alphabet français, où p désigne sa position dans la diagonale de la matrice M .
- Afficher le MOT crypté.

Un exemple de cryptage d'un mot est donné par:

MOT = "SALUT"

Le MOT crypté est : LDSMI

Quelques indications

- On suppose que le caractère qui suit la lettre "Z" est le caractère "A"
- Le code ASCII de la lettre "A" est égal à 65 et "Z" est égale à 90.
- Faire appel aux fonctions **ord** et **chr** natives de PYTHON.

II. Cryptage d'une séquence de mots

Écrire un programme qui permet de remplir un tableau t par n mots ($2 < n < 100$), ces mots ayant une longueur maximale de 20 caractères, on désire les crypter selon le procédé suivant :

- Le premier caractère du mot est considéré comme clé de cryptage.
- Chaque caractère du mot sera codé en ajoutant son code ASCII au code ASCII de la clé. Si cette somme dépasse 127, on en soustrait 127.
- Déterminer le caractère dont le code ASCII est égal à cette somme.

Un exemple de cryptage est donné par :

```
['alerte' 'incendie' 'foret']  
['CNGTVG', 'SXMOXNSO', 'MVYL']
```

Quelques indications

Éviter les exemples avec accentuation. On peut faire appel aux fonctions **ord** et **chr** natives de PYTHON. La qualité du code sera prise en compte.

III. Algorithmes de tri : un peu de complexité

Tri par insertion

Nous avons étudié en cours l'algorithme de tri par insertion qui peut être résumé sur un exemple simple, comme suit

Voici les étapes du tri par insertion de $\text{Tab}=[2, 3, 1, 6, 4, 5]$.

Étape	Tab	Commentaire
0	[2, 3, 1, 6, 4, 5]	Le début [2] est déjà trié. Rien ne change.
1	[2, 3, 1, 6, 4, 5]	3 est déjà à sa place. Rien ne change.
2	[1, 2, 3, 6, 4, 5]	On insère 1 à sa place dans le début [2, 3].
3	[1, 2, 3, 6, 4, 5]	6 est déjà à sa place. Rien ne change.
4	[1, 2, 3, 4, 6, 5]	On insère 4 à sa place dans le début [1, 2, 3, 6].
5	[1, 2, 3, 4, 5, 6]	On insère 5 à sa place dans le début [1, 2, 3, 4, 6].

Tri par fusion : principe de l'algorithme

Le tri par fusion (*merge sort*) est un algorithme de tri efficace qui a pour complexité $\mathcal{O}(n \ln n)$, il se base sur le principe du diviser pour régner. Voici un petit exemple pour décrire les deux étapes sur un exemple de tri de la suite de nombres 5, 1, 3, 8, 9, 6

1ère étape : diviser

5, 1, 3 | 8, 9, 6 -> on divise le tableau en deux.

5, | 1, 3 | 8 | 9, 6 -> on divise en deux les sous tableaux.

5 | 1 | 3 | 8 | 9 | 6 -> chaque sous tableau est de nouveau divisé pour n'avoir plus qu'un seul élément.

2ème étape : fusionner

1, 5 | 3, 8 | 6, 9 -> on prend deux sous tableaux adjacents que l'on fusionne en les ordonnant.

1, 3, 5, 8 | 6, 9 -> on continue la fusion des sous tableaux.

1, 3, 5, 6, 8, 9 -> le tableau ne contient plus de sous tableaux, il est donc trié.

1, 3, 5, 6, 8, 9

1. Implémenter une fonction **tri_insertion** qui permet de trier par insertion une liste de nombres.
2. Implémenter une fonction **tri_fusion** qui permet de trier par fusion une liste de nombres.
3. Proposer une comparaison des temps de calcul des deux fonction précédentes avec la fonction **sorted** native de PYTHON et la méthode **sort** de la classe **list** native de PYTHON.

Quelques indications

- On peut faire appel à la fonction **randint** du module **random** pour générer aléatoirement des listes de nombres entiers.
- On peut utiliser la fonction **timeit** du module **timeit** pour calculer le temps d'exécution d'un appel à une fonction répété un certain nombre de fois.
- Afficher les temps (4 fonctions).