Exercice 1 :

> Essayer de réactiver sa mémoire avant de regarder le code vu en classe.

1. Écrire une classe `Maillon` avec deux attributs :
   - `valeur: int`
   - `suivant: Maillon`
2. Écrire une classe `Liste` avec un attribut `tete: Maillon` initialisé à `None`.
3. Écrire la méthode `ajouter(self, val: int)` → `None` qui ajoute `val` à la liste.
4. Instancier la liste et la remplir avec 10 entiers.
5. Écrire la méthode *impérative* `dernier` qui renvoie le dernier élément de la liste. La méthode renverra `-1` si la liste est vide.
6. Proposer une version récursive de la méthode précédente.
7. Sur papier, schématiser l'algorithme qui renverse l'ordre de la liste.
8. Écrire alors la méthode *impérative* `renverser`.
9. Écrire la méthode *impérative* `dupliquer` qui double chaque élément dans la liste. Par exemple : `1 - 2 - 3` devient `1 - 1 - 2 - 2 - 3 - 3`.

Exercice 2 :

1. Dans un nouveau fichier, importer les classes `Liste, Maillon` créées dans l'exercice précédant.
2. Créer deux listes `l1, l2` et ajouter au moins trois entiers dans chaque liste.
3. Écrire une fonction `concatener(l1: Liste, l2: Liste)` → `Liste` qui renverra une `Liste`, concaténation des deux passées en paramètre. Cette fonction contiendra une fonction interne `concatener_rec(tete1: Maillon, tete2: Maillon)` → `Maillon` qui implémentera le principe de la figure 1. On remarquera en particulier que cette fonction ne copie pas la `Liste` 2.

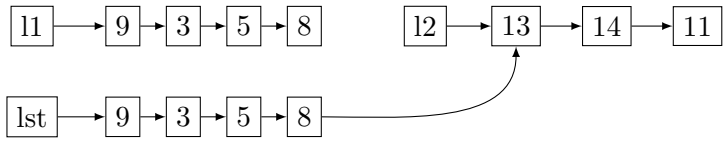

FIGURE 1 – Fonctionnement de **concatener_rec**

4. Quel problème peut-on prévoir lors de la modification d'un élément de la `Liste` ?
5. Estimer la complexité de la concaténation.

Exercice 3 : On décide de créer une liste chaînée à l'aide de tuples.

```
lst = ("a", ("b", ("c", ("d", ()))))
```

Code 1 – "a" est la tête de la liste

1. Écrire la fonction `longueur(lst: tuple)` → `int` qui renvoie la taille de la liste.
2. Écrire la fonction `afficher(lst: tuple)` → `str` qui renvoie une chaîne de caractère de la forme

```
a - b - c - d - fin
```


Christophe Viroulaud