TP 1 : Benchmark de structures de type listes

Pré-requis

— Cours d'introduction aux structures de données et aux listes
— Disposer d'un compte GitLab (ou GitHub selon votre usage)

Objectifs du TP

Dans ce TP, nous allons implémenter une **liste chaînée** (`TPList`) et plusieurs opérations fondamentales associées. Nous comparerons ensuite ses performances à celles des listes Python standards à l'aide de benchmarks simples.

Sur mon GitHub, dans le répertoire `enseignement`, vous trouverez un dossier nommé `TP1_listes`. Clonez ce dépôt.

Vous devrez :
— joindre le fichier `README_tp1_listes.md` à votre dépôt GitHub (certaines réponses y seront rédigées),
— ajouter l'ensemble du code demandé.

Nota bene :
Il est **interdit d'utiliser les structures `set` ou `dict` dans ce TP**, car elles seront étudiées ultérieurement :)

Évaluation

Les TP sont évalués **par groupes de 2 étudiants**, exceptionnellement 3.

Partie écrite

L'évaluation se fait selon les critères suivants (dans l'ordre) :

— Chaque groupe dispose d'un dépôt GitHub avec un nom explicite (par exemple : `TP1_SD_groupeX`)
— Les fichiers suivants sont présents dans le dépôt :
  — `README_tp1_listes.md`
  — `tpList.py`
— Les questions sont correctement traitées dans `README_tp1_listes.md`
— Chaque membre du groupe a effectué plusieurs commits
— Des tests sont présents lorsqu'ils sont demandés
— L'exécution de `python tpList.py` ne génère **aucune erreur**
— Le code est correctement commenté lorsque nécessaire

— Le code est fonctionnel et correct
— L'exécution du code génère un ou plusieurs graphiques (`.png` ou `.pdf`)
— Les graphiques sont corrects et exploitables
— Les réponses écrites sont justes et cohérentes
— Une date de rendu est fixée collectivement :
les commits postérieurs à cette date ne seront pas pris en compte

Partie orale

Pendant les séances, je vous interrogerai sur :
— votre compréhension du TP,
— votre avancement,
— vos choix d'implémentation.

Questions

0 - Découverte du code fourni

Ouvrez le fichier `TPList.py`.
Il contient du **code à trous** que vous devrez compléter pour répondre au TP.

Commencez par examiner la classe `Node` :
— elle représente un nœud d'une liste chaînée,
— chaque nœud contient :
  — une donnée (`self.data`),
  — une référence vers le nœud suivant (`self.next`).
Examinez ensuite la classe `TPList`, qui représente la liste chaînée elle-même.

Dans les commentaires, certaines lignes commencent par `>>>` : ce sont des **doctests**.
Python peut les exécuter automatiquement pour vérifier le comportement du code.

Un doctest est de la forme :

```
>>> resultat = ligne_de_code
resultat_attendu
```

Python vérifie alors que `resultat == resultat_attendu`.

— Si tous les doctests passent, aucun message ne s'affiche lors de l'exécution.
— Sinon, des messages d'erreur sont affichés.

De nombreux tests sont déjà présents.
Ils échoueront au début tant que vous n'aurez pas implémenté les méthodes demandées : c'est normal.

La méthode `__init__(self)` initialise une liste vide.

Si vous exécutez `python TPList.py` maintenant, vous devriez voir apparaître de nombreuses erreurs liées aux tests.

1 - Méthode `append`

Implémentez une méthode :

```
append(self, data)
```

qui ajoute un élément à la fin de la liste chaînée.

Des doctests sont déjà fournis pour cette méthode.

2 - Méthode `get`

Implémentez une méthode :

```
get(self, index)
```

qui renvoie la valeur stockée à un index donné
(équivalent de `ma_liste[index]` pour une liste Python).

3 - Doctests pour `get`

Rédigez des doctests pour la méthode précédente.

Indiquez les cas auxquels il faut penser (indices valides, invalides, liste vide, etc.).

4 - Méthode `delete`

Implémentez une méthode :

```
delete(self, index)
```

qui supprime l'élément situé à l'index donné.

Ajoutez les doctests correspondants.

5 - Méthodes spéciales : `__iter__` et `__len__`

Une méthode `__iter__(self)` est déjà implémentée.

Les doubles underscores (`__`) indiquent une **méthode spéciale** de Python, permettant de personnaliser le comportement des objets.

La méthode `__iter__` définit un **itérateur**, ce qui permet notamment :
- d'utiliser une boucle `for` sur votre `TPList`,
- d'itérer efficacement sur ses éléments.

Surchargez maintenant la méthode spéciale :

```
__len__(self)
```

afin que `len(ma_TP_list)` fonctionne correctement.

Ajoutez les doctests correspondants.

6 - Génération et insertion de k-mers (liste Python)

Dans le `main`, deux fonctions sont déjà fournies :

— une qui génère une séquence aléatoire de nucléotides,
— une qui extrait les k-mers d'une séquence dans une liste Python nommée `kmers`.

Faites croître la taille de la séquence jusqu'à **10 000 bases**.

À l'aide de `time.perf_counter()`, mesurez le temps nécessaire pour insérer les k-mers dans la **liste Python**.

7 - Insertion dans la TPList

Ajoutez un code équivalent pour insérer les k-mers dans votre `TPList`.

Mesurez également le temps d'exécution.

8 - Recherche sans boucle `for` (TPList)

Écrivez une fonction :

```
def contient_TP(L: TPList, e: str):
```

qui vérifie si un élément `e` est présent dans une liste chaînée `L`.

La boucle `for` est interdite.
Vous devez parcourir la liste à l'aide de sa structure (pointeurs).

9 - Recherche générique

Écrivez une fonction :

```
def contient(L, e):
```

qui vérifie si `e` est dans `L`, que `L` soit :
— une liste Python,
— ou une `TPList`.

10 - Surchage et itérateur

À quoi nous a servi la surcharge de `__iter__(self)` ?

11 - Benchmark : liste Python

Passez :
— la liste Python `kmers`,
— le k-mer `lookup`
à la fonction `contient`, et mesurez le temps d'exécution.

12 - Benchmark : TPList

Répétez l'expérience précédente :
— avec `contient`,
— avec `contient_TP`,
— sur votre `TPList`.

13 - Fonction `boucle_naive`

Décommentez la fonction `boucle_naive`.
- Que fait-elle ?
- Testez-la avec `kmers` et avec votre `TPList`.
- Mesurez les temps d'exécution.

14 - Question théorique

À quoi nous a servi la surcharge de `__len__(self)` ?

15 - Analyse

En observant les résultats de `boucle_naive`, que remarquez-vous ?
Faites une courte recherche pour expliquer simplement ce comportement.

16 - Étude de complexité expérimentale

Faites varier la taille des séquences (au moins 4 ou 5 tailles différentes).

Pour chaque taille, collectez les temps d'exécution des fonctions :
- `contient`
- `contient_TP`
- `boucle_naive`

pour les deux types de structures (liste Python et TPList).

Écrivez une fonction qui :
- prend ces temps en entrée,
- affiche un graphique montrant l'évolution des temps d'exécution en fonction du nombre de k-mers.

L'exécution du code doit produire un graphique (`.png` ou `.pdf`).

17 - Interprétation

Au regard de ce que nous avons vu en cours, quels sont vos commentaires sur cette figure ?

18 - Choix expérimental

Pourquoi avoir choisi un k-mer `lookup` arbitraire dans une séquence générée aléatoirement ?
Répétez les expériences de recherche de k-mers : avec des k-mer présent très tôt dans la liste, puis en fin de liste dans une autre expérience. Commentez.

19 - Bonus

Décommentez le code `#bonus`.

Il s'agit également d'une recherche de k-mer.
Bonus pour le ou la premier·e étudiant·e ayant terminé le TP et venant m'expliquer le comportement observé sur les temps d'exécution.