Partiel (2 heures)

Remarques :
- Tous les documents (cours, TD, TP en particulier) sont autorisés.
- Les deux exercices valent environ le même nombre de points. Ne restez pas bloqués trop longtemps sur une question.
- Les trois parties de l'exercice 2 sont assez indépendantes. N'hésitez pas à passer à la suivante si vous êtes bloqués ou ne comprenez vraiment pas une partie.

Exercice 1. *Fonctions récursives*

Écrivez les fonctions suivantes de manière récursive en n'utilisant que les primitives `liste()`, `vide(l)`,`cons(x, l)`,`tete(l)` et `queue(l)` vues en cours et en TD pour manipuler les listes. On suppose que les listes contiennent des entiers.
Remarque : Vous pouvez définir des fonctions annexes (avec d'autres arguments) si vous en avez besoin et les utiliser par la suite dans vos fonctions principales.

1. `millieme(l)` qui renvoie le millième élément de la liste (on suppose que la liste a au moins 1000 éléments) ;
2. `carres(l)` qui renvoie la liste obtenue en élevant tous les éléments de l au carré (sur l'entrée 1, 2, 3 la fonction renvoie 1, 4, 9) ;
3. `max(l)` qui renvoie le plus grand élément de l ;
4. `trois(l)` qui teste si tous les éléments de la liste sont des multiples de 3 ;
5. `plus_grand(l1, l2)` qui compare les éléments des listes l_1 et l_2 (qu'on suppose de même taille) deux à deux et renvoie `True` si chaque élément de l_1 est plus grand que l'élément de l_2 en même position et `False` sinon ;
6. `premier(l)` qui teste si le premier élément est le plus grand élément de la liste ;
7. `range(x, y)` qui renvoie une liste contenant les entiers de x à $(y-1)$;
8. `ajouter_precedents(l)` qui renvoie la liste obtenue en ajoutant à chaque élément de l les éléments qui le précèdent (par exemple, sur l'entrée 1, 2, 5, 3 la fonction renvoie la liste 1, $(1+2)$, $(1+2+5)$, $(1+2+5+3)$ c'est-à-dire $1, 3, 8, 11$) ;
9. `difference(l1, l2)` qui renvoie la liste obtenue en supprimant de l_1 tous les éléments qui apparaissent dans l_2 (par exemple avec $l_1 = 1, 2, 4, 2, 5, 6$ et $l_2 = 3, 5, 2$ la fonction doit renvoyer la liste $1, 4, 6$).
10. `miroir(l1, l2)` qui teste si la liste l_2 est le miroir de la liste l_1 (le premier élément de l_1 est le dernier de l_2, le second de l_1 est l'avant-dernier de l_2, etc.) ;

Exercice 2. *Listes bilatères*

Le but de cet exercice est d'étudier différentes structures de données permettant de représenter des *listes bilatères*, c'est-à-dire des listes dans lesquelles on peut ajouter, lire et supprimer des éléments aux deux bouts. On veut donc pouvoir effectuer facilement les opérations suivantes :
- `liste()` qui créé une nouvelle liste ;
- `vide(l)` qui teste si une liste est vide ;
- `push_g(x, l)` et `push_d(x, l)` qui ajoutent l'élément x à la liste l respectivement à gauche (au début) et à droite (à la fin) ;
- `pop_g(l)` et `pop_d(l)` qui enlèvent l'élément respectivement le plus à gauche et le plus à droite de la liste et renvoient sa valeur.

I. Listes simplement chaînées

On représente dans cette partie les listes bliatères par des listes simplement chaînées comme celles qui ont été vues en cours :

```
class Liste:
  pass
class Noeud:
  pass
def liste():
  l = Liste()
  l.premier = None
  return l
def noeud(x):
  n = Noeud()
  n.valeur = x
  n.suivant = None
  return n
```

1. Écrivez les fonctions `push_g(x, l)` et `pop_g(l)` (qui devraient être très simples à écrire, il n'y a pas de piège). Quelle est la complexité de ces fonctions dans le pire des cas sur une liste de taille n ?

2. Écrivez les fonctions `push_d(x, l)` et `pop_d(l)` (vous pouvez utiliser une boucle `while` pour trouver le dernier nœud de la liste).
Remarque : Pour la fonction `push_d`, pensez à traîter le cas particulier où la liste l est vide.

3. Quelle est la complexité des deux fonctions précédentes ?

II. Listes doublement chaînées

Afin de simplifier les fonctions agissant sur la fin de la liste, on considère maintenant des listes doublement chaînées (comme vues en TD) :

```
class Liste:
  pass
class Noeud:
  pass
def liste():
  l = Liste()
  l.premier = None
  l.dernier = None
  return l
def noeud(x):
  n = Noeud()
  n.valeur = x
  n.suivant = None
  n.precedent = None
  return n
```

Ici, chaque nœud de la liste a non seulement un pointeur vers son successeur mais également un pointeur vers son prédécesseur. De plus chaque liste peut directement accéder à son premier et son dernier élément (la structure est totalement symétrique).

4. Écrivez les fonctions `push_d(x, l)` et `pop_d(l)` sur les listes doublement chaînées.
Remarque : Il n'y a plus de boucle `while` à faire, mais il faut changer plus de pointeurs que dans le cas des listes simplement chaînées. Il faut aussi toujours traîter le cas particulier de la liste vide dans la fonction `push_d` et le cas où la liste devient vide pour la fonction `pop_d`.

5. Quelle est la complexité des fonctions `push_d` et `pop_d` sur les listes doublement chaînées ?

III. Tableaux

On représente maintenant les listes par une structure contenant un grand tableau de taille fixe (un entier `MAX` dont la valeur est choisie pour être plus grande que toutes les listes qu'on considérera) et un entier `taille` indiquant le nombre d'éléments dans la liste, comme vu en cours.
Les éléments de la liste sont stockés dans les cases de 0 (le premier élément, qui est à gauche de la liste) à `taille` $-$ 1 (le dernier, qui est le plus à droite).
On utilise donc les définitions suivantes :

```
class Liste:
  pass
def liste():
  l = Liste()
  l.tableau = [None] * MAX
  l.taille = 0
  return l
```

6. Écrivez les fonction `push_d(x, l)` et `pop_d(l)`. Quelle est leur complexité ?

7. Écrivez les fonctions `push_g(x, l)` et `pop_g(l)`.
Remarque : Lorsque l'on ajoute ou qu'on enlève le premier élément du tableau, il faut décaler toutes les autres valeurs (vers la droite ou vers la gauche).

8. Quelle est la complexité des fonctions précédentes dans le pire des cas ?

IV. Tableaux circulaires

Imposer qu'une extrémité de la liste se trouve sur la case d'indice 0 du tableau force à décaler toutes les valeurs à chaque fois qu'on modifie cette extrémité. Pour résoudre ce problème, on peut utiliser des tableaux circulaires, comme ceux qui ont été vus en cours pour les files.
Une liste est donc maintenant représentée par un tableau de taille fixe et deux indices, un indice `debut` et un indice `fin`. Au lieu de placer les valeurs de la liste dans le tableau à partir de l'indice 0, on les place à partir de l'indice `debut` (qui peut être quelconque). Toutes les autres valeurs de la liste sont ensuites placées dans le tableau sur les cases à droite de la case `debut` jusqu'à la case d'indice `fin` $-$ 1 (la liste est donc de longueur `fin` $-$ `debut`).
Afin de pouvoir avancer les indices `debut` et `fin` dans le tableau indéfiniment, on considère tous les indices modulo la taille du tableau (`MAX`). Ainsi, on dira que la case qui se trouve à droite de la dernière case du tableau (`MAX` $-$ 1) est la case 0.
On utilisera donc les définitions suivantes :

```
class Liste:
  pass
def liste():
  l = Liste()
  l.tableau = [None] * MAX
  l.debut = 0
  l.fin = 0
  return l
```

9. Écrivez les fonctions `push_g(x, l)` et `pop_g(l)` sur cette structure de tableaux circulaires.
Remarque : Pour calculer les indices modulo `MAX` vous pouvez utiliser la syntaxe `i % MAX`.

10. Quelle est la complexité de ces fonctions ?

11. Parmi les quatre structures de données étudiées, lesquelles vous semblent les plus adaptées pour représenter les listes bilatères ?