Examen - 2 heures

Les documents (cours, TD, TP) sont autorisés.
Rappel : Pour ceux qui ne l'ont pas encore fait, le DM doit être envoyé par mail au plus tard ce soir à minuit. Pénalité de 4 points par jour de retard (le premier jour de retard commence ce soir à minuit une, il est donc vivement déconseillé de tenter le diable en essayant d'envoyer le DM « juste avant » minuit...)

Exercice 1. *Fonctions récursives*

✎ Écrivez les fonctions suivantes de manière récursive en n'utilisant que les primitives `liste()`, `vide(l)`, `tete(l)`, `queue(l)` et `cons(x,l)` vues en cours et en TD :
- `plus_souvent(x,y,l)` qui renvoie `True` si l'élément `x` apparaît plus de fois que l'élément `y` dans la liste `l` et `False` sinon ;
- `millieme(l)` qui renvoie le millième élément de la liste `l` (on suppose que la liste `l` a au moins mille éléments donc il n'est pas nécessaire de traiter le cas d'erreur) ;
- `facto_liste(l)` qui renvoie la liste contenant la factorielle de chacun des éléments de la liste `l` (si `l` contient les valeurs 4, 3 et 2, la fonction doit renvoyer une liste contenant les valeurs 24, 6 et 2) ;
- `produit(l1,l2)` qui renvoie la liste contenant les produits des éléments de `l1` et `l2` que l'on suppose être de même longueur (si `l1` contient les entiers 1, 2 et 4 et `l2` contient les entiers 2, 3 et 4, la fonction doit renvoyer une liste contenant les valeurs 2, 6 et 16) ;
- `double(l)` qui renvoie `True` si chaque élément de la liste est supérieur au double de son prédécesseur, et `False` sinon (par exemple sur la liste 2, 5, 10 et 23 la fonction renverra `True` mais sur la liste 2, 4, 9, 12 et 25 elle renverra `False` car 12 est inférieur au double de 9) ;
- `multiplie(l)` qui renvoie le produit de tous les éléments de `l`.

R.

```
def plus_souvent(x, y, l):
  """on définit une fonction annexe qui compte la différence entre le
  nombre de x et le nombre de y rencontrés dans la liste"""
  def aux(x, y, l, dif):
    if vide(l):
      return dif >= 0
    elif tete(l) == x:
      return aux(x, y, queue(l), dif+1)
    elif tete(l) == y:
      return aux(x, y, queue(l), dif-1)
    else:
      return aux(x, y, queue(l), dif)
  return aux(x, y, l, 0)

def millieme(l):
  """on définit une autre fonction qui prend en argument un entier i et
  renvoie le i-eme élément de l"""
  def ieme(l, i):
    if i == 1:
      return tete(l)
    else:
      return ieme(queue(l), i-1)
  return ieme(l, 1000)

def factorielle(n):
```

```
    if n == 0:
      return 1
    else:
      return n * factorielle(n-1)
  def facto_liste(l):
    if vide(l):
      return l
    else:
      return cons(factorielle(tete(l)), facto_liste(queue(l)))

  def produit(l1, l2):
    if vide(l1):
      return liste()
    else:
      return cons(tete(l1)*tete(l2), produit(queue(l1), queue(l2)))

  def double(l):
    if vide(l) or vide(queue(l)):
      return True
    else:
      return 2*tete(l) <= tete(queue(l)) and double(queue(l))

  def multiplie(l):
    if vide(l):
      return 1
    else:
      return tete(l) * multiplie(queue(l))
```

Exercice 2. *Listes circulaires*

Une liste circulaire est une liste dans laquelle le successeur du dernier élément est le premier élément de la liste (si l'on avance dans la liste en suivant le successeur de chaque nœud, on finit par retomber sur le premier nœud).

I. Listes simplement chaînées

Une première méthode pour représenter les listes circulaires est d'utiliser des listes chaînées. La définition des classes utilisées est très proche de la définition des listes chaînées « simples » à ceci près que l'on ne permettra pas à une liste d'être vide (il n'y a plus de pointeur vers `None` dans les listes circulaires). Lorsque l'on créé une nouvelle liste circulaire, il faut donc lui donner au moins un nœud.
On peut utiliser les classes suivantes :

```
class Noeud:
  def __init__(self, x):
    self.valeur = x
    self.suivant = self

class Liste:
  def __init__(self, x):
    n = Noeud(x)
    self.premier = n
```

La figure 1 représente une liste circulaire. Le carré isolé en haut est l'objet de type `Liste`, il contient un unique champ `premier` qui pointe vers le premier nœud de la liste. Les rectangles à deux cases sont les nœuds. Ils contiennent un champ `valeur` contenant un entier et un champ `suivant` qui pointe vers le nœud suivant. On notera le fait que le champ `suivant` du dernier nœud pointe vers le premier nœud.

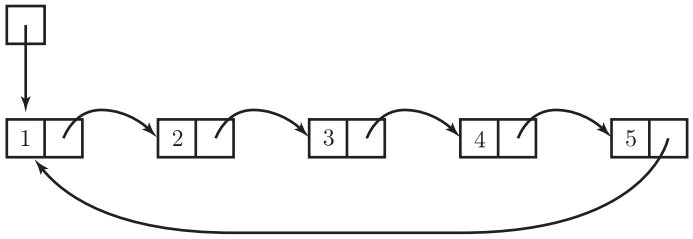

FIGURE 1 – Un exemple de liste circulaire.

1. Expliquez ce que signifie la ligne `self.suivant = self` dans la définition de la classe `Noeud`. Représentez par un schéma le nœud créé par l'instruction `n = Noeud(3)`.

R. Dans le cas d'une liste circulaire, un nœud a toujours un successeur non vide. Quand on créé un nouveau nœud, s'il n'est relié à personne d'autre, il est le seul nœud d'une liste à un élément, et il est donc son propre successeur.

2. Écrivez la fonction `ajoute_tete(x,l)` qui ajoute un nœud contenant la valeur `x` en tête d'une liste circulaire `l` (le premier nœud de la nouvelle liste doit contenir `x` et la liste doit rester une liste circulaire).
Indication : il faut à un moment trouver le dernier nœud de la liste. Il faut pour cela faire une boucle `while` qui parcourt la liste jusqu'à trouver le dernier nœud. On le reconnaît au fait que son successeur est le premier nœud de la liste...

R.

```
def ajoute_tete(x, l):
  n1 = l.premier  # le premier noeud de la liste
  nd = l.premier
  while nd.suivant != n1: # on avance dans la liste jusqu'à ce que n soit
  # le dernier noeud
    nd = nd.suivant
  n = Noeud(x)
  l.premier = n
  n.suivant = n1
  nd.suivant = n
```

3. Écrivez la fonction `ajoute_queue(x,l)` qui ajoute un nœud contenant la valeur `x` à la fin de la liste circulaire `l` (on rajoute le nouveau nœud après le dernier nœud de `l` et l'on indique que le premier nœud de `l` est le successeur du nouveau nœud).

R.

```
def ajoute_queue(x, l):
  n1 = l.premier
  nd = l.premier
  while nd.suivant != n1:
    nd = nd.suivant
  n = Noeud(x)
  n.suivant = n1
  nd.suivant = n
```

4. Quelle est la complexité des deux fonctions précédentes en fonction du nombre n de nœuds dans la liste ?

R. Dans chacune des fonctions, il faut parcourir toute la liste pour trouver le dernier nœud de la liste. La complexité des fonctions est donc $O(n)$.

5. Écrivez une fonction `affiche(l)` qui affiche les éléments d'une liste circulaire en s'arrêtant lorsqu'elle revient sur le premier nœud.

R.

```
def affiche(l):
  n1 = l.premier
  n = l.premier
  while n.suivant != n1:
    print n.valeur
    n = n.suivant
  print n.valeur
```

6. Écrivez une fonction `longueur(l)` qui renvoie la longueur d'une liste, c'est-à-dire le nombre de nœuds qu'elle contient.

R.

```
def longueur(l):
  n1 = l.premier
  n = l.premier
  longueur = 1
  while n.suivant != n1:
    longueur += 1
    n = n.suivant
  return longueur
```

7. Écrivez une fonction `cycle(l)` qui « décale » la liste circulaire `l` d'un cran, c'est-à-dire que le second élément de `l` devient le premier élément, et que le premier élément devient le dernier (la longueur de la liste n'est pas modifiée par cette fonction).
La figure 2 illustre le résultat de la fonction `cycle` sur la liste représentée sur la figure 1.

R.

```
def cycle(l):
  l.premier = l.premier.suivant
```

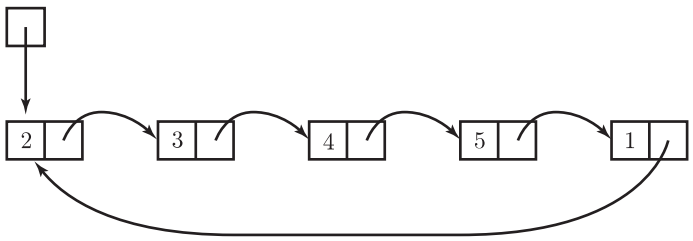

FIGURE 2 – Décalage de la liste illustrée sur la figure 1.

8. Quelle est la complexité de la fonction `cycle` ?

R. La complexité de la fonction `cycle` ne dépend pas de la longueur de la liste. Elle est donc en $O(1)$.

On se donne la fonction suivante :

```
def mystere(p, q):
  l = Liste(0)
  for i in range(1, p):
```

```
    ajouter_queue(i, l)
  for i in range(q):
    cycle(l)
  return l.premier.valeur
```

9. Simulez à la main le déroulement de la fonction pour les paramètres `p = 3` et `q = 5` (au brouillon). Représentez sur votre copie la liste l à la sortie de la première boucle `for` puis à la sortie de la seconde boucle `for`. Que renvoie la fonction dans cet exemple ?

R. Pour les valeurs $p = 3$ et $q = 5$, après la première boucle `for`, la liste contient les valeurs $0, 1, 2$. Après la seconde boucle `for` (on applique 5 fois la fonction `cycle`), la liste contient les valeurs $2, 0, 1$. La valeur renvoyée est 2.

10. De manière générale, quelle est la valeur de `mystere(p,q)` ? (il y a une expression mathématique simple du résultat).

R. À la sortie de la première boucle `for`, la liste l contient les entiers de 0 à $(p-1)$. Puis la seconde boucle fait cycler la liste q fois. Après la seconde boucle, le premier élément de la liste est donc q modulo p. C'est ce que renvoie la fonction `mystere`.

II. Listes doublement chaînées

La plupart des opérations sur les listes circulaires représentées par des listes chaînées nécessitent de trouver le dernier nœud de la liste pour maintenir la circularité. Cette opération est pénible dans le cas des listes simplement chaînées car elle nécessite de parcourir la totalité de la liste pour atteindre le dernier nœud.
Une solution consiste à utiliser des listes doublement chaînées : chaque nœud connaît alors son successeur et son prédécesseur dans la liste. Puisque la liste est circulaire, le prédécesseur du premier nœud est le dernier nœud de la liste.
On utilisera les classes suivantes pour représenter les listes circulaires à l'aide de listes doublement chaînées :

```
class Noeud:
  def __init__(self, x):
    self.valeur = x
    self.suivant = self
    self.precedent = self

class Liste:
  def __init__(self, x):
    n = Noeud(x)
    self.premier = n
```

11. Pourquoi n'est-il pas nécessaire d'avoir un champ `dernier` dans la définition de la classe `Liste` (dans le cas des listes doublement chaînées que l'on avait vues précédemment on avait un champ `dernier` qui permettait de parcourir la liste dans le sens inverse si l'on voulait) ?

R. Ici, on peut directement accéder au dernier élément de la liste à l'aide de la description `l.premier.precedent` (sans avoir à parcourir toute la liste).

12. En utilisant les listes doublement chaînées, réécrivez les fonctions `ajouter_tete(x,l)` et `ajouter_queue(x,l)`.
Remarque : Les fonctions sont plus simples car il n'est plus nécessaire de parcourir toute la liste pour trouver le dernier élément, mais il faut faire attention à bien refaire tous les liens (dans les deux sens) des nœuds qui ont été modifiés.

R.

```
def ajouter_tete(x, l):
  n = Noeud(x)
  n.suivant = l.premier
  n.precedent = l.premier.precedent
  n.suivant.precedent = n
  n.precedent.suivant = n
  l.premier = n
def ajouter_queue(x, l):
  n = Noeud(x)
  n.suivant = l.premier
  n.precedent = l.premier.precedent
  n.suivant.precedent = n
  n.precedent.suivant = n
```

13. Quelle est la complexité des deux fonctions de la question précédente en fonction du nombre de nœuds de la liste ?

R. La complexité de ces fonctions est maintenant en $O(1)$.

14. Que deviennent les fonctions `longueur(l)` et `cycle(l)` dans le cas des listes doublement chaînées ?

R. Il n'est pas nécessaire de modifier ces fonctions (elles marchent telles qu'elles ont été écrites). La fonction `longueur` ne peut pas être accélérée dans le cas des listes doublement chaînées car on doit tout de même parcourir la liste entière pour compter le nombre d'éléments.

III. Tableaux

Une troisième manière de représenter les listes circulaires est d'utiliser des tableaux. On représente alors une liste circulaire de manière très similaire à la représentation des listes « simples » mais avec quelques légères différences tout de même.
Une liste circulaire est représentée par un tableau de taille `MAX` (où `MAX` est un entier suffisamment grand fixé à l'avance). Par ailleurs, on mémorise également deux valeurs : la taille de la liste (`longueur`) et la position de la tête de liste (`tete`).
Une représentation d'une liste circulaire doit vérifier les propriétés suivantes :
– les n éléments de la liste occupent les n premières cases du tableau ;
– pour tout nœud de la liste, si sa valeur est dans la case i du tableau, la valeur de son successeur est dans la case $i+1$ si $i < n-1$ et dans la case 0 si $i = n-1$;
– la valeur de la tête de la liste se trouve dans la case `tete`.
La figure 3 illustre deux représentations possibles de la liste circulaire représentée par la figure 1.
On utilisera la classe suivante pour décrire une liste circulaire à l'aide de tableaux :

```
class Liste:
  def __init__(self, x):
    self.table = [None] * MAX
    self.table[0] = x
    self.tete = 0       # position de la tête de liste dans le tableau
    self.longueur = 1   # nombre d'éléments dans la liste
```

Remarque : Il peut paraître compliqué de ne pas imposer que la tête de liste se trouve dans la case 0 du tableau, mais ce choix est fait pour faciliter la fonction `cycle(l)` qui est caractéristique aux listes circulaires...

15. Écrivez la fonction `ajouter_tete(x,l)`
Indication : Il faut ajouter la nouvelle valeur à la place de la tête de liste. Avant cela, il faut décaler toutes les valeurs depuis la tête de la liste vers la droite...

R.

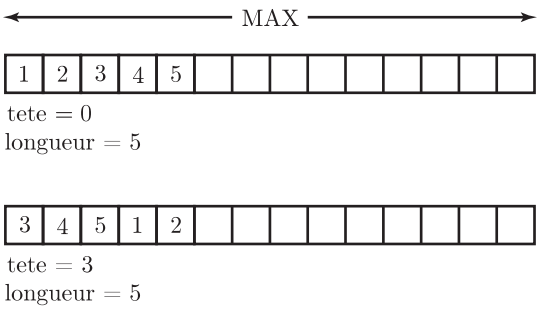

FIGURE 3 – Deux représentations possibles de la même liste circulaire.

```
def ajouter_tete(x, l):
  # il faut d'abord déplacer toutes les valeurs entre la fin du tableau
  # et la position de la tête de liste vers la droite
  for i in range(self.longueur-1, self.tete-1, -1):
    self.table[i+1] = self.table[i]
  self.table[self.tete] = x    # on ajoute x
  self.longueur += 1           # on modifie la longueur de la liste
```

16. Écrivez la fonction `cycle(l)`. Quelle est sa complexité ?

R.

```
def cycle(l):
  l.tete = (l.tete + 1) % l.longueur
```

La complexité de la fonction est constante ($O(1)$).

17. Écrivez la fonction `affiche(l)` qui affiche les éléments de la liste circulaire `l` en commençant par la tête de la liste.
Indication : Vous pourrez éventuellement utiliser l'opérateur `a % b` qui permet de renvoyer le reste de la division euclidienne de `a` par `b` (par exemple `25 % 11` vaut 3 car $25 = 11 \times 2 + 3$). En utilisant astucieusement cet opérateur il est possible de parcourir toute la liste en une seule boucle `for`...

R.

```
def affiche(l):
  for i in range(l.longueur):
    print l.table[(l.tete + i) % l.longueur]
```