Licence Informatique L2

Structure de données et algorithmes
Contrôle continu

Durée : 1 heure 30
Aucun document autorisé

Il sera tenu compte dans la correction de la qualité de la rédaction

1 Preuve d'algorithmes et complexité

1. On considère la fonction `factorielle` qui calcule la factorielle d'un entier n. Pour rappel :

$$n! = \prod_{i=1}^{i=n} i = 1 \times 2 \times 3 \times ... \times (n-1) \times n \tag{1}$$

   (a) Écrivez une fonction récursive en Python permettant d'effectuer la factorielle de l'entier n (pour $n \geq 0$).

   (b) Donnez la formule récursive de la complexité $T(n)$ de cet algorithme, en vous appuyant sur le nombre d'appels de l'algorithme à lui-même, et en évaluant le nombre total d'opérations réalisés dans le corps de l'algorithme. Vous supposerez que les tests, les opérations arithmétiques, les affectations et les opérations de sortie comptent pour une unité de complexité.

   (c) Démontrez par récurrence que cette complexité $T(n)$ vaut $5 \times n + 2$.

   (d) Déduisez l'ordre de grandeur asymptotique de l'algorithme `factorielle`.

2. On considère la série mathématique suivante qui permet de calculer le nombre d'Euler e (tel que $Ln(e) = 1$), qui s'appuie sur la factorielle n ! (avec 0 ! = 1) :

$$n! = e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + ... \tag{2}$$

   (a) Écrivez la fonction itérative `euler_it(n)` s'appuyant sur cette série pour calculer une valeur approchée de e. Cette fonction prendra un paramètre naturel n donnant le nombre d'itérations à effectuer. Vous considérerez que n est supérieur ou égal à 1 (ne pas le tester).

   (b) Prouvez cet algorithme itératif, en montrant qu'il termine et qu'à la fin de la ième itération on a bien calculé la somme :

$$e_i = \sum_{k=0}^{i} \frac{1}{k!} \tag{3}$$

   (c) Déterminez la complexité de la fonction itérative `euler_it` en fonction de n. Dans votre calcul vous pourrez reprendre le résultat de la complexité obtenu pour la fonction factorielle.

   (d) Déduisez l'ordre de grandeur asymptotique de l'algorithme itératif.

2 Listes chaînées

Dans cet exercice nous allons nous intéresser à une variante des listes dans lesquelles on marque certains éléments pour qu'ils soient ignorés quand on parcourt la liste. L'utilisation principale d'une telle liste, encore appelée `liste à trous`, est la gestion très simple de la suppression d'éléments (on se contente de marquer l'élément comme supprimé, mais il reste dans la structure de données).

Pour représenter des listes à trous on considère la classe `Noeud` en lui ajoutant un attribut `vide` qui contient un booléen indiquant si le noeud doit ou non être ignoré (par défaut un nouveau noeud ne doit pas être ignoré, la valeur de l'attribut `vide` est dans ce cas `False`).

```
class Noeud
    def __init__(self,x):
        self.valeur = x
        self.vide = False
        self.suivant = None


class Liste:
    def __init__():
        self.premier = None
        self.taille = 0
```

1. Écrivez la fonction `supprime(self,x)` qui prend en argument une liste `self` et un entier x et supprime de la liste toutes les occurrences de x.
   Indication : il faut parcourir la liste comme dans le cas usuel mais lorsqu'on trouve un noeud à supprimer, on se contente de changer la valeur de l'attribut `vide` en `True` puis on passe au noeud suivant. Il n'y a donc pas de liens à changer dans la chaîne.
2. Écrivez la fonction `longueur(self)` qui renvoie la longueur d'une liste à trous.
   Indication : cette fonction est très proche de la fonction écrite pour les listes chaînées usuelles, à ceci près qu'il ne faut pas compter les noeuds pour lesquels le champ `vide` vaut `True`.
3. De quoi dépend la complexité de la fonction `longueur` ? Quels sont les inconvénients des listes à trous ? Décrivez une situation où il serait très long de calculer la longueur d'une liste n'ayant pourtant que très peu d'éléments.

   Pour limiter le problème de la question précédente qui apparait lorsque l'on supprime beaucoup de noeuds dans une liste à trous, on va essayer de réutiliser au maximum les noeuds vides lorsque l'on insère un nouvel élément dans la liste.

   On considère maintenant des listes à trous triées, c'est-à-dire que les valeurs se trouvant dans les noeuds non-vides sont rangées de la plus petite (en tête de liste) à la plus grande (en queue de liste). **Les valeurs se trouvant sur les noeuds vides n'ont pas d'importance.**

   Lorsque l'on insère une nouvelle valeur dans la liste, on parcourt la liste comme dans le cas classique jusqu'à trouver une valeur supérieure à la valeur à insérer. Cependant, une fois cet emplacement trouvé, au lieu de créer un nouveau noeud, on commence par regarder si le noeud se trouvant juste avant le noeud où l'on doit insérer est un noeud $vide$. Si c'est le cas, on modifie la valeur de ce noeud et on le rend actif. La figure 1 illustre l'insertion d'un nouvel élément dans une liste à trous. Lorsqu'il n'y a pas d'élément vide à réutiliser, on procède comme dans les listes chaînées usuelles.
4. Écrivez la fonction `insere(self,x)` qui insère la valeur x dans la liste à trous triée `self`.
   Indication : il faut que votre fonction réutilise les noeuds vides disponibles, comme décrit précédemment.

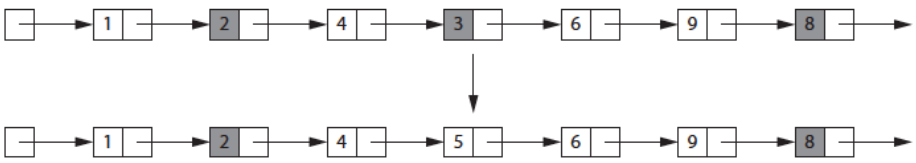

FIGURE 1 – Exemple d'insertion de la valeur 5 dans une liste à trous triée. **Les valeurs des éléments grisés n'ont pas d'importance (non triés)**. Les noeuds en gris sont les noeuds vides. On parcourt la liste jusqu'à trouver la première valeur supérieure à la valeur à insérer (ici c'est le 6), puis on regarde si le noeud précédent cette valeur est vide. Comme il l'est, on modifie sa valeur et on l'active.

FIGURE 2 – Résultat de la fonction *nettoie* sur la seconde liste de la figure 1.

5. Écrivez la fonction `nettoie(self)` qui supprime de la liste tous les noeuds vides.
   Indication : il faut parcourir la liste et à chaque fois qu'on rencontrer un noeud vide, on relie le noeud précédent au prochain noeud non vide. Attention, il est possible que le premier noeud de la liste soit vide. Il faut alors modifier l'attribut *premier* de $self$ pour qu'il pointe vers le premier noeud non vide. La figure 2 illustre le résultat de la fonction *nettoie* sur la liste de la figure 1.