Université de Lille — Faculté des Sciences et Technologies — Département Informatique

L1 MASS/SESI/PEIP - 2018–2019

Algorithmes et Programmation 1

DS2 - 27/04/2019

durée 2h - documents non autorisés (sauf memento Python3)

Ce devoir contient trois exercices indépendants. Indiquez sur la copie votre parcours et le numéro de votre groupe. Sauf mention explicite du contraire, il n'est pas attendu de documentation complète pour les fonctions demandées mais uniquement les conditions d'utilisation (CU). En revanche, toute autre fonction doit l'être.

Exercice 1 *Document composite*

Le but de cet exercice est de produire un fichier texte constitué de lignes puisées dans d'autres fichiers texte.

On appelle *patron de document* un fichier texte dont les lignes sont toutes composées d'un nom de fichier suivi du caractère `':'`, suivi d'un nombre, suivi d'un second caractère `':'`, suivi d'un second nombre au moins égal au premier. Voici un exemple de patron :

```
cigale.txt:1:5
toto.txt:10:20
bidule.py:0:10
```

On suppose que les noms de fichier ne peuvent pas contenir le caractère `':'`.

Le patron donné en exemple ci-dessus décrit un texte constitué des lignes 1 à 4 du texte contenu dans le fichier `cigale.txt`, suivies des lignes 10 à 19 du texte contenu dans le fichier `toto.txt`, suivies des lignes 0 à 9 du texte contenu dans le fichier `bidule.py`.

Question 1 Réalisez une procédure paramétrée par le nom d'un fichier patron et le nom d'un fichier résultat qui permet à partir du patron de construire le fichier obtenu en mettant bout à bout les morceaux de fichiers dont les noms figurent dans le patron.

Vous pourrez faire l'hypothèse que
- les lignes contenues dans le patron sont bien conformes à la description donnée ci-dessus ;
- les fichiers mentionnés dans le patron existent bien ;
- et ces fichiers contiennent un nombre suffisant de lignes.

Exercice 2 *Correction automatique d'un questionnaire*

Dans cet exercice, les bonnes réponses à un questionnaire à choix unique sont mémorisées dans un dictionnaire qui associe le nom de chaque question (la clé) au numéro de l'unique bonne réponse (la valeur). Les propositions de réponses sont numérotées de 1 à 4. Par exemple :

```
bonnes_reponses = {'Ex1Q1' : 1, 'Ex1Q2' : 3, 'Ex1Q3' : 1, 'Exo2Q1' : 2}
```

D'autre part, les réponses données par un étudiant sont mémorisées dans un autre dictionnaire comportant les même clés, mais dont les valeurs sont les réponses cochées par l'étudiant. L'absence de réponse à une question est représentée par la valeur 0. Par exemple :

```
reponses_timoleon = {'Ex1Q1' : 1, 'Ex1Q2' : 1, 'Ex1Q3' : 1, 'Exo2Q1' : 0}
```

Question 1 Réalisez la fonction `note_questionnaire` qui prend en paramètre le dictionnaire des bonnes réponses et un dictionnaire des réponses d'un étudiant et qui renvoie la note de cet étudiant. Nous considérons que une bonne réponse vaut 1 point, une mauvaise vaut -1 point et une absence de réponse vaut 0. Par exemple, avec ses réponses, Timoleon a une note de 1.

Pour faire des statistiques sur les résultats d'une promotion d'étudiants, une liste de dictionnaires de réponses à un même questionnaire est construite. Chaque dictionnaire y représente les réponses d'un étudiant.

Question 2 Réalisez la fonction `statistiques_questionnaire` qui prend en paramètre un dictionnaire de bonnes réponses, une liste de réponses d'étudiants pour le même questionnaire et qui renvoie un dictionnaire associant à chaque question (la clé) un triplet indiquant, dans l'ordre, le nombre de bonnes réponses, le nombre de

mauvaises réponses et le nombre d'étudiants qui n'ont pas répondu, sur l'ensemble de la promotion. Par exemple, une statistique pour une promotion de 100 étudiants pourrait être :

```
{'Ex1Q1' : (40, 37, 23), 'Ex1Q2' : (52, 40, 8), 'Ex1Q3' : (39, 29, 32), 'Exo2Q1' : (60, 35, 5)}
```

Exercice 3 *Le deuxième plus petit*

Les listes considérées dans cet exercice sont des listes de nombres entiers tous différents dont la longueur, notée n, est au moins égale à deux. L'objectif de cet exercice est de comparer quelques méthodes pour déterminer le deuxième plus petit élément de telles listes. Par exemple, pour la liste [3, 0, 5, 2, 9, 4, 6, 8], désignée par `liste_exemple` dans la suite, le deuxième plus petit élément est 2.

Méthode 1 Une première idée pour résoudre ce problème consiste à trier la liste et considérer le deuxième élément de la liste triée.

Question 1 En utilisant la méthode `sort` des listes, programmez une première version nommée `deuxieme_plus_petit1` de la fonction souhaitée. Vous prendrez garde à ne provoquer aucun effet de bord sur la liste comme le montre l'exemple qui suit.

```
>>> deuxieme_plus_petit1(liste_exemple)
2
>>> liste_exemple
[3, 0, 5, 2, 9, 4, 6, 8]
```

Question 2 En supposant que la méthode `sort` a été programmée avec l'un ou l'autre des deux algorithmes de tri vus en cours, exprimez en fonction de la longueur n de la liste le nombre de comparaisons d'éléments de cette liste effectuées pour calculer le deuxième plus petit élément. Veuillez préciser l'algorithme dont vous indiquez le nombre de comparaisons d'éléments.

Méthode 2 Dans le tri par sélection, on est amené à chercher l'indice du plus petit élément d'une (tranche de) liste.

Question 3 Réalisez la fonction `indice_plus_petit` qui renvoie l'indice du plus petit élément de la liste passée en paramètre.

```
>>> indice_plus_petit(liste_exemple)
1
```

Question 4 Exprimez en fonction de n le nombre de comparaisons d'éléments de liste effectuées par cette fonction.

On peut trouver le deuxième plus petit élément si on connaît l'indice du plus petit élément. Il suffit de parcourir la liste et de retenir le plus petit élément rencontré dont l'indice est différent de celui du plus petit.

Question 5 Programmez une deuxième version `deuxieme_plus_petit2`.

Question 6 Exprimez en fonction de n le nombre de comparaisons effectuées par cette version.

Méthode 3 Dans la deuxième méthode la liste est parcourue deux fois : une fois pour détérminer le plus petit élément et une seconde fois pour trouver le deuxième plus petit. La troisième méthode va faire le travail en un seul parcours. Pour cela on garde en mémoire les deux plus petits éléments rencontrés, et on les met à jour au fur et à mesure du parcours de la liste.

Question 7 Programmez la troisième version `deuxieme_plus_petit3`.

Question 8 Exprimez le nombre de comparaisons effectuées dans le meilleur et le pire des cas que vous décrirez soigneusement.

Méthode 4 Cette dernière méthode est un peu plus complexe à programmer. Elle repose sur le principe d'un tournoi. On compare chaque paire consécutive d'éléments de la liste et on ne retient que les plus petits de ces paires. Il nous reste une liste avec deux fois moins d'éléments. On itère le meme processus avec cette nouvelle liste jusqu'à obtenir un seul élément qui est alors le plus petit de tous. La figure 1 illustre un tel tournoi avec les éléments de la liste [3, 0, 5, 2, 9, 4, 6, 8]. Le niveau le plus bas représente le premier tour dans lequel tous les éléments de la liste participent. Puis vient le deuxième tour auquel seuls les gagnants du premier tour participent. Etc...

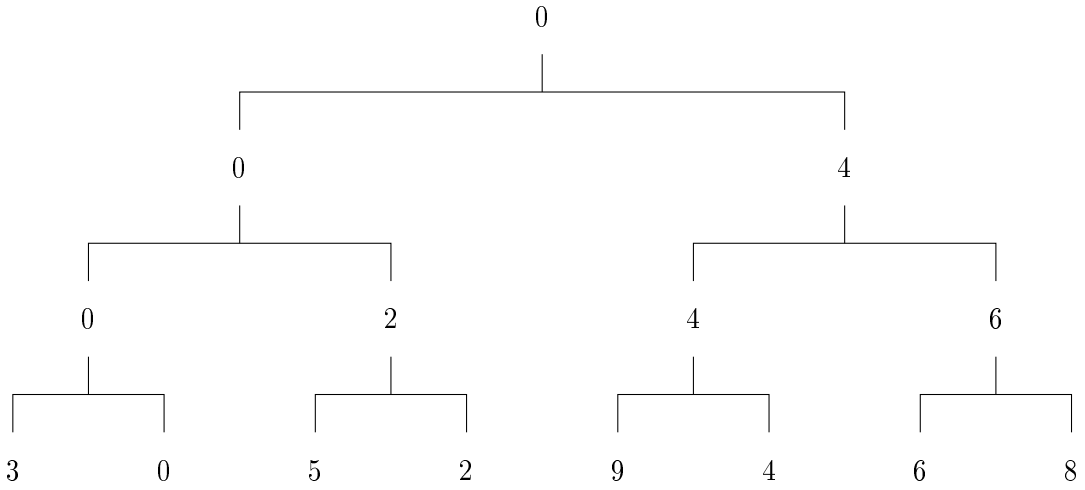

FIGURE 1 – Un tournoi avec les éléments de la liste [3, 0, 5, 2, 9, 4, 6, 8]

Lorsqu'on a déterminé le plus petit de tous (dans notre exemple c'est 0) le deuxième plus petit est l'un des éléments que le plus petit a rencontré (et battu) au cours du tournoi. Il suffit de le chercher dans la liste des éléments que le plus petit a rencontrés (dans notre exemple c'est [3, 2, 4]).

Question 9 Dans l'exemple illustré par la figure 1, combien de comparaisons d'éléments de liste sont -elles effectuées pour trouver le deuxième plus petit ?

Question 10 Plus généralement, pour une liste de longueur $n = 2^p$, $p \in \mathbb{N}$ non nul, avec cet algorithme, combien de comparaisons sont-elles effectuées ? Vous exprimerez ce nombre en fonction de n.
