DEVOIR D'INFORMATIQUE (13/01/2017)

La compression bzip

Dans ce sujet, il est question de l'algorithme de Burrows-Wheeler qui compresse très efficacement des données textuelles.

Tout au long du sujet, il est possible d'utiliser les fonctions ou procédures demandées dans les questions précédentes du sujet, même si ces questions n'ont pas été traitées.

Nous attacherons la plus grande importance à la lisibilité du code produit par les candidats ; aussi, nous encourageons les candidats à introduire des procédures ou fonctions intermédiaires lorsque cela en simplifie l 'écriture.

Complexité. La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P a une complexité en $O(f(n))$ s'il existe $K > 0$ tel que la complexité de P est inférieure à $K\,f(n)$ pour tout n.

De manière générale, on s'attachera à garantir une complexité aussi petite que possible dans les fonctions écrites. Le candidat y sera tout particulièrement sensible lorsque la complexité d'une fonction est demandée. Dans ce cas, il devra de plus justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Listes en Python. On rappelle qu'en Python, les listes sont des tableaux dynamiques à une dimension. Sur les listes, on dispose des opérations suivantes, qui sont de complexité constante :

- [ ] crée une liste vide c'est-à-dire ne contenant aucun élément.
- si n est un entier, **[x]*n** crée une liste de longueur n où chaque élément est égal à la valeur de **x**.
- **len**(**liste**) renvoie la longueur de la liste **liste**.
- **liste**[i] renvoie l'élément d'indice i de la liste **liste** s'il existe, ou produit une erreur sinon (noter que les éléments sont indicés à partir de 0).
- **liste**[$-i$] renvoie l'élément d'indice **len**(**liste**)-i de la liste **liste** s'il existe ou produit une erreur sinon. En particulier, **liste**[-1] renvoie le dernier élément de la liste.
- **liste**[$i : j$] renvoie la liste extraite de **liste** formée des éléments d'indices i *inclus* à l'indice j *non inclus*. En particulier, **liste**[i : **len**(**liste**)], que l'on peut abréger en **liste**[i :], représente tous les éléments de la liste à partir de celui d'indice i, et **liste**[$0 : i$], que l'on peut abréger en **liste**[: i], représente les i premiers éléments de la liste.
- **liste.append**(**x**) ajoute la valeur de **x** à la fin de la liste **liste** qui s'allonge ainsi d'un élément.
- **liste.extend**(**liste2**) ajoute tous les éléments de la liste **liste2** à la fin de la liste **liste**.

Enfin, l'instruction **x in liste** renvoie le booléen **True** ou **False** selon que **x** appartient à la liste **liste** ou pas. Cette opération est de complexité $O(n)$ où n est la longueur de la liste.

Important : L'usage de toute autre fonction sur les listes telle que **liste.insert(i,x)**, **liste.remove(x)**, **liste.index(x)**, ou encore **liste.sort(x)** est rigoureusement interdit (ces fonctions devront être programmées explicitement si nécessaire).

Préliminaire

Chaque caractère alphabétique peut être repéré par un numéro compris entre 0 et 255 (son code ASCII). Par exemple, les lettres majuscules de 'A' à 'Z' correspondent aux codes de 65 à 90 et les lettres minuscules de 'a' à 'z' aux codes de 97 à 122. La fonction Python **chr** permet de passer de l'entier au caractère correspondant, la fonction **ord** réalise la conversion inverse.
Exemple : **chr**(100) renvoie le caractère 'd', **ord**('A') renvoie la valeur 65.

Question 1.

a) Écrire une fonction **transforme**(**texte**) qui prend en argument un texte contenu dans la chaîne de caractères **texte** et qui renvoie une liste **t** de même longueur que **texte** telle que, pour tout indice i, $\mathbf{t}[i]$ est le code du caractère $\mathbf{texte}[i]$.

Exemple : si **texte** = 'ABRACADABRA', on aura, puisque **ord**('R')=82,

$$\mathbf{t}=[65, 66, 82, 65, 67, 65, 68, 65, 66, 82, 65].$$

b) Évaluer la complexité de cette fonction (en fonction de la longueur n de **t**).

Dans toute la suite du problème, nous supposerons que cette fonction a préalablement été appliquée au texte à compresser. Ainsi, ce texte sera donné dans une liste **t** de taille n ; chaque case de cette liste, d'indice i compris entre 0 et $n-1$, contient donc un entier entre 0 et 255. On confondra donc chaque lettre avec l'entier qui la représente.

I. Compression par redondance

La compression par redondance compresse un texte d'entrée qui possède des répétitions consécutives de lettres (ou d'entiers dans notre cas). Dans un premier temps, on calcule les fréquences d'apparition de chaque caractère dans le texte d'entrée. Puis on compresse le texte.

Question 2.

a) Écrire la fonction **occurrences**(**t**) qui prend en argument la liste **t** et qui retourne une liste **occ** de longueur 256 telle que $\mathbf{occ}[i]$ est le nombre d'occurrences de l'entier i dans **t**.

Exemple : si $\mathbf{t} = [65, 66, 82, 65, 67, 65, 68, 65, 66, 82, 65]$, on aura :

$$\mathbf{occ}[65]=5,\ \mathbf{occ}[66]=2,\ \mathbf{occ}[67]=1,\ \mathbf{occ}[68]=1,\ \mathbf{occ}[82]=2$$
$$\forall\, i \in [\![0\,;255]\!],\ i \notin \{65, 66, 67, 68, 82\} \implies \mathbf{occ}[i]=0.$$

b) Évaluer la complexité de cette fonction (en fonction de la longueur n de **t**).

Question 3.

Écrire la fonction **mini**(**t**) qui prend en argument le texte **t** et qui retourne le plus petit entier i de l'intervalle $[\![0\,;255]\!]$ qui apparaît le moins souvent dans le texte **t** (le nombre d'occurrences de cet entier peut être nul). Cette fonction fera évidemment appel à la fonction de la question précédente.

Exemple : si $\mathbf{t} = [0, 0, 0, 3, 3, 2, 3, 3, 3, 3, 3, 5, 5, 5, 5, 4]$, on aura **mini**(**t**)=1 car 1 est le plus petit entier qui n'apparaît pas dans cette liste.

L'entier **mini**(**t**) servira de *marqueur*. On note μ ce marqueur, et pour simplifier, on suppose que ce marqueur est nul. Ainsi μ est le plus petit entier i tel que **occ**$[i]$=0 quand **occ**=**occurrences**(**t**).

La compression par redondance du texte **t** fonctionne comme suit : on rajoute μ au début du texte, puis toute répétition maximale contigüe dans le texte d'une lettre où $\mathbf{t}[i]=\mathbf{t}[i+1]=\cdots=\mathbf{t}[j]=k$ est codée par les trois entiers μ, $(j-i)$, k lorsque $j-i\geqslant 3$; si une lettre apparaît moins de 3 fois consécutives, chaque occurrence de cette lettre est codée par cette même lettre.

Par exemple, si le tableau **t** contient les valeurs $[0,0,0,3,3,2,3,3,3,3,3,3,5,5,5,5,4]$, le marqueur est 1 et le texte **t0** compressé est alors

$$\underbrace{1}_{\mu},\underbrace{0,0,0}_{0,0,0},\underbrace{3,3}_{3,3},\underbrace{2}_{2},\underbrace{1,5,3}_{3,3,3,3,3,3},\underbrace{1,3,5}_{5,5,5,5},\underbrace{4}_{4}.$$

Question 4.

Écrire la fonction **compresse**(**t**) qui prend comme paramètre la liste **t** et retourne une liste d'entiers **t0** représentant le texte compressé.

Question 5.

Pour pouvoir décoder un texte **t0** ainsi compressé, il suffit de connaître le marqueur utilisé. Or ce marqueur est le premier entier du texte compressé.

Écrire la fonction **decompresse**(**t0**) qui prend comme paramètre le texte compressé **t0** et retourne le texte **t**.

II. Transformation de Burrows-Wheeler

Le codage par redondance n'est efficace que si le texte présente de nombreuses répétitions consécutives de lettres. Ce n'est évidemment pas le cas pour un texte pris au hasard. La transformation de Burrows-Wheeler est une transformation qui, à partir d'un texte donné, produit un autre texte contenant exactement les mêmes lettres mais dans un autre ordre où les répétitions de lettres ont tendance à être contigües. Cette transformation est bijective.

Considérons par exemple le texte d'entrée `concours`. Pour simplifier la présentation, nous utilisons ici des caractères pour la liste d'entrée. Cependant, dans les programmes, on considèrera toujours (comme dans la première partie) que le texte d'entrée est une liste d'entiers compris entre 0 et 255 inclus. Le principe de la transformation suit les trois étapes suivantes :

1 - On regarde toutes les rotations du texte. Dans notre cas, il y en a 8 qui sont :

concours
oncoursc
ncoursco
courscon
oursconc
ursconco
rsconcou
sconcour

2 - On trie ces rotations par ordre lexicographique (l'ordre du dictionnaire) :

concour**s**
courscon**c**
ncoursc**o**
oncours**c**
oursconc
rsconco**u**
sconcou**r**
ursconc**o**

3 - Le texte résultant est formé par toutes les dernières lettres des mots dans l'ordre précédent, soit `snoccuro` dans l'exemple, ainsi que de l'indice de la lettre dans ce texte résultant qui est la première lettre du texte original, soit 3 dans notre exemple. On appelle cet entier la *clé* de la transformation.

On remarque que les deux `c` du texte de départ se retrouvent côte à côte après la transformation. En effet, comme le tri des rotations regroupe les mêmes lettres sur la première colonne, cela conduit à rapprocher aussi les lettres de la dernière colonne qui les précèdent dans le texte d'entrée.

On le constate aussi sur la chaîne :

'concours de l ecole polytechnique et des ecoles normales superieures'

dont la transformée par Burrows-Wheeler est :

'stlseesesmeeesn udlt pdllr icrn ooaoroh ccpcncu iueoureeee eyqseol' ,

etc...

En pratique, on ne va pas calculer et stocker l'ensemble des rotations du mot d'entrée. On se contente de noter par $rot[i]$ la i-ème rotation du mot. Ainsi, dans l'exemple, $rot[0]$ représente le texte d'entrée `concours`, $rot[1]$ représente `oncoursc`, $rot[2]$ représente `ncoursco`, etc...
On notera que les textes représentés par $rot[i]$ peuvent être facilement obtenues en `Python` à l'aide des fonctions d'extraction de liste rappelées au début du sujet.

Attention : *rot* représente ici une notation, vous ne devez pas créer ni utiliser une variable `Python` correspondante !

Question 6.

Écrire la fonction **comparerRotations**(**t**, i, j) qui prend comme arguments le texte **t** de longueur n et deux indices i, j et qui renvoie, en temps linéaire par rapport à n :

- 1 si $rot[i]$ est (strictement) plus grand que $rot[j]$ dans l'ordre lexicographique,
- -1 si $rot[i]$ est (strictement) plus petit que $rot[j]$ dans l'ordre lexicographique,
- 0 sinon.

On exige une complexité en $O(n)$, où n est la taille de **t**.

On suppose disposer d'une fonction **triRotations**(**t**) qui trie les rotations du texte donné dans le tableau **t** en utilisant la fonction**comparerRotation**. Elle retourne une liste d'entiers **r** représentant les numéros des rotations $(rot[r[0]] \leqslant rot[r[1]] \leqslant \cdots \leqslant rot[r[n-1]])$. Cette fonction réalise dans le pire des cas $O(n \ln n)$ appels à la fonction de comparaison.

Question 7.

a) Écrire une fonction **codageBW**(**t**) qui prend en paramètre la liste **t** et qui renvoie une liste **t0** contenant le texte après transformation, ainsi que la clé.

b) Donner un ordre de grandeur du temps d'exécution de la fonction **codageBW** en fonction de n.

Pour réaliser l'ensemble du codage, il ne reste plus qu'à réaliser la compression par redondance sur la transformée **t0** du texte d'entrée **t**.

III. Transformation de Burrows-Wheeler inverse

Pour décoder le texte **t0** (`snoccuro` dans l'exemple) avec sa clé (ici 3) obtenu après transformation, on construit d'abord une liste **triCars** de même taille qui contient les mêmes lettres que le texte **t0** mais dans l'ordre lexicographique croissant. Dans l'exemple, **triCars** = $[c, c, n, o, o, r, s, u]$.

Question 8.

Écrire une fonction **frequences**(**t0**) qui prend comme argument une liste **t0** correspondant au texte codé (avec la clé dans la dernière case), et qui renvoie un tableau de taille 256 contenant le nombre d'occurrences de chaque lettre dans **t0**.

Question 9.

Écrire la fonction **triCarsDe**(**t0**) qui part du texte codé **t0** et qui renvoie, en temps linéaire par rapport à n, la liste **triCars** décrit précédemment. Cette fonction utilisera la fonction **frequences** précédente.

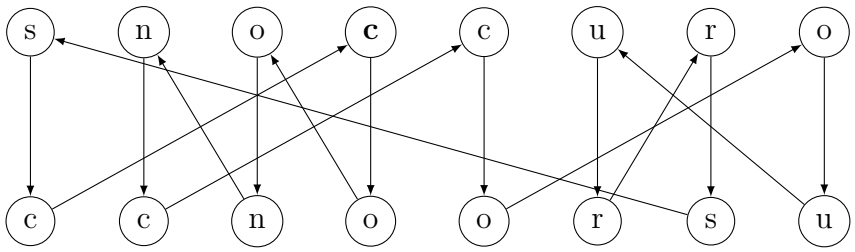
On exige une complexité en $O(n)$, où n est la taille de **t0**.

Puis on considère le texte codé **t0** et le tableau **triCars** précédent (la lettre de position égale à la clé est représentée en gras) :

s	n	o	**c**	c	u	r	o
c	c	n	o	o	r	s	u

À chaque lettre de la première ligne, on associe la lettre de la seconde à la même position.

À chaque lettre de la deuxième ligne, on associe la même lettre de même rang dans la première ligne. La figure suivante montre ces deux correspondances :



On retrouve le texte de départ `concours` en partant de la clé (position de la lettre en caractère gras) et en suivant les flèches du dessin précédent.

Il faut donc construire la liste **indices** tel que **indices**$[i]$ est l'indice de la lettre **triCars**$[i]$ dans le texte **t0**. Si plusieurs occurrences de cette lettre figurent dans **t0**, on fait correspondre celle qui figure au même rang dans **t0**. La liste **indices** donne donc la correspondance représentée par les flèches de la seconde ligne vers la première. Sur l'exemple, la liste **indices** contient les valeurs $[3, 4, 1, 2, 7, 6, 0, 5]$.

Question 10.

Écrire la fonction **trouverIndices**(**t0**) prenant en paramètre le texte **t0** codé et qui retourne la liste **indices** précédemment décrite.

Quel est son temps d'exécution en fonction de la longueur de **t0** ?

Question 11.

Écrire une fonction **decodageBW**(**t0**) qui prend comme paramètre un texte **t0** et retourne le texte **t** d'origine.

Quel est son temps d'exécution en fonction de la longueur de **t0** ?