DEVOIR LIBRE 01

INFORMATIQUE

CLASSE DE TSI2

A rendre le 15 Novembre 2019

La rigueur du raisonnement et la clarté de la présentation seront prises en compte dans la notation.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie. Toutes les fonctions étudiées doivent donc être écrites dans ce langage. Toute version simplement algorithmique des fonctions étudiées (sauf dans le cas précisé dans l'énoncé où l'on demande de développer un exemple) ne sera pas comptabilisée dans le barème.
Enfin, un certain nombre de commandes Python pouvant être utiles sont rappelées en fin d'énoncé.

Compression bzip

Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction,multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute en temps $O(n^\alpha)$ s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn^\alpha$.

Dans ce sujet, il sera question de l'algorithme de Burrows-Wheeler qui compresse très efficacement des données textuelles. Le texte d'entrée à compresser sera représenté par un tableau t contenant des entiers compris entre 0 et 255 inclus.

Partie 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 entier dans le texte d'entrée. Puis on compresse le texte.

1. Écrire la fonction *occurrences*(t) qui prend en argument un tableau d'entrée et qui retourne un tableau r de taille 256 tel que $r[i]$ est le nombre d'occurences de i.
   Indication : on commencera par créer un tableau r de taille 256 rempli de 0.
2. Écrire la fonction *mini*(t) qui prend en argument le tableau t et qui retourne le plus petit entier de l'intervalle $[0, 255]$ qui apparaît le moins souvent dans le tableau t. (Le nombre d'occurences de cet entier peut être nul.)
   Indication : on commencera par calculer le tableau r grâce à la fonction précédente. On crée ensuite une variable c_{min} qui contient le caractère ayant le moins d'occurences dans le texte, parmi tous les caractères déjà examinés. Initialement, que vaut c_{min} ? On fera une boucle.
   Donner un exemple « à la main » de calcul de *mini*(t) à partir d'un tableau t que vous choisissez.

T.S.V.P →

L'entier $mini(t)$ servira de marqueur. La compression par redondance du texte t fonctionne comme suit : toute répétition contigue (donc répétée au moins une fois) d'un entier où $t[i] = t[i+1] = ... = t[j] = k$ est codée par les trois entiers

$$mini(t), j - i, k.$$

Toute apparition unique d'un entier k est codée par ce même entier.
On crée ainsi un tableau t' compressé ayant pour première valeur $mini(t)$ que l'on note *marqueur* puis les codes calculés selon le processus précédent.
Par exemple, si le tableau t est :

$$t = (0, 0, 3, 2, 3, 3, 3, 3, 3, 3, 5)$$

alors le marqueur $mini(t)$ est 1 car 1 n'apparaît pas dans ce tableau (et c'est le plus petit entier ainsi). Le texte t' compressé commence par *marqueur* suivi de $1, 1, 0$ à cause de $0, 0$ de t, puis 3 seul, puis 2 seul, puis $1, 5, 3$ car il y a six 3 qui se suivent et enfin 5 seul.
Finalement :

$$t' = (marqueur, 1, 1, 0, 3, 2, 1, 5, 3, 5)$$

3. Compresser de la même façon le texte $t = (0, 2, 2, 2, 1, 4, 2, 2, 4, 3, 6, 6, 1, 1, 1, 1)$.
4. Pour construire le nouveau tableau t', il faut donc calculer les valeurs $j - i$ dans le cas de répétitions de l'élément $t[i]$. Plus précisément, on part d'une position d'indice i, on cherche la position du plus grand indice $j \geq i$ tel que : $t[i] = ... = t[j]$.
Écrire une fonction $plage(t, i)$ qui a pour variable locale j initialisé à $i + 1$.
Indication : dans cette fonction, on utilisera cette boucle : tant que $j < n$ et que $t[j] = t[i]$, on incrémente j. Quelle valeur doit-on retourner après la boucle ?
5. Écrire la fonction $codage(t)$ qui prend pour paramètre le tableau t et retourne un tableau d'entiers $t1$ représentant le texte compressé (pour Python, il vaut mieux appeler le nouveau $t1$ que t').
Indication : on appelera n la longueur de t. Puis marqueur la valeur $mini(t)$. On crée la liste $t1$ initialement réduite à marqueur. On parcourt alors le tableau t avec l'indice i. On calcule pour chaque valeur de i la valeur $j = plage(t, i)$. Si $j = i$, le caractère $t[i]$ est codé par lui-même dans le tableau $t1$. Sinon, la plage $t[i...j]$ est codé par trois caractères, comme expliqué plus haut. Le traitement de cette plage étant terminé, on reprend le parcours pour trouver la plage suivante : i prend la valeur $j + 1$ (juste après la plage qui vient d'être étudiée).

On désire maintenant décoder le tableau, c'est-à-dire partir d'un texte t' version compressée de t et retrouver t.

6. Décompresser « à la main » le tableau :

$$t' = (2, 2, 3, 1, 4, 7, 2, 1, 3, 0, 2, 4, 6, 8).$$

7. Écrire maintenant une fonction $decodage(t1)$ qui part donc d'un tableau compressé $t1$ et qui retourne t, version décompréssée.
Indication : on commence par identifier le marqueur : c'est le premier élément du tableau $t1$. Puis on parcourt ce tableau, via un indice k, initialisé à 1. On crée également un tableau vide t. Lorsque l'on rencontre en case d'indice k du tableau $t1$ le marqueur alors il faut ajouter au tableau t une plage formée par $j - i + 1$ éléments (à écrire avec l'indexation des éléments de $t1$) égaux à un certain élément de $t1$ que vous devez déterminer en fonction de k. Lorsque l'on ne rencontre pas le marqueur, on ajoute cet élément au tableau t.

Partie 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 contigues. Cette transformation est bijective.

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

α) 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

β) On trie ces rotations par ordre lexicographique (l'ordre du dictionnaire)

concour**s**
coursco**n**
ncoursc**o**
oncours**c**
ourscon**c**
rsconco**u**
sconcou**r**
ursconc**o**

γ) Le texte résultat 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 résultat 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 derniére colonne qui les précèdent dans le texte d'entrée.

On le constate aussi sur la chaîne *concours* ⊔ *de* ⊔ *l* ⊔ *ecole* ⊔ *polytechnique* dont la transformée de Burrows-Wheeler est : *sleeeeen* ⊔ *dlt* ⊔ *ucn* ⊔ *ooohcpcc* ⊔ *iuryqo*.

Le symbole ⊔ marque la séparation entre les mots et est considéré comme une lettre.

1. Appliquer le processus précédent à la main à : *mathematiques* et donner sa transformée de Burrows-Wheeler. (On fera deux tableaux-colonnes.) Quelle est la valeur de la clé ?

   Revenons au cas général, 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 le premier exemple, $rot[0]$ représente le texte d'entrée *concours*, $rot[1]$ représente *oncoursc*, $rot[2]$ représente *ncoursco*, etc.

2. Écrire la fonction $comparerRotations(t, i, j)$ qui prend pour arguments le texte t et deux indicesi, j, et qui renvoie, en temps linéaire par rapport à $n = len(t)$ une variable notée res qui vaut :
   1 si $rot[i]$ est plus grand que $rot[j]$ dans l'ordre lexicographique.
   -1 si $rot[i]$ est plus petit que $rot[j]$ dans l'ordre lexicographique.
   0 sinon.
   Indication : les éléments du tableau $rot[i]$ (resp. $rot[j]$) sont $t\,[(i+k)\,mod\,n]$ (resp. $t\,[(j+k)\,mod\,n]$. On part de l'hypothèse que ces deux tableaux sont égaux ($res = 0$), jusqu'à ce qu'on ait trouvé un indice prouvant le contraire. Il faut alors s'arrêter de tester (on sort de la boucle lorque res n'est plus égal à 0). Il ne faut pas non plus tester indéfiniment les mêmes éléments dans le cas où $rot[i] = rot[j]$
   C'est pourquoi la boucle principale commencera par : $while\,k < n\,and\,res == 0$:
   On utilisera aussi les commandes if, $elif$ et $else$ dans cette boucle.
3. Faire fonctionner à la main $comparerRotations$ avec $t = [mathematiques]$, $i = 4$ et $j = 11$. Que vaut alors res ?
4. On suppose maintenant disposer d'une fonction $triRotations(t)$ qui trie les rotations du texte donné dans le tableau t en utilisant la fonction $comparerRotation$. Elle doit retourner un tableau d'entiers r représentant les numéros des rotations

   $$rot[r[0]] \leq rot[r[1]] \leq ... \leq rot[r[n-1]]$$

   Cette fonction réalise dans le pire des cas $O(n \ln n)$ appels à la fonction de comparaison (en effet, on fait un tri-fusion pour construire $triRotations$ mais cela c'est une autre histoire !)
   Écrire la liste r correspondante à $t = [mathematiques]$.
5. Écrire une fonction $codageBW(t)$ qui prend en paramètre le tableau t et qui renvoie un tableau tt contenant le texte après transformation. De plus, la clé sera stockée dans la dernière case de ce tableau. Ainsi $codageBW([concours)$ renvoie $[snoccuro3]$.
   Indication : On commence par rentrer $triRotations(t)$ dans r. On remarque ensuite que la lettre d'indice k du mot tt (sauf $tt[n]$ qui est la clé) est la dernière lettre (i.e celle d'indice $n-1$) du mot $rot[r[k]]$ (i.e la lettre d'indice $n+r[k]-1\,mod\,n$ du mot t. Cet indice est aussi égal à $r[k]-1$ sauf si $r[k] = 0$. Mais comme la case d'indice -1 désigne la case d'indice $n-1$, cette formule convient pour tout k. Au départ tt est la liste vide puis on la remplit dans une boucle $for\,k\,in\,range(n)$. Quant à la clé, c'est la position k où est stockée (dans le tableau tt) la première lettre du mot t, i.e celle d'indice 0. C'est donc l'unique entier k tel que $r[k] - 1 = 0$.
6. Donner un ordre de grandeur du temps d'éxécution (c'est-à-dire la complexité) de la fonction $codageBW$ en fonction de n.

Quelques commandes Python utiles

1) $n = len(l)$ traduit que la liste l a n éléments.
Le premier élément de l est $l[0]$ et le dernier est $l[n-1]$.
2) $l = [\,]$ désigne la liste vide.
3) $r = [0] * 25$ cree une liste r remplie de 0 et de taille 25.
4) $tt.append(t[k])$ rajoute $t[k]$ en fin de la liste tt.
5) $return(t)$ renvoie t.
6) $j == i$ signifie que j et i sont égaux.
7) $j = i$ signifie que j devient i.
8) $k+ = 3$ signifie que l'on rajoute 3 à k.
8) $t = t + [t1[i]] * (t1[j] + 1)$ signifie qu'à la liste t, on rajoute une plage formée par $t1[j] + 1$ éléments égaux à $t1[i]$.