Devoir en temps limité n° 2 — 2h

Ce devoir est à composer sur machine. Seul le code PYTHON est à compléter et il sera récupéré automatiquement. Il n'y a donc rien à rendre ni à rédiger sur feuille.

Vous devez compléter le fichier `ds2.py` qui se trouve à la racine de votre répertoire, *sans en modifier le nom ni l'emplacement*. Le code devra impérativement compiler : *aucune* erreur ne doit apparaître lorsque l'on exécute le script. Il est impératif de veiller au strict respect de ces consignes.

Vous pouvez vérifier que vos fonctions sont correctement prises en compte et passent les premiers tests (il y en aura d'autres) en exécutant le fichier `verif_ds2.py` qui se trouve aussi à la racine de votre répertoire. Attention, sous PYZO, il faut bien utiliser le raccourci F5, c'est-à-dire « *Démarrer le script* ».

✔

On prendra bien soin de tester toutes les fonctions sur des cas triviaux, puis un peu plus complexes. La vérification ne garantit aucunement que vos fonctions sont correctes.

On rappelle que l'on dispose des commandes suivantes, bien utiles :

- `help` qui permet d'obtenir de l'aide sur une fonction, par exemple `help(list.append)`.
- `dir` qui permet de lister les fonctions d'un module (par exemple `dir(math)`) ou les méthodes disponibles pour un objet (par exemple `dir(list)`).

Il y a également la fenêtre d'aide sous PYZO.

Lorsque l'on pose des hypothèses sur les arguments (par exemple qu'un argument est un entier naturel), il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée (par exemple si l'argument est strictement négatif).

Compression BZIP

BZIP est un algorithme de compression de données (et le nom d'un logiciel libre qui l'implémente) qui offre un bon taux de compression. Il est fondé sur la transformée de BURROWS-WHEELER qui est une méthode de réorganisation des données.

Le sujet comporte 4 parties qui ne sont pas indépendantes. Il est nécessaire de les lire et de les traiter dans l'ordre, mais il n'est pas nécessaire d'avoir intégralement résolu une partie pour passer à la suivante. Dans la première partie, on s'intéresse à l'encodage d'un texte par des listes d'entiers. La deuxième partie étudie un algorithme simple de compression par redondance. La troisième partie s'intéresse à la transformée de BURROWS-WHEELER qui permet d'améliorer l'algorithme de compression. Enfin, la dernière partie étudie la transformation inverse, afin de pouvoir décompresser un texte compressé.

I Encodage d'un texte par des entiers

Un texte est une (longue) suite de lettres. On pourrait donc naturellement choisir les chaînes de caractères (le type `str` en PYTHON) pour représenter un texte. Cependant, il nous sera plus aisé par la suite de manipuler des listes d'entiers plutôt que des chaînes de caractères. On commence donc par écrire les fonctions qui permettent d'encoder et de décoder un texte.

⚠ Dans toute la suite, une *lettre* désignera un entier entre 0 et 255 et un *texte* désignera une liste non vide de lettres (d'entiers, donc).

En PYTHON, on peut associer à un caractère son code UNICODE avec la fonction `ord`. Par exemple `ord('a')` renvoie l'entier `97`. Réciproquement, la fonction `chr` permet de retrouver le caractère associé à un code UNICODE. On a donc naturellement `chr(97)` qui renvoie la chaîne de caractères `'a'`.

Q 1. Écrire une fonction `encode(chaine)` qui encode une chaîne de caractères et qui renvoie le texte associé. Par exemple, `encode("Python")` doit renvoyer le texte `[80, 121, 116, 104, 111, 110]`. On suppose que la chaîne de caractères ne comporte que des caractères dont les codes Unicode sont des entiers entre 0 et 255 et on ne demande pas de le vérifier.

Q 2. Écrire une fonction `est_texte(liste)` qui vérifie qu'une liste d'entiers relatifs représente bien un texte, c'est-à-dire que la liste est non vide et que tous ses éléments appartiennent bien à l'intervalle $[\![0, 255]\!]$.

Q 3. Écrire une fonction `decode(texte)` qui décode un texte et qui renvoie la chaîne de caractères associée. Par exemple, `decode([80, 121, 116, 104, 111, 110])` doit renvoyer la chaîne de caractères `"Python"`. On pourra utiliser la méthode `join`. On suppose ici que l'argument est un texte et il n'est pas nécessaire de le vérifier.

II Compression par redondance

La compression par redondance compresse un texte d'entrée qui possède des répétitions consécutives de lettres. Dans un premier temps, on calcule les fréquences d'apparition de chaque lettre dans le texte d'entrée, puis on compresse le texte.

Q 4. Écrire une fonction `zeros(n)` prenant en entrée un entier `n` et renvoyant la liste PYTHON contenant n fois l'entier 0. Par exemple, `zeros(3)` doit renvoyer la liste `[0, 0, 0]`.

Q 5. Écrire une fonction `indice_du_minimum(liste)` qui renvoie l'indice de l'élément minimal d'une liste non vide d'entiers. Si plusieurs indices réalisent le minimum, on choisira l'indice le plus petit. Par exemple, `indice_du_minimum([42, 100, -12, 100, -12, 0])` vaut `2`.

Q 6. Écrire une fonction `compte_occurrences(texte)` qui prend en argument un texte et qui renvoie une liste `nb_occurrences` de taille 256 telle que `nb_occurrences[i]` corresponde au nombre d'occurrences de la lettre `i` dans le texte pour $0 \leqslant i < 256$.

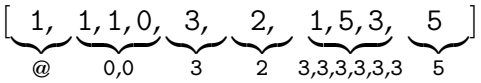
Q 7. Écrire une fonction `lettre_plus_rare(texte)` qui prend en argument un texte et qui renvoie la lettre qui apparaît le moins souvent dans le texte (le nombre d'occurrences de cette lettre peut être nul). Si plusieurs lettres sont possibles, on prendra la plus petite. Par exemple, `lettre_plus_rare([3, 42, 3, 1, 1, 0, 4])` vaut `2`.

La lettre la plus rare du texte servira de *marqueur*. On la note `@` par la suite. Pour simplifier, on suppose dans toute la suite que le marqueur n'apparaît pas dans le texte, autrement dit que son nombre d'occurrences est nul.

La compression par redondance d'un texte fonctionne comme suit :

- on ajoute `@` au début du texte ;
- toute apparition unique d'une lettre est codée par cette même lettre ;
- toute répétition maximale contiguë d'une lettre `k` entre les positions `i` et `j`, c'est-à-dire que l'on a `texte[i] = texte[i + 1] = ... = text[j] = k` et pas mieux, est codée par la suite formée des trois entiers `@, (j - i), k`.

Par exemple, pour le texte `[0, 0, 3, 2, 3, 3, 3, 3, 3, 3, 5]`, le marqueur est la lettre `1` (car `1` n'apparaît pas dans le texte) et le texte compressé est :



Q 8. Compléter la fonction `compression_par_redondance` qui prend en argument un texte et un marqueur (qui par hypothèse n'appartient pas au texte) et qui renvoie le texte compressé par redondance. Par exemple, `compression_par_redondance([0, 0, 3, 2, 3, 3, 3, 3, 3, 3, 5], 1)` doit renvoyer le texte compressé `[1, 1, 1, 0, 3, 2, 1, 5, 3, 5]`.

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

Q 9. (*Plus difficile*) Écrire une fonction `decompression_par_redondance` qui prend en argument un texte compressé par cette méthode et qui renvoie le texte original non compressé. On suppose que l'argument de cette fonction correspond bien à un texte correctement compressé et on ne demande pas de le vérifier.

III Transformation de BURROWS-WHEELER

Le codage par redondance n'est efficace que si le texte présente de nombreuses répétitions consécutives de lettres, ce qui n'est en général pas le cas pour un texte usuel. 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, pour lequel les répétitions de lettres ont tendance à être contiguës. Cette transformation est bijective.

Considérons par exemple la chaîne `concours`, correspondant au texte `[99, 111, 110, 99, 111, 117, 114, 115]`. Pour simplifier la présentation, nous utilisons ici des chaînes de caractères. Cependant, dans les programmes, on considère toujours, comme dans la première partie, des textes sous forme de listes 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`, et `sconcour`.

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

0 concour**s**
3 courscon**n**
2 ncoursco**o**
1 oncoursc**c**
4 ourscon**c**
6 rsconco**u**
7 sconcou**r**
5 ursconc**o**

3- 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 dernière colonne qui les précèdent dans le texte d'entrée. On peut le constater sur l'exemple : `concours de l ecole polytechnique` dont la transformée par BURROWS-WHEELER est `sleeeeen dlt ucn ooohcpcc iuryqol`.

En pratique, on ne va pas calculer ni stocker toutes les rotations du mot d'entrée. On va trier les rotations pour l'ordre lexicographique en faisant un nombre « raisonnable » de comparaisons entre certaines rotations.

On note r_i la i-ème rotation du mot. Ainsi, dans l'exemple, r_0 représente l'entrée `concours` et r_2 représente `ncoursco`.

Q 10. Modifier la fonction `compare_rotations(texte, i, j)` qui prend comme arguments un texte et deux indices i et j, pour qu'elle renvoie :

- 1 si r_i est strictement plus grand que r_j pour l'ordre lexicographique ;
- -1 si r_i est strictement plus petit que r_j pour l'ordre lexicographique ;
- 0 sinon.

Par exemple, `compare_rotations([99, 111, 110, 99, 111, 117, 114, 115], 0, 2)` doit renvoyer l'entier `-1`.

On donne une fonction `trie_rotations(texte)` qui trie les rotations d'un texte suivant l'ordre lexicographique. Elle renvoie une liste d'entiers représentant les indices des rotations dans l'ordre lexicographique (voir figure 1). Par exemple, si votre fonction `compare_rotations` est bien correcte, et si on note `t = encode("concours")`, alors l'appel `trie_rotations(t)` renvoie l'ordre `[0, 3, 2, 1, 4, 6, 7, 5]` que l'on note `r`.

Q 11. Écrire une fonction `cle_transformation_bw(ordre_rotations)` qui prend en argument la liste des indices des rotations dans l'ordre lexicographique (que l'on obtiendrait sur un texte avec la fonction précédente) et qui renvoie la clé de la transformation. Par exemple, `cle_transformation_bw(r)` doit renvoyer `3`. *Indication : quel est l'indice de la rotation qui envoie la première lettre du texte initial en dernière position ?*

Q 12. Écrire une fonction `transformation_bw(texte, ordre_rotations)` qui prend en paramètre un texte et l'ordre des rotations comme ci-dessus et qui renvoie la transformée de BURROWS-WHEELER du texte. Par exemple, `decode(transformation_bw(t, r))` doit renvoyer la chaîne de caractères `"snoccuro"`.

On peut en déduire une fonction de compression en combinant la transformée de BURROWS-WHEELER avec la compression par redondance. Cette fonction vous est donnée.

Le fichier `alice_in_wonderland.txt`, à la racine de votre répertoire, contient le texte original du livre *Alice au pays des merveilles* de Lewis CARROLL. L'appel `chaine_alice()` renvoie une chaîne de caractères correspondant à ce livre. Vous n'avez pas besoin de comprendre comment fonctionne cette fonction. Le ratio de compression d'une méthode est défini par la formule :

$$\frac{\text{taille}(\text{texte}_{\text{compressé}})}{\text{taille}(\text{texte}_{\text{original}})}$$

Q 13. Affecter les variables `RATIO_COMPRESSION_REDONDANCE` et `RATIO_COMPRESSION_BW` avec le taux de compression des deux méthodes pour le texte *Alice au pays des merveilles*.

IV Transformation de Burrows-Wheeler inverse

Cette partie, pour les plus rapides, est un peu plus difficile.

Pour retrouver l'original d'un texte à partir de sa transformée `transf`, en connaissant sa clé `c`, on construit d'abord une liste `transf_triee` de même taille que `transf` qui contient les mêmes lettres que `transf` dans l'ordre croissant. Dans notre exemple, correspondant au texte tranformé `"snoccuro"`, `trans = [115, 110, 111, 99, 99, 117, 114, 111]` et donc `trans_triee = [99, 99, 110, 111, 111, 114, 115, 117]`.

L'étape suivante consiste à aligner les listes `trans` et `lettres_tries`. À chaque lettre de `transf`, on associe la lettre de `transf_triee` qui est à la même position. À chaque lettre de `transf_triee`, on associe la même lettre de même rang relatif dans `transf`. La figure 1 illustre ces deux correspondances.

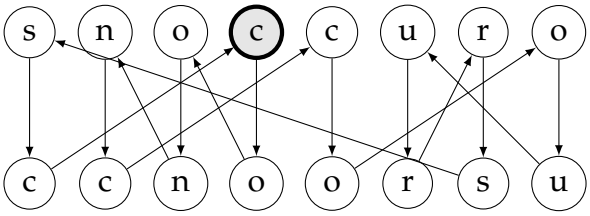


FIGURE 1 – Correspondance entre `transf` et `transf_triee`.

On retrouve alors le texte de départ `concours` en partant de la clé (mise en gras sur la figure) et en suivant les flèches.

Il nous faut donc, pour une lettre de la liste `trans_triee` avoir accès à l'indice de cette lettre dans le texte `transf`. On pourrait construire une liste d'indices à partir de `transf` et `transf_triee` mais il vaut mieux construire conjointement la liste triée et les indices. On obtient cela en annotant chaque lettre avec son indice avant de trier, ce que l'on peut réaliser assez simplement avec une liste de couples (lettre, indice).

Q 14. Écrire une fonction `ajoute_indices(liste)` qui pour une liste $[x_0, x_1, \ldots, x_{n-1}]$ en argument renvoie la liste de couples $[(x_0, 0), (x_1, 1), \ldots, (x_{n-1}, n-1)]$. Par exemple l'appel `ajoute_indices([3, 1, 3, 2])` doit renvoyer `[(3, 0), (1, 1), (3, 2), (2, 3)]`. On dit que l'on a « marqué » les éléments de la liste par leurs indices dans cette liste.

Q 15. Écrire une fonction `trie_lettres(texte_marque)` qui trie les couples (lettre, indice) d'un texte marqué (avec la fonction précédente) suivant l'ordre lexicographique pour les couples (on trie par rapport aux lettres et pour une même lettre on conserve l'ordre des indices). Par exemple, `trie_lettres([(3, 0), (1, 1), (3, 2), (2, 3)])` doit renvoyer la liste `[(1, 1), (2, 3), (3, 0), (3, 2)]`

Indication : on pourra s'inspirer et généraliser la Q 6. en commençant par créer une liste de taille 256 *initialisée avec des listes vides (différentes !). Dans la case d'indice i de cette liste on ajoutera successivement la lettre i avec chacun de ses indices rencontrés. Il suffira ensuite de parcourir les listes dans le bon ordre pour obtenir le résultat désiré.*

Q 16. Écrire une fonction `inverse_bw(trans, cle)`, qui prend en arguments une transformée de BURROWS-WHEELER d'un texte et la valeur de la clé, et qui retrouve le texte d'origine.

Q 17. Écrire une fonction `decompression_bw(compr, cle)`, qui prend en arguments un texte compressé par la méthode de BURROWS-WHEELER et la valeur de la clé, et qui retrouve le texte d'origine.

— FIN DU SUJET —