IC Devoir 3 - 2h - 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 $\mathcal{O}(n^\alpha)$, s'il existe $K>0$ tel que pour tout n, $T_n(f) \leq Kn^\alpha$.
Afin d'optimiser la complexité, on n'utilisera pas les méthodes connues du type `list` de PYTHON.

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` (liste) contenant des entiers compris entre 0 et 255 inclus.

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 `t` ; et qui retourne un tableau `r` de taille 256 tel que `r[i]` est le nombre d'occurrences de `i` dans `t` pour $0 \leq i < 256$.
2. Écrire la fonction `min(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'occurrences de cet entier peut être nul)

L'entier `min(t)` servira de *marqueur*. On note `m` pour ce marqueur et, pour simplifier, on suppose que son nombre d'occurrences est nul. Donc `r[m]` contient la valeur 0 quand `r=occurrences(t)`.
La compression par redondance du texte `t` fonctionne comme suit :

- toute répétition maximale contigüe d'une lettre où `t[i]`, `t[i+1]`, ..., `t[j]` contiennent `k` (avec `t[i-1]` et `t[j+1]` ne contenant pas `k`) est codée par les trois entiers `m,(j-i), k`
- toute apparition unique d'une lettre est codée par cette même lettre.

Par exemple, considérons le tableau `t` :

`[0,0,3,2,3,3,3,3,3,3,5]`

Le marqueur est donc 1 car 1 n'apparaît pas dans ce tableau. Le texte `tt` compressé est alors `[1,1,1,0,3,2,1,5,3,5]` :

$$\underbrace{1}_{m},\underbrace{1,1,0}_{0,0},\underbrace{3}_{3},\underbrace{2}_{2},\underbrace{1,5,3}_{3,3,3,3,3},\underbrace{5}_{5}$$

3. Compléter la fonction `tailleCodage(t)` suivante (au proposer une fonctionne équivalente de votre choix) qui prend comme argument le tableau `t` ; et qui calcule la taille `nn` du texte compressé. Dans l'exemple précédent, `nn` vaut 10.

```
1  def tailleCodage(t):
2      nn=1   # le marqueur
3      i,j=0,1
4      while i+j<len(t):
5          # calcul du nombre de repetitions
6          if t[i+j]==t[i]:
7              j=...
8          # si la lettre est isolee
9          elif j==1:
10             nn,i=...
11         # si la lettre n'est pas isolee
12         else:
13             nn,i,j=...
14     # gestion de la derniere lettre
15     if ...:
16         ...
17     else:
18         ...
19     return(nn)
```

4. Écrire la fonction `codage(t)` qui prend comme paramètre le tableau `t` ; et qui retourne un tableau d'entiers `tt` représentant le texte compressé.

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

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 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 :

(i) – 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
```

(ii) – On trie ces rotations par ordre lexicographique (l'ordre du dictionnaire).

```
concours
courscon
ncoursco
oncoursc
oursconc
rsconcou
sconcour
ursconco
```

(iii) – 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` dont la transformée par Burrows-Wheeler est `sleeeeen dlt ucn ooohcpcc iuryqo`.

5. Écrire une fonction `rot(t,i)` qui prend comme arguments le texte `t` ; et qui retourne sa ième rotation, c'est-à-dire les lettres `t[i]`, `t[i+1]` , ... Par exemple, `rot(t,0)` retourne le texte d'entrée `concours`, `rot(t,1)` retourne `oncoursc`, `rot(t,2)` retourne `ncoursco`, etc.

6. Écrire la fonction `comparerRotations(t,i,j)` qui prend comme arguments le texte `t` et deux indices `i`, `j` ; et qui renvoie, en temps linéaire par rapport à `n` :
- 1 si la ième rotation est plus grand que la jème dans l'ordre lexicographique,
- -1 si la ième rotation est plus petit que la jème dans l'ordre lexicographique,
- 0 sinon.

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 un tableau d'entiers `r` représentant les numéros des rotations ordonnées (le résultat de `rot(t,r[0])` est plus petit que celui de `rot(t,r[1])`, qui est plus petit que celui de `rot(t,r[n-1])` , etc.). Cette fonction réalise dans le pire des cas $\mathcal{O}(n\ln(n))$ appels à la fonction de comparaison où n est la longueur de `t`.

7. Écrire une fonction `codageBW(t)` qui prend en paramètre le tableau `t` ; et qui renvoie un tableau contenant le texte après transformation. (La clé sera stockée dans la dernière case de ce tableau)

8. Donner un ordre de grandeur du temps d'exécution de la fonction `codageBW` en fonction de `n` la taille de `t`.

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

III. Transformation de Burrows-Wheeler inverse

Pour décoder le texte `tt` (`snoccuro3` dans l'exemple) de taille `nn` de contenu égal à `n+1` obtenu après transformation, on construit d'abord un tableau `triCars` de taille `n` qui contient les mêmes lettres que le texte `tt` mais dans l'ordre lexicographique croissant. Dans l'exemple, `triCars` contient `[c,c,n,o,o,r,s,u]`.

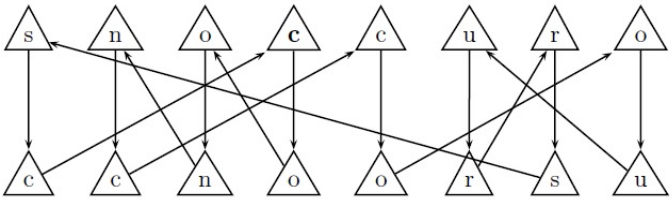
9. Écrire une fonction `frequences(tt)` qui prend comme argument un tableau `tt` correspondant au texte codé (avec la clé dans la dernière case) ; et qui renvoie un tableau de taille 256 contenant le nombre d'occurrences de chaque lettre dans `tt`.

10. Écrire la fonction `triCarsDe(tt)` qui part du texte codé `tt` de taille `nn` ; et qui renvoie, en temps linéaire par rapport à `n`, le tableau `triCars` décrit précédemment.

Puis on considère le texte codé `tt` et le tableau `triCars` précédent (la clé est représentée soulignée).

s	n	o	**c**	c	u	r	o	3
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 le tableau `indices` tel que `indices[i]` est l'indice de la lettre `triCars[i]` dans le texte `tt`. Si plusieurs occurrences de cette lettre figurent dans `tt`, on fait correspondre celle qui figure au même rang dans `tt`. Le tableau `indices` donne donc la correspondance représentée par les flèches de la seconde ligne vers la première. Sur l'exemple, le tableau `indices` contient les valeurs `[3,4,1,2,7,6,0,5]`.

11. Écrire la fonction `trouverIndices(tt)` prenant en paramètre le texte `tt` codé ; et qui retourne le tableau `indices` précédemment décrit.
Quel est son temps d'exécution en fonction de `n` la longueur du mot initial ?

12. Écrire une fonction `decodageBW(tt)` qui prend comme paramètre un texte `tt` ; et retourne le texte `t` d'origine.
Quel est son temps d'exécution en fonction de `n` ?

IV. Compléments - (bonus)

13. Écrire une fonction `decodage(tt)` qui prend comme argument le tableau `tt` représentant un texte compressé par redondance et retourne le tableau `t` du texte originel.
14. Écrire un fonction `numerise(mot)` qui prend comme argument une chaine de caractères `mot` et qui retourne la tableau d'entiers associés. On utilisera les fonctions Python `chr` et `ord`. Exemple

```
>>> ord('A')
65
>>> chr(65)
'A'
```

15. Écrire un fonction `denumerise(t)` qui prend comme argument un tableau `t` d'entiers entre 0 et 255 et qui retourne la chaine de caractères associée.
16. Écrire la fonction `triRotations(t)`, définie à la partie 2) qui trie les rotations du texte donné dans le tableau `t` en utilisant la fonction `comparerRotation`. En particulier, le tri fusion permet d'obtenir la complexité recherchée.
17. Écrire les fonctions `compresseBW(mot)` et `decompresseBW(tt)` qui finalisent le travail.

TD 3- Proposition de solutions

V. Compression par redondance

1. Fonction `occurences(t)` :

```
1 def occurrences(t):
2     r=[0]*256
3     for e in t:
4         r[e]=r[e]+1
5     return(r)
```

- ligne 2 : création d'une liste de zéros de taille 256
- lignes 3 à 4 : on parcourt le texte, et comptabilise chaque caractère rencontré.

Remarque : La boucle `for` parcout directement les éléments de `t` et non les positions des éléments de `t`

2. Fonction `min(t)`

```
  def min(t):
      r=occurrences(t)
      m=0
      for i in range(1,256):
          if r[i]<r[m]:
              m=i
      return(m)
```

Fonction classique de recherche d'un minimum dans une liste. Attention, ici on veut la position du minimum dans une liste de 256 éléments : cette position est le caractère le moins présent dans le texte.

3. Fonction `tailleCodage(t)`

```
1 def tailleCodage(t):
2     nn=1 # le marqueur
3     i,j=0,1
4     while i+j<len(t):
5         # calcul du nombre de repetitions
6         if t[i+j]==t[i]:
7             j=j+1
8         # si la lettre est isolee
9         elif j==1:
10            nn,i=nn+1,i+1
11        # si la lettre n'est pas isolee
12        else:
13            nn,i,j=nn+3,i+j,1
14    # gestion de la derniere lettre (repetee)
15    if j==1:
16        nn=nn+1
17    else:
18        nn=nn+3
19    return(nn)
```

- ligne 2 : l'indice `i` permet de parcourir le texte lettre après lettre ; l'indice `j` dénombre les répétitions éventuelles ; la variable `nn` est initialisée à 1 pour le marqueur.
- lignes 3 à 12 : pour chaque lettre, il y a trois possibilités :

➢ soit c'est une répétition de la précédente : dans ce cas `j` est augmenté de 1 et on considère la lettre suivante

➢ soit elle est différente et la précédente était isolée (`j` contient 1) alors la lettre est codée par 1 caractère (la lettre elle-même) : `nn=nn+1`

➢ soit elle est différente et la précédente n'était pas isolée (`j>1` répétitions) alors la lettre est codée par 3 caractères (`m`, `j-1` et la lettre elle-même) : `nn=nn+3`

- lignes 14 à 17 : gestion de la dernière lettre suivant si elle est répétée ou non

Exemple :

```
>>> t=[0,0,3,2,3,3,3,3,3,3,5]
>>> tailleCodage(t)
10
```

4. Fonction `codage(t)`

```
1 def codage(t):
2     m=min(t)
3     tt=[m]
4     i,j=0,1,
5     while i+j<len(t):
6         if t[i+j]==t[i]:
7             j=j+1
8         elif j==1:
9             tt=tt+[t[i]]
10            i=i+1
11        else:
12            tt=tt+[m,j-1,t[i]]
13            i,j=i+j,1
14    if j==1:
15        tt=tt+[t[i]]
16    else:
17        tt=tt+[m,j-1,t[i]]
18    return(tt)
```

Cette fonction est construite sur la même démarche que `tailleCodage`.

Attention ! Éviter les affectations simultanées qui mélangeant des indices et des tableaux ; l'ordre des instructions impacte le résultat.

⇛ Un autre approche avec un boucle `for` est :

```
def codage(t):
    v,m,r=t+[-1],min(t),0
    tt=[m]
    for i in range(len(t)):
        if v[i]!=v[i+1]:
            if r==0:
                tt.append(v[i])
            else :
                tt=tt+[m,r,v[i]]
            r=0
        else:
            r=r+1
    return(tt)
```

• ligne 2 : l'ajout d'une valeur "impossible" à t (renommer par v) êrùet de gérer efficacement la dernière lettre.
La variable r vaut 0 par défaut dès que l'on rencontre une nouvelle lettre, sinon elle est incrémentée.
Exemple :

```
>>> t=[0,0,3,2,3,3,3,3,3,3,5]
>>> codage(t)
[1, 1, 1, 0, 3, 2, 1, 5, 3, 5]
```

VI. Transformation de Burrows-Wheeler

5. Fonction rot(t,i)

```
rot=lambda t,i:t[i:]+t[:i]
```

6. Fonction comparerRotations(t,i,j)

```
def comparerRotations(t,i,j):
    a,b=rot(t,i),rot(t,j)
    for k in range(len(t)):
        if a[k]>b[k]:
            return(1)
        elif a[k]<b[k]:
            return(-1)
    return(0)
```

Autre version n'utilisant pas rot :

```
1 def comparerRotations(t,i,j):
2     n=len(t)
3     for k in range(n):
4         if t[(i+k)%n]>t[(j+k)%n]:
5             return(1)
6         elif t[(i+k)%n]<t[(j+k)%n]:
7             return(-1)
8     return(0)
```

La comparaison lexicographique consiste à comparer les caractères des deux objets l'un après l'autre en partant de la gauche. Dès que les caractères diffèrent, on peut conclure. Si le parcours n'a pas permis de conclure, c'est que les mots sont identiques.

Ici, on peut travailler directement sur le tableau t en considérant comme points de départ des deux mots les positions i et j dans t.

Attention ! Ligne 3 et 5 : on note l'importance de veiller à ce que l'indice de position soit toujours parmi les valeurs possibles, d'où l'introduction d'une congruence.

7. Fonction codageBW(t)

```
1 def codageBW(t):
2     r=triRotations(t)
3     tt=[t[e-1] for e in r]
4     for i in range(len(r)):
5         if r[i]==1:
6             cle=i
7     return(tt+[cle])
```

Les étapes sont :
• ligne 2 : effectue le tri des rotations
• ligne 3 : extraire les dernières lettres de chaque rotation : c'est-à-dire la position qui précède celle de début de la rotation, d'où le t[e-1]. L'utilisation d'une liste définie par compréhension est efficace à écrire
• ligne 4 à 6 : le clé est la position de la rotation commençant par la deuxième lettre du texte. Nous avons donc cherché la position de 1 dans r.
Exemples utilisant les fonctions introduites dans les compléments :

```
>>> t=numerise('concours')
>>> tt=codageBW(t)
>>> denumerise(tt[:-1])
snoccuro
>>> tt[-1]
3
>>> t=numerise('concours de l ecole polytechnique')
>>> tt=codageBW(t)
>>> denumerise(tt[:-1])
sleeeeen dlt ucn ooohcpcc iuryqol
>>> tt[-1]
23
```

8. Étude de la complexité temporelle de la fonction codageBW en fonction de n :
• l'appel à la fonction triRotations(t) admet une complexité en $\mathcal{O}(n\ln(n))$ d'appels à la fonction de comparaison comparerRotations qui est de complexité n : soit un complexité globale de $\mathcal{O}(n^2\ln(n))$
• la construction du texte transformé à partir de la liste triée des rotations, revient à parcourir le texte une fois : $\mathcal{O}(n)$
• la recherche de la clé revient à parcourir le texte à nouveau : $\mathcal{O}(n)$
Par addition des complexités, puisque ces opérations sont séquentielles, il vient que la complexité temporelle de la fonction codageBW pour un mot de n lettres est $\mathcal{O}(n^2\ln(n))$.

VII. Transformation de BW inverse

9. Fonction `frequences(tt)`

```
def frequences(tt):
    f=256*[0]
    for e in tt[:-1]:
        f[e]=f[e]+1
    return(f)
```

Cette fonction est une variante de la fonction occurrences en tenant compte du fait que le dernier caractère est la clé et ne doit pas être comptabilisé.
On aurait pu donner la définition suivante :

```
frequences=lambda tt:occurences(tt[:-1])
```

10. Fonction `triCarsDe(tt)`

```
def triCarsDe(tt):
    f=frequences(tt)
    triCars=[]
    for i in range(256):
        triCars=triCars+f[i]*[i]
    return(triCars)
```

Exemple :

```
>>> t=numerise('concours')
>>> tt=codageBW(t)
>>> denumerise(triCarsDe(tt))
ccnoorsu
```

On retrouve bien la première colonne de la phase 2 de la transformation.

11. Fonction `trouverIndices(tt)`

```
1 def trouverIndices(tt):
2     ind=256*[[]]
3     for i in range(len(tt)-1):
4         ind[tt[i]]=ind[tt[i]]+[i]
5     indices=[]
6     for i in ind:
7         indices=indices+i
8     return(indices)
```

- lignes 2 à 4 : on procède comme dans frequences mais cette fois, on enregistre la position de l'élément rencontré.
- lignes 5 à 7 : il suffit maintenant de concaténer les sous-listes de `ind` ; les positions sont respectivement celles des éléments de `triCars` dans `tt` puisque dans `triCars` les lettres sont dans l'ordre lexicographique.

La complexité de cette fonction est $\mathcal{O}(\text{nn}) = \mathcal{O}(n)$ car $\text{nn} = n-1$.
En effet, il y a deux parcours (en séquentiel) d'un tableau de longueur nn.
D'autres idées étaient possibles :
⇛ L'idée suggérée par l'énoncé est de rechercher dans `tt` la position des lettres de `triCars`. Comme `tt` est quelconque, pour chaque nouvelle lettre de `triCars`, il convient de recommence la recherche au début de `tt`. En revanche, pour la recherche de lettres qui sont répétées, comme elles sont consécutives dans `triCars`, il suffit de poursuivre le parcours dans `tt` afin de repérer la nouvelle occurrence. La complexité est de $\mathcal{O}(n^2)$.
⇛ On peut utiliser le fait que la liste `triCars` est ordonnée et donc au lieu de chercher la position des lettres de `triCars` dans `tt` (qui est aléatoire), on fait l'inverse : on cherche la position des lettres de `tt` dans `triCars`. Ensuite, il suffit de renverse l'information, ce qui nécessite un simple parcours de la liste. La complexité est $\mathcal{O}(n\ln(n))$.

12. Fonction `decodageBW(tt)`

```
1 def decodageBW(tt):
2     indices=trouverIndices(tt)
3     k=tt[-1] # la cle
4     t=[tt[k]]
5     for i in range(len(tt)-2):
6         k=indices[k]
7         t.append(tt[k])
8     return(t)
```

La fonction suit la démarche proposée en procédant de proche en proche pour reconstruire le texte originel.
La complexité de `decodageBW(tt)` est :
- ligne 2 : $\mathcal{O}(n)$ (voir question précédente)
- ligne 3 à 7 : $\mathcal{O}(n)$ (la travail est linéaire en la taille du texte)

Additionnant ces complexités car les opérations sont séquentielles, on trouve que la complexité de `decodageBW(tt)` est $\mathcal{O}(n)$.
Exemple :

```
>>> t=numerise('concours')
>>> tt=codageBW(t)
>>> t=decodageBW(tt)
>>> denumerise(t)
concours
```

VIII. Compléments - (bonus)

13. Fonction `decodage(tt)`

```
def decodage(tt):
    m,i,t=tt[0],1,[]
    while i<len(tt):
        # cas d'une lettre repetee
        if tt[i]==m:
            t=t+(tt[i+1]+1)*[tt[i+2]]
            i=i+3
        # cas d'une lettre isolee
        else:
            t=t+[tt[i]]
            i=i+1
    return(t)
```

Exemple :

```
>>> t=[0,0,3,2,3,3,3,3,3,3,5]
>>> tt=codage(t)
>>> decodage(tt)
[0, 0, 3, 2, 3, 3, 3, 3, 3, 3, 5]
```

14. Fonction `numerise(mot)`

```
def numerise(mot):
    return([ord(e) for e in mot])
```

Exemple :

```
>>> numerise('concours')
[99, 111, 110, 99, 111, 117, 114, 115]
```

15. Fonction `denumerise(t)` :

```
def denumerise(t):
    return(''.join([chr(e) for e in t]))
```

Exemple :

```
>>> denumerise(t)
concours
```

16. Fonction `triRotations(t)`

Nous mettons en place le tri fusion qui répond favorablement à la contrainte de complexité. Pour cela il suffit d'adapter le tri vu en cours avec l'ordre induit par la fonction `comparerRotations`.

```
def fusion(t,L1,L2):
    if len(L1)==0:
            return(L2)
    if len(L2)==0:
        return(L1)
    if comparerRotations(t,L1[0],L2[0])==-1:
        return([L1[0]]+fusion(t,L1[1:],L2))
    else:
        return([L2[0]]+fusion(t,L1,L2[1:]))

def tri_fusion(t,L):
    if len(L)<2:
        return(L)
    else:
        k=len(L)//2
        return(fusion(t,tri_fusion(t,L[:k])
                ,tri_fusion(t,L[k:])
```

```
def triRotations(t):
    r=list(range(len(t)))
    return(tri_fusion(t,r))
```

La complexité du tri fusion est $\mathcal{O}(n\ln(n))$ en l'opération de comparaison. Ici, la comparaison des objets est réalisé par `compareRotations` qui consite à un parcours du texte soit $\mathcal{O}(n)$. La complexité globale est donc de $\mathcal{O}(n^2\ln(n))$.

17. Fonctions `compresseBW(mot)` et `decompresseBW(tt)`

```
def compresseBW(mot):
    t=numerise(mot)
    T=codageBW(t)
    return(codage(T))
```

```
def decompresseBW(tt):
    tt=decodage(tt)
    t=decodageBW(tt)
    return(denumerise(t))
```

Exemple :

```
>>> tt=compresseBW('concours')
>>> tt
[0, 115, 110, 111, 0, 1, 99, 117, 114, 111, 3]
>>> decompresseBW(tt)
concours
```