> Dans tout l'énoncé, l'argument `n` des fonctions a été supprimé car il est accessible avec la fonction `len` en Python.

1. Compression par redondance

1 On crée un tableau `r` de taille 256 pour stocker les résultats. Puis, pour chaque lettre x de `t`, on incrémente la case correspondante.

```
def occurrences(t):
    r=[0 for i in range(256)]
    for x in t:
        r[x]+=1
    return r
```

> Ne pas oublier d'initialiser le tableau `r`, c'est-à-dire d'en remplir toutes les cases à la création. C'est le rôle de la première boucle `for`, qui écrit 0 partout.

2 Commençons par calculer le tableau `r` contenant le nombre d'occurrences de chaque lettre grâce à la question 1. On parcourt ensuite ce tableau en conservant en mémoire l'indice `m` de la position de la plus petite valeur rencontrée.

```
def min(t):
    r=occurences(t); m=0
    for i in range(len(t)):
        if t[i]<t[m]:
            m=i
    return m
```

> Le nom de fonction imposé par l'énoncé, « `min` », n'est pas très heureux. Il s'agit en effet du nom d'une fonction déjà existante dans la plupart des langages. Dans une copie écrite, il est néanmoins préférable de se conformer au choix de l'énoncé.

3 On parcourt le tableau à l'aide d'un compteur `occ` qui compte les occurrences consécutives de la dernière lettre lue. La longueur du codage `l` est calculée au fur et à mesure. Le parcours démarre à l'indice 1 avec un compteur initialisé à 1 (qui correspond au départ au nombre d'occurences de la première lettre).

- Tant qu'on lit des lettres identiques, on se contente d'incrémenter le compteur `occ`.
- Dès qu'une nouvelle lettre se présente, on ajoute à `l` la longueur du motif qui code les occurences de la lettre précédente : 1 si elle apparaît une seule fois, ou 3 si elle apparaît au moins deux fois.

A noter qu'en fin de parcours, il faut refaire un calcul pour prendre en compte le nombre d'occurrences de la toute dernière lettre.

```
def taillecodage(t):
    l=1; occ=1
    for i in range(1,len(t)):
        if t[i]==t[i-1]:
            occ+=1
        else:
            if occ==1:
                l+=1
```

```
            else:
                l+=3
            occ=1
    if occ==1:
        l+=1
    else:
        l+=3
    return l
```

> Dans une application réelle, on peut supposer que les entiers du texte encodé seront notés chacun par une lettre dans le fichier compressé final. Ils doivent donc être dans l'intervalle $[0; 255]$. Si dans le texte à encoder une lettre ℓ est répétée plus de 256 fois, par exemple 267 fois, on peut coder ce bloc par $\#, 255, \ell, \#, 10, \ell$, ce qui signifie « $256 + 11$ répétitions de ℓ ».

4 Le code est sensiblement le même si ce n'est qu'on construit le codage au fur et à mesure au lieu de simplement calculer sa longueur. Au départ, `l` est réduit à # qui est calculé par un appel à `min`. Le reste du code est quasiment inchangé si ce n'est qu'on ajoute soit une lettre, soit un triplet en bout de la chaîne `l` à chaque fois que l'on découvre un nouveau caractère.

```
def codage(t):
    p=min(t); l=[p]; occ=1; n=len(t)
    for i in range(1,n):
        if t[i]==t[i-1]:
            occ+=1
        else:
            if occ==1:
                l=l+[t[i-1]]
            else:
                l=l+[p,occ-1,t[i-1]]
            occ=1
    if occ==1:
        l=l+[t[n-1]]
    else:
        l=l+[p,occ-1,t[n-1]]
    return l
```

> La fonction `taillecodage` n'est pas nécessaire en Python puisque l'on peut concaténer des chaînes de caractères. A l'époque du sujet, le langage le plus utilisé était Maple et le sujet ne permettait pas explicitement cette opération. C'est la raison pour laquelle il invitait à calculer d'abord la longueur du codage dans la question 3, pour ensuite initialiser un tableau de la bonne taille en question 4, avant de le remplir. Si on ne suppose plus que le marqueur n'apparaît pas dans le texte à encoder, on peut le coder par « $\#, 0, \#$ ». La seconde position du triplet est le nombre de répétitions moins un, car même une occurrence isolée du marqueur est codée par un triplet.

2. Transformation de Burrows-Wheeler

5 Dans cette question, tous les indices sont pris modulo n où n est la longueur du tableau. Rappelons que, par définition, la lettre d'indice k dans `rot[i]` est `t[i+k]`. Pour comparer `rot[i]` et `rot[j]`, on compare donc `t[i]` à `t[j]`, puis, en cas d'égalité, `t[i+1]` à `t[j+1]`, et ainsi de suite. Si on atteint la fin de la boucle, c'est que les deux rotations ont exactement les mêmes lettres, donc que les mots sont égaux. Il faut donc renvoyer 0.

```
def comparerRotations(t,i,j):
    n=len(t)
    for k in range(n):
        if t[(i+k)%n]<t[(j+k)%n]:
            return -1
        if t[(i+k)%n]>t[(j+k)%n]:
            return 1
    return 0
```

6 Définissons `r` comme proposé par l'énoncé, à l'aide de `triRotations`. Commençons par chercher la clef : c'est l'index `i` tel que la rotation rot[`r[i]`] se termine par la première lettre du texte d'origine, c'est-à-dire tel que `r[i]` $= 1$ (c'est en effet la rotation où les lettres ont été décalées une fois vers la gauche). Ceci est fait dans la boucle `while`.

Créons alors un tableau `s` que l'on renverra. La case `i` de `s` contient la dernière lettre de rot[`r[i]`], autrement dit la lettre de `t` indexée par `r[i]` $+ (n-1) \mod n$.

```
def codageBW(t):
    r=triRotations(t); n=len(t)
    ind=0
    while r[ind]!=1:
        ind+=1
    s=[]
    for i in range(n):
        s.append(t[(r[i]+n-1)%n])
    s.append(ind)
    return s
```

> La fonction `triRotations` qui n'est pas demandée par l'énoncé peut s'écrire de la manière suivante :
>
> ```
> def triRotations(t):
>     n=len(t); r=[i for i in range(n)]
>     def fusion(d,m,f):
>         l=[]; i=d;j=m
>         while (i<m) and (j<=f):
>             if comparerRotations(t,r[i],r[j])>0:
>                 l.append(r[j]); j+=1
>             else:
>                 l.append(r[i]); i+=1
>         while i<m:
>             l.append(r[i]); i+=1
>         for k in range(len(l)):
>             r[d+k]=l[k]
> ```

```
    def tri_entre(a,b):
        if a<b:
            m=(a+b)//2
            tri_entre(a,m)
            tri_entre(m+1,b)
            fusion(a,m+1,b)
    tri_entre(0,n-1)
    return r
```

L'implémentation proposée ici est celle d'un tri par partition-fusion. Contrairement aux algorithmes de tris naïfs comme le tri bulle ou le tri par insertion, celui-ci est de complexité quasi-linéaire (c'est à dire en $O(n \ln n)$ dans le pire cas).

7 Les deux boucles `while` et `for` de la fonction `codageBW` s'exécutent en temps $O(n)$, car elles ne contiennent que des opérations élémentaires. Il en va de même pour la création du tableau.

Reste à compter le temps d'exécution de l'appel à `triRotations`. Cette fonction fait $O(n \log n)$ appels à la fonction de comparaison ($O(n \log n)$ n'est donc pas directement la complexité de `triRotations`). Or, la fonction `comparerRotations`, qui génère au plus n comparaisons de lettres, s'exécute en temps $O(n)$. On en déduit que `triRotations` prend un temps $O(n \times n \log n)$. Au total,

Le temps d'exécution de `codageBW` est $O(n) + O(n^2 \log n) = O(n^2 \log n)$.

Attention ici à bien lire le sujet : l'ordre de grandeur $O(n \ln n)$ est le nombre de **comparaisons** effectuées par l'algorithme de tri `triRotations`, ce n'est donc pas le nombre d'opérations élémentaires effectuées, qui est plus important. Ce pire cas est atteint quand toutes les comparaisons prennent effectivement un temps proportionnel à n (au moins en moyenne sur toutes les comparaisons). Cela se produit par exemple pour le mot de longueur n « abababab... ».

3. Transformation de Burrows-Wheeler inverse

8 Il suffit d'utiliser la fonction `occurrences` avec t' et $n' - 1$.

L'énoncé rappelle gentiment que la dernière case ne contient pas de lettre.

```
def frequences(t2):
    return occurrences(t2[:-1])
```

9 Grâce à un tableau `freqs` contenant la fréquence de chaque lettre calculé par la question précédente, il suffit d'écrire dans le tableau `triCars` les lettres dans l'ordre, chacune autant de fois qu'elle apparaît dans le texte d'origine.

```
def triCars(t2):
    occ=frequences(t2)
    s=[]
    for i in range(256):
        s=s+[i]*occ[i]
    return s
```

Cette fonction exécute une opération élémentaire par lettre du tableau `triCars` et s'exécute donc en $O(n)$.

10 On peut proposer deux réponses à cette question : une première version, naïve, mais de complexité quadratique, et une seconde de complexité linéaire mais plus difficile à trouver et qui nécessite de modifier les codes des questions précédentes. On peut raisonnablement penser que la seconde version n'était pas attendue par les correcteurs de l'épreuve, et que la première suffit pour obtenir tous les points à cette question. C'est par conséquent celle-ci qui est rédigée ici. La seconde version (qui en réalité est plus simple à écrire) est proposée en remarque.

Définissons tout d'abord un tableau `rangs`, tel que `rangs[i]` contient le rang de la lettre `triCars[i]`. Par exemple, si `triCars` est

$$[a, a, b, e, e, e, n, s]$$

`rangs`'écrit alors $[1, 2, 1, 1, 2, 3, 1, 1]$

Puis l'on crée et remplit un tableau `indices` en suivant directement la définition : on lit une à une les lettres de `triCars` (stockées dans une variable `s`) et leur rang (stocké dans la variable `compteur`), on parcourt t' en décrémentant `compteur` chaque fois que l'on rencontre cette lettre, et quand on en a rencontré r, on écrit la position de la dernière occurence de la lettre cherchée dans `indices`. Voici le code :

```
def trouverIndices(t2):
    s=triCars(t2); n=len(s)
    rang=[1]
    for i in range(1,n):
        if s[i]==s[i-1]:
            rang.append(rang[-1]+1)
        else:
            rang.append(1)
    indices=[]
    for i in range(n):
        compte=rang[i]; j=0
        while compte>0:
            if t2[j]==s[i]:
                compte-=1
            j+=1
        indices.append(j-1)
    return indices
```

Pour chacune des n' lettres de `resultat`, on parcourt $O(n) = O(n')$ lettres de t', le coût de `trouverIndicesSimple` est donc $O(n'^2)$.

Une méthode plus simple consisterait à réécrire la fonction `frequences` en adaptant la fonction `occurrences`. Expliquons un schéma de cette méthode. Dans la nouvelle fonction `frequences`, on stocke dans `resultat[i]` non pas le nombre de lettres i rencontrées mais la liste des positions de ces lettres dans le texte d'origine. Quand dans `occurrences` on lisait la lettre `t[i]`, on ajoutait 1 à la case `resultat[t[i]]`. À la place, on ajoute donc maintenant la position i à la liste `resultat[t[i]]`. Puisqu'on lit les lettres du texte d'origine dans l'ordre, les listes sont triées.

Dans l'ancienne version, si `freqs[i]` contenait k, on écrivait k fois (la lettre i) dans `triCars`. La case `resultat[i]` de la nouvelle fonction `frequences` contient non pas k, mais les k positions de la lettre i dans le texte d'origine. Il suffit donc d'écrire ces k positions, à la suite, dans le tableau `indices` demandé par l'énoncé.

```
def occurrences2(t):
    r=[[] for i in range(256)]
    for i in range(len(t)):
        r[t[i]].append(i)
    return r

def frequences2(t2):
    return occurrences2(t2[:-1])

def trouverIndices2(t2):
    indices=[]
    occ=frequences2(t2)
    for x in occ:
        indices=indices+x
    return indices
```

Pourquoi cette transformation permet-elle de retrouver le texte original ? Expliquons cela à partir de l'exemple « MISSISSIPI », qui a suffisamment de lettres répétées pour éviter de tomber, par chance, sur un cas qui marcherait tout seul. Pour distinguer les lettres identiques, numérotons-les : $MI_1S_1S_2I_2S_3S_4I_3PI_4$.

Explicitons la signification des flèches sur la figure de l'énoncé : une flèche descendante indique quelle est la lettre suivante dans le texte décodé ; une flèche montante indique où a été déplacée cette lettre dans le texte encodé.

Calculons tout d'abord les rotations du mot :

$M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4$
$I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M$
$S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1$
$S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1\;S_1$
$I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1\;S_1\,S_2$
$S_3\,S_4\,I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2$
$S_4\,I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3$
$I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4$
$P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3$
$I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P$

Ce qui donne, une fois ces rotations triées par ordre alphabétique :

$I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P$
$I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,\mathbf{S_4}$
$I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1\;S_1\,\mathbf{S_2}$
$I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M$
$M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4$
$P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;S_3\,S_4\,I_3$
$\mathbf{S_4}\,I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2\;\mathbf{S_3}$
$\mathbf{S_2}\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1\;\mathbf{S_1}$
$\mathbf{S_3}\,S_4\,I_3\;P\;I_4\;M\,I_1\;S_1\,S_2\,I_2$
$\mathbf{S_1}\,S_2\,I_2\;S_3\,S_4\,I_3\;P\;I_4\;M\,I_1$

Connaissant le texte encodé (« pssmiissii »), qui est la dernière colonne du second tableau, il est facile de reconstituer la première colonne de ce tableau, cette colonne est en effet `triCars`. Puisque la première ligne de ce tableau est une rotation du texte original, on sait déjà que P sera suivi de l'un des I. Il en va de même pour chaque ligne. Ceci permet de justifier les flèches descendantes (verticales) de la dernière figure de l'énoncé.

Il ne reste plus qu'à justifier les flèches montantes de cette même figure. Il suffit pour cela de montrer que, comme l'affirme l'énoncé, les lettres identiques

de la colonne de gauche (par exemple $S_4S_2S_3S_1$) sont classées dans le même ordre dans la colonne de droite. Dans la colonne de gauche, elles sont classées dans l'ordre des lignes dont elles sont le début, donc dans l'ordre des mots S_4IPIMISSIS, S_2ISSIPIMIS, S_3SIPIMISSI et S_1SISSIPIMI. Tous ces mots commençant par S, les lettres S_1, S_2, S_3 et S_4 sont donc classées dans l'ordre des mots IPIMISSIS, ISSIPIMIS, SIPIMISSI et SISSIPIMI (*ie* les mêmes mots sans le S initial).

Si l'on restaure le S, mais à la fin de ces derniers, on obtient des rotations du texte à encoder, dans notre cas IPIMISSISS$_4$, ISSIPIMISS$_2$, SIPIMISSIS$_3$ et SISSIPIMIS$_1$. Or ces rotations sont aussi classées dans l'ordre des mots IPIMISSIS, ISSIPIMIS, SIPIMISSI et SISSIPIMI (*ie* les même rotations mais sans le S final). CQFD.

11 L'énoncé a détaillé la méthode. On utilisera trois variables :

- le tableau `indices`, calculé par la question précédente ;
- `res`, que l'on renverra ;
- la position courante `p` initialisée à la valeur de la clé.

On répète alors $n' - 1$ fois l'opération suivante : ajouter dans `resultat` (dans la première case libre) la lettre située à la position courante dans t', puis « suivre les flèches » pour connaître la nouvelle position courante.

```
def decodageBW(t2):
    n=len(t2)-1; indices=trouverIndices(t2);
    res=[]; p=t2[n-1]
    for i in range(n):
        res.append(t2[p])
        p=indices[p]
    return res
```

La boucle `for` et l'appel à `trouverIndices` s'effectuent en $O(n')$. Par conséquent,

Le temps d'exécution de `decodageBW` est $O(n')$.

Voici quelques remarques de culture générale.

La seconde passe de l'algorithme présenté ici (la compression par redondance) est assez naïve. En pratique, le programme `bzip2` utilise un autre algorithme qui compresse mieux.

La méthode étudiée ici a un coût $O(n^2 \log n)$, ce qui rend très long la compression de fichiers de plusieurs Mo. En pratique, on divise le fichier en plusieurs blocs de taille fixe. On perd alors un peu en facteur de compression, mais on gagne en temps d'exécution. Beaucoup de programmes de compression ont un paramètre permettant d'ajuster ce compromis.

Il ne faut pas tester un algorithme de compression sur un texte aléatoire, qui n'est pas compressible. Au contraire, prendre par exemple une page Web au hasard ou un fichier de texte sur son ordinateur. En effet, la plupart de ces documents ne sont pas aléatoires et se prêtent bien à la compression.

Nous avons vu ici une compression « non destructive » ou « sans pertes », car toutes les transformations sont bijectives. Pour compresser des fichiers multimédia, on peut s'autoriser à dégrader ou supprimer des détails que l'on perçoit peu ou pas. Typiquement pour des images, du son ou de la vidéo, on peut choisir d'amoindrir la précision des couleurs ou de ne pas coder un

son faible masqué par un son fort. Ceci permet d'obtenir une taille après compression bien plus petite, mais réalise une compression destructive. C'est le cas des formats JPG (tandis que PNG est sans pertes), MP3 ou OGG, MPG et d'autres.