Corrigé : compression bzip (X mp 2007)

Partie I. Compression par redondance

Question 1.

```
def occurrences(t):
    r = [0] * 256
    for c in t:
        r[c] += 1
    return r
```

Question 2. Il s'agit de trouver l'indice de l'élément minimal dans le tableau des occurrences calculé à la question précédente.

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

Question 3. La fonction qui suit utilise trois invariants : s désigne la longueur du codage (sa valeur initiale est 1 puisque le codage commence par le marqueur), c la lettre courante et l le nombre de répétitions de la lettre c.

Question 4. La démarche est semblable, mais cette fois-ci s contient le codage au lieu de sa longueur.

```
def tailleCodage(t):
    s = 1
    c, l = t[0], 0
    for i in range(1, len(t)):
        if t[i] == c:
            l += 1
        else:
            if l == 0:
                s += 1
            else:
                s += 3
            c, l = t[i], 0
    if l == 0:
        s += 1
    else:
        s += 3
    return s
```

```
def codage(t):
    clef = mini(t)
    s = [clef]
    c, l = t[0], 0
    for i in range(1, len(t)):
        if t[i] == c:
            l += 1
        else:
            if l == 0:
                s.append(c)
            else:
                s.extend([clef, l, c])
            c, l = t[i], 0
    if l == 0:
        s.append(c)
    else:
        s.extend([clef, l, c])
    return s
```

Question 5.

```python
def decodage(tprime):
    clef = tprime[0]
    t = []
    i = 1
    while i < len(tprime):
        if tprime[i] == clef:
            for j in range(tprime[i+1]+1):
                t.append(tprime[i+2])
            i += 3
        else:
            t.append(tprime[i])
            i += 1
    return t
```

Partie II. Transformation de Burrows-Wheeler

Question 6.

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

Question 7. La dernière lettre de la rotation numérotée par l'entier r_i est la lettre d'indice $(r_i + n - 1) \bmod n$ dans le tableau t. D'où la fonction :

```python
def codageBW(t):
    n = len(t)
    rot = triRotations(t)
    tprime = []
    for i, r in enumerate(rot):
        tprime.append(t[(r - 1) % n])
        if r == 1:
            clef = i
    return clef, tprime
```

Question 8. La fonction `comparerRotations` possède une complexité en $O(n)$; sachant que la fonction `triRotations` fait appel $O(n \log n)$ fois à cette fonction, le calcul du tableau `rot` possède une complexité en $O(n^2 \log n)$. Le reste du calcul du tableau t' se réalise en temps linéaire, donc la complexité de la fonction `codageBW` est en $grand O(n^2 \log n)$.

Partie III. Transformation de Burrows-Wheeler inverse

Question 9. On procède à un tri par dénombrement :

```python
def triCarsDe(tprime):
    occ = occurrences(tprime)
    triCars = []
    for i in range(256):
        for _ in range(occ[i]):
            triCars.append(i)
    return triCars
```

Cette fonction possède une complexité en $O(256 + n)$, soit en temps linéaire par rapport à n si n est grand devant 256.

Question 10. Il est important de noter que dans le tableau `triCars`, les lettres semblables sont rangées dans des cases *consécutives*. La fonction qui suit utilise un invariant d qui indique l'indice du début de la recherche d'une lettre dans le tableau t' ; on aura $d = 0$ si c'est la première fois que l'on recherche cette lettre dans t' et $d = d' + 1$ lorsque la lettre aura déjà été cherchée et trouvée à l'emplacement d' lors de la dernière recherche.

```
def trouverIndices(tprime):
    triCars = triCarsDe(tprime)
    indices = []
    for i, c in enumerate(triCars):
        if i == 0 or triCars[i-1] != triCars[i]:
            d = 0
        else:
            d += 1
        while tprime[d] != c:
            d += 1
        indices.append(d)
    return indices
```

Le calcul du tableau `triCars` se réalise en $O(n)$, puis, pour chaque élément de ce tableau, la recherche de la correspondance dans le tableau t' prend un temps en $O(n)$. La complexité de la fonction `indices` est donc en $O(n^2)$.

Remarque. Une autre démarche est possible, de complexité linéaire. Au lieu d'utiliser la fonction `triCarsDe`, on utilise la fonction suivante, qui calcule un tableau répertoriant la position dans le tableau `triCars` du premier caractère de chaque type.

```
def reperes(tprime):
    occ = occurrences(tprime)
    r = [0] * 256
    for i in range(255):
        r[i+1] = r[i] + occ[i]
    return r
```

Il reste ensuite à parcourir le tableau `tprime` en incrémentant la case correspondante de ce tableau à chaque fois qu'un caractère est traité.

```
def trouverIndices(tprime):
    r = reperes(tprime)
    indices = [0] * len(tprime)
    for i, c in enumerate(tprime):
        indices[r[c]] = i
        r[c] += 1
    return indices
```

Question 11. `indices`$[c]$ désigne l'indice dans t' de la lettre qui suit la lettre $t'[c]$ dans le mot codé t. D'où la fonction :

```
def decodageBW(clef, tprime):
    indices = trouverIndices(tprime)
    c = clef
    t = [tprime[c]]
    for _ in range(len(tprime)-1):
        c = indices[c]
        t.append(tprime[c])
    return t
```

Cette fonction est de complexité quadratique si on utilise la première version de `trouverIndices` ou linéaire avec la seconde version.

Question 12. Le tableau t' est formé des dernières lettres des rotations de t triées par ordre lexicographique ; par construction le tableau `tricars` est constitué des premières lettres des rotations de t ordonnées de la même façon. Ainsi, la correspondance qui va du tableau t' vers le tableau `triCars` associe à chaque dernière lettre d'une rotation de t sa première lettre, *qui n'est autre que la lettre qui la suit dans le mot* t. Il suffit donc de partir de la clef, qui indique la première lettre du mot t, et de suivre cette correspondance pour reconstituer le mot initial.