Devoir Surveillé #2 MP2I ITC (avec solutions)

Mercredi 2026-06-02 ; durée : deux heures

Ce sujet est constitué de deux exercices indépendants, qui peuvent être traités dans n'importe quel ordre.

Remarques préliminaire

Dans toute question, il est possible d'admettre un résultat d'une question précédente du même exercice. Notamment, il est possible d'utiliser une fonction dont l'écriture a été demandée, même si la question correspondante n'a pas été traitée.
Quand les spécifications d'une fonction font une supposition sur un argument (par exemple qu'il est toujours égal à une `list` non vide), il n'est pas nécessaire dans une implémentation de vérifier que cela est bien le cas.

Exercice 1. *Doudou le hamster*

Doudou est un hamster domestique (*mesocricetus auratus*) dont la vie obéit à des règles très simples, fort bien décrites sur la page Wikipédia à son sujet. On en recopie ici l'essentiel (utile pour cet exercice) :

Doudou le hamster ne connaît que trois endroits dans sa cage : les copeaux où il dort, la mangeoire où il mange et la roue où il fait de l'exercice. Ses journées sont assez semblables les unes aux autres. Toutes les minutes, il peut soit changer d'activité, soit continuer celle qu'il était en train de faire.

- *Quand il dort, il a 9 chances sur 10 de ne pas se réveiller la minute suivante.*
- *Quand il se réveille, il y a 1 chance sur 2 qu'il aille manger et 1 chance sur 2 qu'il parte faire de l'exercice.*
- *Le repas ne dure qu'une minute, après il fait autre chose.*
- *Après avoir mangé, il y a 3 chances sur 10 qu'il parte courir dans sa roue, mais surtout 7 chances sur 10 qu'il retourne dormir.*
- *Courir est fatigant pour Doudou ; il y a 8 chances sur 10 qu'il retourne dormir au bout d'une minute. Sinon il continue en oubliant qu'il est déjà un peu fatigué.*

On peut représenter le processus suivi par Doudou par un graphe pondéré $\mathcal{G}$, représenté graphiquement à la Figure 1.

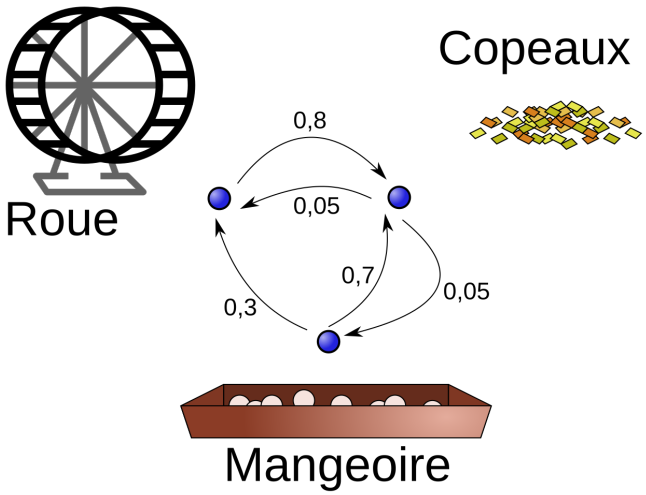

FIGURE 1 – Représentation graphique du graphe $\mathcal{G}$ Crédit : *gravgun* — CC-BY-SA.

Si l'on associe les entiers 0, 1, 2 aux sommets *roue*, *copeaux*, *mangeoire* de ce graphe, on peut alors le représenter en Python comme la `list` de `list` `G1` ci-dessous :

```
G1 = [
        [(1,0.8)],              # 0 : roue
        [(0, 0.05), (2, 0.05)], # 1 : copeaux
        [(0, 0.3), (1, 0.7)]    # 2 : mangeoire
    ]
```

où la présence d'un arc (i, j) de poids w est donnée par l'existence d'une paire `(j, w)` dans `G1[i]`

1. Le graphe `G1` est il un graphe orienté ou non-orienté ?

   `G1` est orienté : il possède par exemple un arc $2 \to 0$ mais pas d'arc $0 \to 2$.

2. Donnez une représentation Python par matrice d'adjacence du même graphe pondéré $\mathcal{G}$ (pour la même association entre sommets et entiers que précédemment). Explicitez votre choix pour représenter l'absence d'arc. (Il n'est cependant pas demandé de justifier la pertinence de celui-ci dans la modélisation de la vie de Doudou.)

   On choisit de représenter une absence d'arc par une valeur `False`, et un arc de poids w par son poids. On obtient alors :

   ```
   G2 = [ [False, 0.8, False],
          [0.05, False, 0.05],
          [0.3, 0.7, False  ] ]
   ```

3. Donnez un chemin dans $\mathcal{G}$ qui contient au moins deux sommets distincts et qui n'est pas un cycle.

   Par exemple *roue* $\to$ *copeaux*.

4. Donnez un chemin dans G qui est un cycle.

   Par exemple *roue* $\to$ *copeaux* $\to$ *roue* (le graphe étant orienté), ou *roue* $\to$ *copeaux* $\to$ *mangeoire* $\to$ *roue*.

Pour son anniversaire Doudou a le droit de choisir une nouvelle cage plus grande (contenant par exemple une baignoire, *plusieurs* endroits où dormir, etc. Autant de nouvelles zones qui seront modélisées par de nouveaux sommets dans un graphe). Doudou est cependant inquiet qu'une nouvelle cage soit un prétexte pour le forcer à faire de l'exercice contre son gré : il refusera catégoriquement de s'installer dans une cage qui ne lui permette pas de se reposer immédiatement après toute activité (par exemple une cage ayant une unique zone de repos à laquelle on ne pourrait accéder depuis une unique mangeoire qu'en montant un plan incliné).

Dans toute la suite de l'exercice, on s'intéressera à des *plans de cage* modélisés par des **graphes non-orientés et non-pondérés**, dont les sommets sont des entiers naturels consécutifs commençant à 0. Ils seront représentes en Python par `list` de `list` d'adjacence : soit $\mathcal{P}$ un tel graphe de n sommets $\mathcal{S} = \{0, \ldots, n-1\}$ et d'arêtes $\mathcal{A}$, une représentation Python `P` sera telle que la présence d'un arc $(i, j) \in \mathcal{A}$ est équivalente à la présence de la valeur `j` dans la `list` `P[i]` et à celle de la valeur `i` dans `P[j]`.

5. On va représenter l'ensemble des sommets de repos par une `list`. Écrivez une fonction Python `est_repos(s, R)` qui renvoie `True` si l'élément `s` appartient à la `list` `R` (représentant ces sommets de repos) et `False` sinon.

   On propose :

   ```
   def est_repos(s, R):
       for t in R: # O(len(R)) itérations
           if t == s: # O(1)
               return True
       return False
   ```

6. Donnez le coût temporel pire-cas asymptotique (sous forme d'une notation O) de votre fonction `est_repos` en fonction de la longueur (notée R) de `R`.

   Le coût est un $O(R)$ (*cf.* les commentaires accompagnant le code).

7. Aidez Doudou en écrivant une fonction Python `Doudou_compatible1(P, R)` qui prend en entrée :

   — une représentation (comme ci-dessus) `P` d'un graphe ;

— une `list` R de tous les sommets de P qui sont des sommets de repos ;

et renvoie `True` si tout sommet de P est ou bien un sommet de repos, ou bien adjacent (voisin) d'un sommet de repos.

On propose une solution qui passe par une seconde fonction intermédiaire qui teste si l'intersection de deux `list` est non vide, mais ce n'est pas nécessaire :

```
def contient_repos(A, R): # coût O(len(A) x len(R))
    for a in A: # O(len(A)) itérations
        if est_repos(a, R): # O(len(R))
            return True
    return False

def Doudou_compatible1(P, R):
    for s in range(len(P)): # O(len(P) itérations
        if not est_repos(s, R): # O(len(R)
            if not contient_repos(P[s], R): # O(len(P[s]) x len(R))
                return False
            # else: P[s] contient un sommet de repos
        # else: s est un sommet de repos
    return True
```

8. Donnez le coût (temporel pire-cas asymptotique sous forme d'une notation O) de votre fonction `Doudou_compatible1` en fonction du nombre de sommets S et du nombre d'arêtes A de P, ainsi que du nombre R de sommets de repos. On cherchera à obtenir la majoration *la plus fine possible*.

Sur la base des commentaires accompagnant le code ci-dessus, une majoration brutale de `len(P[s])` par le nombre de sommets S du graphe donne un $O(S \times (R + S \times R))$ ce qui par le fait que $R \leq S$ se simplifie en $O(S^2 \times R)$. On peut être plus fin en majorant globalement le coût de tous les appels à `contient_repos` par $\sum_{s \in S} d(s) \times R = A \times R$, avec $d(s)$ le degré de s, auquel il ne reste plus qu'à ajouter les S appels à `est_repos`. Ceci donne :

$$O(R \times (S + A))$$

9. Expliquez comment améliorer le coût de votre fonction en utilisant un dictionnaire pour représenter l'ensemble des sommets de repos. Écrivez la fonction correspondante `Doudou_compatible1bis` et donnez son coût (temporel pire-cas asymptotique sous forme d'une notation O).

On peut avantageusement remplacer la représentation des sommets de repos par un dictionnaire dont les clefs sont exactement ces derniers (associés à des valeurs quelconques, par exemple `True`). Ceci permet alors de tester si un sommet est de repos en temps constant (dans le modèle de coût approché qui est le nôtre).
On peut par exemple implémenter ceci comme :

```
def contient_reposbis(A, R): # coût O(len(A))
    for a in A: # O(len(A)) itérations
        if a in R: # O(1)
            return True
    return False

def Doudou_compatible1bis(P, R):
    for s in range(len(P)): # O(len(P)) itérations
        if not s in R: # O(1)
            if not contient_reposbis(P[s], R): # O(len(P[s]))
                return False
    return True
```

ce qui par une analyse similaire à la question précédente est de coût $O(S + A)$.

Doudou a reçu un prospectus vantant les mérites d'une cage possédant un nombre impressionnant de zones (pas moins de 37 emplacements de sieste, 12 baignoires, 8 roues, 14 toboggans…). De nature méfiante, Doudou aimerait cependant s'assurer du fait qu'il sera capable d'atteindre n'importe quelle zone de la cage par ses propres moyens (il ne voudrait surtout pas d'une « cage » en plusieurs parties dont certaines seraient partagées avec d'autres hamsters à tour de rôle, *via* un système complexe de réservation et de transferts) !

Autrement dit, soit $\mathcal{P} = \mathcal{S}, \mathcal{A}$ le graphe (non-orienté, non-pondéré) modélisant le plan de cette cage, Doudou voudrait vérifier qu'il est bien possible d'atteindre n'importe quel sommet du graphe depuis n'importe quel autre sommet (éventuellement en passant par des sommets intermédiaires).

10. Comment appelle-t'on les graphes non-orientés possédant la propriété recherchée par Doudou ?

Ce sont des graphes connexes. (Telle que formulée, la propriété évoque plutôt la connexité forte, mais celle-ci coïncide avec la connexité pour les graphes non-orientés.)

11. Écrivez une fonction Python `Doudou_compatible2(P)` qui pour `P` une représentation de graphe par `list` de `list` d'adjacence renvoie `True` s'il satisfait la propriété ci-dessus et `False` sinon. Cette fonction devra être de coût (temporel pire-cas asymptotique) linéaire en la taille de la représentation `P`, mais l'on ne demande pas de démontrer ce coût. On pourra supposer que `P` est non vide (possède au moins un sommet).

Il suffit d'effectuer un parcours de graphe depuis un sommet quelconque et de vérifier que tous les sommets ont été visités à l'issue de celui-ci.
On propose ci-dessous un parcours en profondeur implémenté itérativement, mais bien d'autres choix sont possibles.

```
def Doudou_compatible2(P):
    vis = [False for _ in range(len(P))]
    a_visiter = [0] # len(P) > 0
    while (len(a_visiter) > 0):
        s = a_visiter.pop()
        if not vis[s]: # on n'a pas encore visité s
            vis[s] = True
            for t in P[s]:
                a_visiter.append(t)
    # vis[i] vaut True ssi. i est accessible depuis 0,
    # c.à.d. ss.'il est dans la même composante connexe que 0
    for s in range(len(P)):
        if not vis[s]:
            return False
    return True
```

Exercice 2. *BWT*

Exercice adapté de l'épreuve de composition d'informatique MP option SI & PC du concours d'admission 2007 de l'École Polytechnique & de l'École supérieure de physique et de chimie industrielles.

La *transformation de Burrows-Wheeler* (ou BWT) est une transformation inversible qui à partir d'un texte donné (ou plus généralement, une suite d'octets) produit un autre texte contenant exactement les mêmes lettres (le même nombre de fois) mais dans un ordre où les répétitions d'une même lettre ont tendance à être contiguës. Cette propriété de la transformation de Burrows-Wheeler en fait un élément important de l'algorithme de compression *bzip2*.
Le but de cet exercice est d'implémenter cette transformation pour des suite d'octets, **que l'on représentera en Python par des `list` (supposées non vides) dont tous les éléments sont des entiers entre `0` et `255` (bornes incluses)**. Cependant, pour plus de lisibilité on décrit le fonctionnement de la BWT sur le texte « `doudou` » représenté comme un mot sur l'alphabet latin. La transformation procède en trois étapes :

Calcul des rotations du texte

Pour un texte de longueur n, on calcule les n rotations circulaires obtenues en décalant cycliquement toutes les lettres de une position (à gauche, dans notre exemple) de façon répétée. Pour `doudou` on obtient :

```
doudou
oudoud
udoudo
doudou
oudoud
udoudo
```

Tri

On trie les rotations par ordre lexicographique (l'ordre du dictionnaire). On obtient :

```
doudou
doudou
oudoud
oudoud
udoudo
udoudo
```

Extraction de la dernière colonne

Le résultat de la transformation est donné par les dernières lettres des rotations triées (dans l'ordre du tri), ainsi que l'indice (en partant de zéro) de la lettre dans ce texte résultant qui est la première lettre du texte original. Dans notre exemple le texte est `uuddoo`, et en supposant que le tri est *stable* cet indice est égal au rang de la première rotation valant `oudoud`, soit 2. Le résultat de BWT appliqué à `doudou` est donc la paire `(uuddoo, 2)`

En pratique (pour des raisons d'efficacité), on ne va pas calculer est stocker l'ensemble des rotations du mot d'entrée. On se contente de noter $\rho_i(t)$ la i-ème rotation à gauche d'un texte t. Ainsi pour t valant `doudou`, ρ_0 représente le texte initial `doudou`, ρ_1 représente `oudoud`, etc.

1. Écrivez une fonction Python `compare_rotations(t, i, j)` qui prend en entrée un texte `t` (représenté par une `list`) et deux indices `i`, `j` entre `0` et `len(t) - 1` et qui renvoie :

   — `1` si $\rho_i(t)$ est plus grand que $\rho_j(t)$ dans l'ordre lexicographique ;
   — `-1` si $\rho_i(t)$ est plus petit que $\rho_j(t)$ dans l'ordre lexicographique ;
   — `0` sinon.

   On demande une fonction de coût temporel linéaire en la longueur de `t`, mais il n'est pas nécessaire de montrer celui-ci.

   La d-ème lettre dans une rotation à gauche de i positions est la $i+d$-ème lettre du texte initial, modulo sa longueur. On propose :

```python
def compare_rotations(t, i, j):
    n = len(t)
    for d in range(n):
        tid = t[(i+d)%n]
        tjd = t[(j+d)%n]
        if tid < tjd:
            return -1
        if tid > tjd:
            return 1
    return 0
```

On suppose disposer d'une fonction `tri_rotations(t)` qui trie les rotations du texte `t` en utilisant la fonction `compare_rotation`. Elle renvoie une `list` d'entiers `r` représentant les numéros des rotations dans l'ordre lexicographique. (Ainsi, si l'on note $\leq$ cet ordre, `r` est tel que $\rho(\texttt{r}[0]) \leq \rho(\texttt{r}[1]) \leq \cdots$.)

Un exemple d'une telle fonction (de coût quadratique en la longueur de son argument) est :

```python
def tri_rotations(t):
    n = len(t)
    r = [i for i in range(n)]
    for i in range(n - 1):
        mi = i
        for j in range(i + 1, n):
            if compare_rotations(t, r[j], r[mi]) < 0:
                mi = j
        r[i], r[mi] = r[mi], r[i]
    return r
```

2. Écrivez une fonction Python `BWT(t)` qui prend en entrée un texte `t` (représenté par une `list`) et renvoie un `tuple (tb, c)` où `tb` est une nouvelle `list` d'entiers représentant le texte après transformation (`uuddoo` dans notre exemple) et `c` la clef associée (`2` dans notre exemple).

On construit `tb` comme la `list` des dernières lettres de `t` dans l'ordre des rotations données par `r`. La clef `c` est donnée par l'indice de `1` dans `r`, ce que l'on calcule *via* une fonction `findi` introduite pour l'occasion :

```
def findi(e, x):
    for i in range(len(x)):
        if x[i] == e:
            return i
    return -1

def BWT(t):
    n = len(t)
    r = tri_rotations(t)
    c = findi(1, r)
    tb = [t[(i+n-1)%n] for i in r]
    return tb, c
```

Pour implémenter la transformation inverse (décrite plus bas ci-dessous), nous allons d'abord devoir trier le tableau `tb`. On propose de faire cela en utilisant un algorithme de tri par dénombrement.

3. Écrivez une fonction Python `occurrences(t)` qui pour une `list t` dont tous les éléments sont garantis être des entiers entre `0` et `255` (bornes incluses) renvoie une `list` de longueur 256 dont le i-ème élément (en partant de zéro) contient le nombre d'occurrence de l'entier i dans `t`.

On propose :

```
def occurrences(t):
    occ = [0 for _ in range(256)]
    for e in t:
        occ[e] = occ[e] + 1
    return occ
```

4. Écrivez une fonction Python `tri_denom(t)` qui pour une `list t` comme à la question précédente renvoie une **nouvelle** `list` de même longueur qui contient les mêmes éléments que `t` avec même nombre d'occurrences, et qui est triée.
On demande un coût temporel linéaire en la longueur de `t`, mais on ne demande pas de le montrer. (Rappel : 256 est une constante.)

On propose :

```
def tri_denom(t):
    occ = occurrences(t)
    ts = []
    for i in range(256):
        for j in range(occ[i]):
            ts.append(i)
    return ts
```

Malgré sa double boucle cette fonction est bien de coût linéaire en la longueur de `t` : l'appel à `occurrences` est de coût linéaire en cette longueur, et le nombre total d'itérations du corps (de coût constant) de la seconde boucle est égal à la longueur de `t`.

Transformation inverse

Pour inverser la BWT en `(tb, c)`, on superpose (virtuellement) le texte transformé `tb` et sa version triée `tbs`. Pour notre exemple on obtient :

```
uuddoo
ddoouu
```

On considère alors (conceptuellement) le graphe orienté dont les sommets sont les lettres de `tb` et `tbs`, et les seuls arcs sont ceux tels que pour tout `i` entre `0` et `len(tb)-1` :

— il existe un arc entre `tb[i]` et `tbs[i]`
— soit `v` la valeur de `tbs[i]` et k son rang dans `tbs` parmi les lettres de `tbs` égales à `v`, alors il existe un arc entre `tbs[i]` et `tb[j]` où `tb[j]` est la k-ième lettre de `tb` égale à `v`.

On illustre ce graphe pour notre exemple dans la Figure 2.

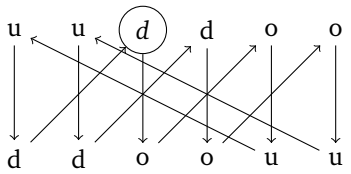

FIGURE 2 – Graphe orienté utilisé pour le calcul de la BWT inverse.

Pour calculer la transformation inverse, on construit le cycle de longueur `len(tb)` débutant en `tb[c]` (où `c` est la clef produite à l'issue de la BWT), et l'on renvoie le texte constitué par la suite des sommets dans l'ordre dans lequel ils sont rencontrés dans le cycle, en supprimant les doublons structurellement présents.

On peut remarquer que pour représenter le graphe ci-dessus, il est relativement inutile de chercher à représenter les arcs entre sommets de `tb` et `tbs` ; on va donc se contenter de stocker les « listes d'adjacences » (qui n'auront jamais qu'un seul élément…) des sommets de `tbs`.

5. Écrivez une fonction Python `adjs(tb)` qui pour `(tb, c)` résultat d'un appel à `BWT` renvoie une `list` `tbsa` de même longueur que `tb` telle que `tbsa[i]` contient l'indice `j` de l'unique sommet tel qu'il existe un arc entre `tbs[i]` et `tb[j]` dans le graphe ci-dessus.

On propose la fonction suivante, de coût quadratique en la longueur de son argument. Elle utilise une variante de la fonction `findi` précédente qui prend un argument supplémentaire indiquant le début de la zone de recherche, et une `list` `start` qui mémorise le rang de la dernière occurrence « utilisée » de chaque lettre dans `tb`.

```
def find_nexti(e, i, x):
    for j in range((i+1),len(x)):
        if x[j] == e:
            return j
    return -1

def adjs(tb):
    n = len(tb)
    tbs = tri_denom(tb)
    start = [-1 for _ in range(256)]
    tbsa = [-1 for _ in range(n)]

    for i in range(len(tb)):
        v = tbs[i]
        j = find_nexti(v, start[v], tb)
        tbsa[i] = j
        start[v] = j

    return tbsa
```

Pour notre exemple, `adjs` devrait renvoyer `[2, 3, 4, 5, 0, 1]`.

6. Écrivez une fonction Python `IBWT(tb, c)` qui pour `(tb, c)` résultat d'un appel à `BWT(t)` renvoie le texte initial `t`.

On n'a pas besoin de recalculer `tbs` : il suffit de déduire les lettres de `t` à chaque « rebond » vers `tb`. On peut par ailleurs exploiter le fait que l'on connaît trivialement la longueur du cycle à suivre, puisqu'elle est égale à celle du texte à reconstruire.
On propose alors :

```
def IBWT(tb, c):
    n = len(tb)
```

```
    tbsa = adjs(tb)
    t = []
    i = c
    for _ in range(n):
        t.append(tb[i])
        i = tbsa[i]
    return t
```