Corrige — Codage de Huffman

SUP

I Algorithmes de cours

Question 1. *Exponentiation rapide*

coeff
2

c.f. cours

Question 2. *Maximum*

coeff
2

c.f. cours

II Introduction

III Codage des chaînes de caractères

Question 3. *Entiers représentables sur 8 bits*

coeff
2

$[-128, 127]$

Question 4. *Chaîne vers tableaux*

coeff
2

```
array([110, 97, 100, 105, 110, 101])
```

Question 5. *Code binaire d'un entier*

coeff	code	complexité
2	6	3

```python
def code_vers_int(code : list[int]) -> int :
    val = 0
    for i in range(8) :
        val = val * 2
        val = val + code[i]
    if val > 127 :
        val = val - 256
    return val
```

Complexité : O(1). Une boucle de taille bornée et que des appels elémentaires.

Question 6. *Code binaire d'un tableau*

coeff	erreur	extraction codes	création tableau	complexité
3	2	3	2	2

```python
def code_vers_array(code : list[int]) -> array[int8] :
    assert len(code) % 8 == 0
    val_list = []
    for i in range(0, len(code), 8) :
        code_x = code[i:i+8]
        x = code_vers_int(code_x)
        val_list.append(x)
    return creer_array(val_list)
```

Complexité : O(`len(code)`). Pour une itération fixée :

- slice de 8 éléments : O(1)
- appel de code_vers_int : O(1)
- `append` : O(1)

Donc la boucle a une complexité en O(`len(code)`). La complexité de `creer_array` est en O(`len(val_list)`) donc en O(`len(code)`) aussi ce qui amène une complexité de O(`len(code)`).

IV Compression de chaînes de caractère

Question 7. *Décompression*

coeff
2

```
'nad'
```

Question 8. *Ambigüité*

coeff
2

Deux chaînes possèdent la compression donnée dans l'énoncé : `'na'` et `'di'`. Impossible de savoir laquelle des deux a généré cette compression, donc impossible de décompresser.

Question 9. *Codes Préfixe*

coeff	vérification préfixe	vérification pour toute paire
4	4	5

```
def prefixe(s1 : list[int], s2 : list[int]) -> bool :
    if len(s1) > len(s2) :
        return False
    for i in range(len(s1)) :
        if s1[i] != s2[i] :
            return False
    return True

def sans_prefixe(d : dict) -> bool :
    for c1,v1 in d.items() :
        sans_prefixe = True
        for c2,v2 in d.items() :
            if c1 != c2 and prefixe(v1,v2) :
                return False
    return True
```

Question 10. *Compression*

coeff	concatenation code	completion 0	conversion array
2	4	3	2

```
def compression(s : str, d : dict) -> array[int8] :
    code = []
    for c in s :
        code += d[c]
    for i in range(len(code) % 8) :
        code.append(0)
    return code_vers_array(code)
```

V Arbres

Question 11. *Accessibilité*

coeff	parcours en largeur	renvoie du booléen
3	8	2

Une file avec liste aurait été acceptée.

```python
from collections import deque

def accessible(g : dict, s ) -> bool :
    marquage = {}
    fifo = deque()
    marquage[s] = True
    fifo.append(s)
    while len(fifo) > 0 :
        u = fifo.popleft()
        for v in g[u] :
            if v in g :
                marquage[v] = True
                fifo.append(v)
    return len(marquage) == len(g)
```

Question 12. *Degrés*

coeff	degré entrants	degrés sortant	dictionnaire
2	3	3	3

```python
def degres(g : dict) -> dict :
    d = {}
    for s in g.keys() :
        d[s] = 0
    for s,v in g.items() :
        for x in v :
            d[x] += 1
    for s in g.keys()
        d[s] = (d[s], len(g[s]))
    return d
```

Question 13. *Vérification*

coeff	degrés	accessibilité	feuilles	complexite
3	4	1	1	3

```python
def est_arbre_huffman(g : dict) -> bool :
    d = degres(g)
    if d[0][0] != 0 : # (2)
        return False
    if not accessible(g, 0) : # (3)
        return False
    for s in d.keys() :
        if s != 0 and d[s][0] != 1 : # (4)
            return False
```

```
        if d[s][1] >= 2 : # (5)
            return False
        if est_ascii(s) : # (6)
            if d[s][0] != 0 :
                return False
        else :
            if d[s][0] == 0 :
                return False
    return True
```

Complexité en $O(n+m)$ qui est celle de la fonction `degres`.

Question 14. *Unicité du chemin*

coeff
3

Initialisation : 0 est le seul sommet au niveau 0 et est accessible depuis lui-même. Il n'est pas accessible depuis un autre sommet car son degré entrant est nul.

Hérédité : Supposons la propriété vrai au niveau n et soit s au niveau $n+1$. On sait qu'il existe un chemin $c = 0 \to \cdots \to t \to s$ de longueur $n+1$. Soit $c' = 0 \to \cdots \to t' \to s$ un chemin menant à s. Par définition, s ne possède qu'un seul prédécesseur donc $t' = t$. Par ailleurs, t est au niveau n car c est de longueur minimale et s est de niveau $n+1$. Par hypothèse de récurrence, c est le seul chemin de source 0 et de destination t. De ce fait $c' = c$. Il y a bien unicité de chemin menant à s.

VI Codage de Huffman

Question 15. *Code d'un arbre de Huffman*

coeff	parcours	calcul dictionnaire
3	5	4

Le parcours en profondeur n'a pas besoin de marquer les sommets puisque chaque sommet est accessible par un unique chemin. Donc impossible d'arriver à un sommet par deux endroits différents.

Le parcours en profondeur rempli le dictionnaire `codes` par effet de bord.

```
def code_arbre(a : dict) -> dict :
    return parcours_profondeur(a, 0, [], {})

def parcours_profondeur(a, s, chemin_vers_s, codes) :
    if a[s] == [] :
        codes[s] = chemin_vers_s
        return codes
    for i in range(len(code[s])) :
        chemin_vers_i = chemin_vers_s.copy()
        chemin_vers_i.append(i)
```

```
        parcours_profondeur(a, i, chemin_vers_i, codes)
    return codes
```

Question 16. *Nombre d'occurrences*

coeff
2

```
def occurences(s : str) -> dict :
    d = {}
    for c in s :
        if c in d :
            d[c] += 1
        else :
            d[c] = 1
    return d
```

Question 17. *Paires minimale de clés*

coeff
2

```
def paire_min(d : dict) -> dict :
    k1, v1 = None, float("inf")
    k2, v2 = None, float("inf")
    for k,v in d.items() :
        if v < v1 :
            k2, v2 = k1, v1
            k1, v1 = k, v
        elif v < v2 :
            k2, v2 = k, v
    return k1, k2
```

Question 18. *Construction d'un arbre de Huffman optimal*

coeff
2

```
def huffman(s : str) -> dict :
    d = occurences(s) # (1)
    # (2)
    g = {}
    for k in d.keys() :
        g[k] = []
```

```
    # (3)
    i = 1
    while len(d) > 1 : # (10)
        k1, k2 = paire_min(d) # (4)
        v1, v2 = d.pop(k1), d.pop(k2) # (5)
        if len(d) == 0 :
            g[0] = [k1,k2] # (6)
        else :
            g[i] = [k1,k2] # (7)
            d[i] = v1 + v2 # (8)
            i = i + 1 # (9)
    return g
```

Question 19. *Terminaison*

coeff	variant	preuve
2	3	6

Le variant est V = `len(d)`.

Initialisation. A l'entrée de la boucle. V est un entier strictement positif car la chaîne contient au moins deux caractères différents.

Hérédité. Supposons $V \in \mathbb{N}$ à l'entrée d'une itération de boucle. La condition d'entrée signifie que $V \geq 2$. Le corps de boucle supprime deux éléments distinct de `d`, car `paire_min` renvoie deux clés différentes. Après cette opération on a $V' = V - 2$. Puis les opérations suivantes augmentent V de 1 ou le laisse inchangé. Donc à la fin de la boucle, $V' - 1 \leq V'' \leq V'$ et $V - 2 \leq V'' \leq V - 1$. D'où $V'' \in \mathbb{N}$ et $V'' < V$.

V est bien un variant : à chaque itération V est un entier naturel et V décroit d'itération en itération. Donc la boucle termine.

Question 20. *Codage de Huffman*

coeff
1

```
def codage_huffman(s : str) -> array[int8] :
    a = huffman(s)
    code = code_arbre(a)
    return compression(s,code)
```

Question 21. *Décodage de Huffman*

coeff	suivi du chemin	construction de la chaîne
4	6	3

```python
def decodage_huffman(comp : array[int8], h : dict) -> str:
    decomp = ""
    # parcours de chemin
    s = 0 # sommet courant de l'arbre
    for i in range(len(comp)) :
        bit = comp[i]
        s = h[s][bit] # on suit le chemin gauche (0) ou droite (1)
        if len(h[s]) == 0 : # s est une feuille
            decomp += s # s est donc un caractère.
            s = 0 # on reprend au début.
    return
```

Question 22. *Code Préfixe*

coeff
2

Prenons deux codes, `l1` et `l2` de deux caractères différent `x1` et `x2`. Considérant sans perte de généralité que `l1` est de longueur n_1 plus courte ou égale à `l2`.

`l1` correspond exactement à un chemin `c1` dans l'arbre de huffman allant de 0 à `x1`. `l2[:`n_1`]` correspond exactement à un chemin `c2` dont la destination sera noté `s`. Montrons que `s` est différent de `x1`, alors par unicité des chemin dans l'arbre de huffman, on aura montré que `l1` est différent de `l2[:`n_1`]`, donc que `l1` n'est pas préfixe de `l2`, puis `x1` et `x2` étant quelconque, que le code est préfixe.

Si `s = x2` alors par hypothèse, `s` $\neq$ `x1`. Enfin si `s` $\neq$ `x2` alors `s` est un sommet intermédiaire du chemin menant à `x2`. Donc `s` a un successeur, et donc `s` n'est pas une feuille, donc pas un caractère `ASCII`. D'où `s` $\neq$ `x1`. CQFD.