Codage de Huffman

SUP

I Algorithmes de cours

Donner la version **récursive** des deux algorithmes ci-dessous.

Question 1. *Exponentiation rapide*

Entrées : a un entier et b un entier positifs.
Sortie : a^b

Question 2. *Maximum*

Entrées : une liste d'entiers relatifs.
Sortie : le maximum de la liste

II Introduction

Les chaînes de caractères ASCII sont stockées de manière contiguë dans la mémoire vive de l'ordinateur, à la manière des *tableaux* `numpy`. En conséquence, une chaîne de 12 caractères occupe exactement 13 octets. Les 12 premières cases sauvegardent les 12 premiers caractères dans l'ordre et la treizième contient un caractère spécial `'\0'` marquant la fin de la chaîne. Chaque caractère ASCII est représenté par son code ASCII, compris entre 0 et 127 (*c.f* table en annexe), `'\0'` a en particulier `0` comme code ASCII.

On s'intéresse dans ce sujet à un algorithme de *compression sans perte* d'une chaîne de caractère. Cet algorithme, le *codage de Huffman*, permet de représenter une chaîne de caractère en occupant moins d'espace que le codage ASCII natif.

On considérera que l'on dispose dans ce sujet du type `array` et `int8` du module `numpy`. Un *tableau* désignera le type `array` et un *entier 8 bits* le type `int8`. Contrairement à un entier `Python`, un entier de type `int8` occupe exactement un octet, et contrairement à une liste `Python` un `array` de `n` entiers 8 bits occupe exactement `n` octets.

De ce fait une chaîne de caractère correspond exactement à un tableau d'entiers 8 bits qui contient le code ASCII de ses caractères. On dira que qu'un tableau d'entier 8 bits *représente* une chaîne s'il correspond aux codes ASCII des caractères de cette dernière *privée du caractère nul final*, `'\0'`. Par exemple, la chaîne de caractère `"Linux␣Is␣Not␣UniX"` est représentée par :

```
array([76, 105, 110, 117, 120, 32, 73, 115, 32, 78, 111, 116, 32, 85, 110, 105, 88])
```

On disposera des fonctions suivantes :

```
from numpy import int8, array

def creer_array(l : list[int]) -> array[int8] :
    """
    Entrée : une liste d'entiers Python représentables sur 8 bits.
    Sortie : un tableau d'entiers 8 bits avec ces mêmes valeurs.
    Complexité : O(len(l))
    """

def str_vers_array(s : str) -> array[int8] :
    """
    Entrée : une chaîne de caractère.
    Sortie : un tableau d'entiers 8 bits représentant s.
    Complexité : O(len(s))
    """

def array_vers_str(a : array[int8]) -> str :
    """
    Entrée : un tableau d'entiers 8 bits.
    Sortie : une chaîne de caractère représentée par a.
    Complexité : O(len(a))
    """
```

Enfin, on représente un *code binaire* en `Python` par une liste de 0 et de 1 (type entier `Python`). On dit que `l` est le code binaire d'un entier 8 bit, noté `x`, si `l` contient 8 éléments qui forment l'écriture binaire de `x`. L'écriture est faîte de gauche à droite, *i.e* `l[0]` est le bit de poids fort. Par exemple, le code binaire de `144` est la liste :

```
[1, 0, 0, 1, 0, 0, 0, 0]
```

Le code binaire d'un tableau d'entier 8 bits est la *concaténation* des codes binaires de chaque élément du tableau. Le code binaire d'une chaîne de caractère *suivant le code ASCII* est le code binaire du tableau qui le représente. Par exemple, le code binaire de **`bin`** est :

```
[0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0 ]
```

III Codage des chaînes de caractères

Question 3. *Entiers représentables sur 8 bits*

Donner l'ensemble des entiers `Python` représentables en complément à 2 sur 8 bits.

Question 4. *Chaîne vers tableaux*

Quel est le tableau renvoyé par `str_vers_array("nadine")` ?

Question 5. *Code binaire d'un entier*

Écrire une fonction `code_vers_int(code : list[int]) -> int` qui prend le code binaire d'un entier représentable en complément à 2 sur 8 bits et qui renvoie l'entier `Python` correspondant.

Donner, en la justifiant, la complexité de votre fonction.

Question 6. *Code binaire d'un tableau*

Écrire une fonction `code_vers_array(code : list[int]) -> array[int8]` qui prend le code binaire d'un tableau d'entiers 8 bits et qui renvoie le tableau correspondant.

Votre fonction devra renvoyer une erreur si la longueur de la liste n'est pas divisible par 8.

Donner, en la justifiant, la complexité de votre fonction.

IV Compression de chaînes de caractère

La compression de données consiste à changer la représentation de la donnée de manière à ce qu'elle prenne moins de place. On dit qu'elle est sans perte si les données compressées peuvent être décompressée à l'identique.

On peut obtenir une compression simplement en changeant la représentation binaire des caractères. Dans le codage ASCII, chaque caractère est codé sur 8 bits, quelle que soit sa fréquence d'apparition dans un texte. En choisissant de coder sur les caractères ayant la plus grande fréquence d'apparition sur moins de bits, on peut gagner en espace.

Un *code* est un dictionnaire dont les clés sont les caractères de la table ASCII et les valeurs sont des codes binaires (tels que définis en introduction).

Le code binaire d'une chaîne de caractère *suivant* le code `d` est la *concaténation* des codes binaires de chaque caractère de la chaîne tels que donnés dans le dictionnaire `d`. Supposons par exemple donné un code `d` tel que `d['b'] = [0,0]`, `d['i'] = [1,0,1,1]` et `d['n'] = [1, 1,0,1]`. Le code binaire de `"bin"` suivant `d` est :

```
[0, 0, 1, 0, 1, 1, 1, 1, 0, 1]
```

La *compression suivant* le code `d` d'une chaîne de caractère `s` est le tableau d'entiers 8 bits dont le code binaire est le code obtenu en complétant le code binaire de `s` suivant `d` par des 0 jusqu'à avoir une longueur divisible par 8.

Dans l'exemple précédent, on complète le code binaire en

`[0,0,1,0,1,1,1,1,0,1,0,0,0,0,0,0]` et

on obtient comme compression de `"bin"` suivant `d` le tableau `array([47,64])`, qui prend un octet de moins que le tableau représentant `"bin"`.

Question 7. *Décompression*

On considère un code `d` tel que `d['a'] = [1,0,0]`, `d['n'] = [1,1]` et `d['d'] = [0, 1, 0, 1]`.

Quel est la chaîne de caractère dont le code binaire suivant le code `d` est `[1,1,1,0,0,0,1,0,1]`.

Question 8. *Ambigüité*

On considère un code `d` tel que :

`d['a'] = [1,0,0]`, `d['i'] = [0]`, `d['n'] = [1,1]` et `d['d'] = [1, 1, 1, 0]`.

En examinant le code binaire `[1, 1, 1, 0, 0]`, expliquer pourquoi le code `d` ne peut pas être utilisé pour compresser les chaînes de caractères.

Question 9. *Codes Préfixe*

On dit qu'un code binaire `l1` de longueur n_1 est *préfixe* d'un code binaire `l2` de longueur $n_2 \geq n_1$ si et seulement si `l2[:`n_1`] = l1`.

On dira qu'un code `d` est un *code préfixe* si et seulement si pour toute paire de caractères `c1` $\neq$ `c2`, `d[c1]` n'est pas préfixe de `d[c2]`.

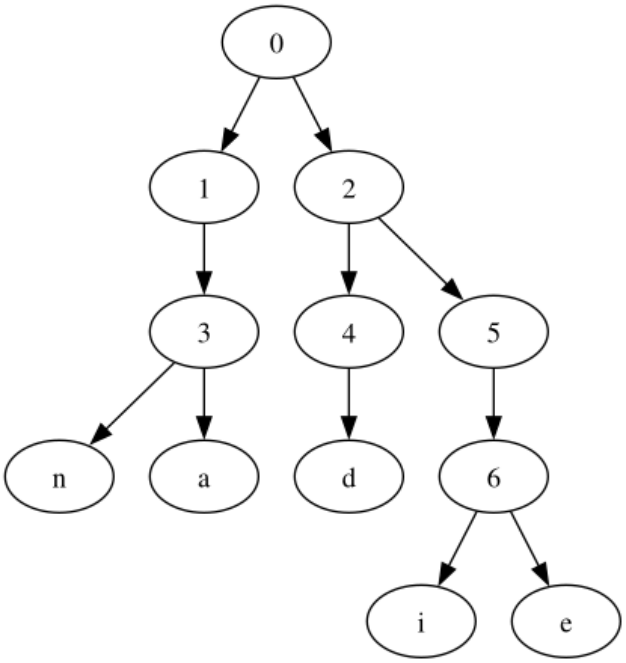
Implémenter une fonction `sans_prefixe(d : dict) -> bool` qui renvoie `True` si et seulement si `d` est un code préfixe.

Question 10. *Compression*

Implémenter une fonction `compression(s : str, d : dict) -> array[int8]` qui prend en entrée une chaîne `s` et un code préfixe `d` et qui renvoi la compression de `s` suivant `d` comme définie en introduction de la section.

V Arbres

On note $\mathcal{A}$ l'ensemble des caractères de la table ASCII. On rappelle que d_- (resp. d_+) représente le degré entrant (resp. sortant) d'un sommet dans un graphe orienté.



On appellera *arbre de Huffman* un graphe orienté (S, A) tel que :

1. $\exists n \in \mathbb{N}, \exists \mathcal{F} \subseteq \mathcal{A}, [\,0, n\,] \uplus \mathcal{F} = \mathrm{S}$. *i.e* le graphe possède n sommets numéroté de 0 à n, inclus, puis le reste des sommets sont des caractères de la table ASCII.
2. Le sommet 0 est appelé la *racine* de l'arbre et vérifie $d_-(0) = 0$
3. $\forall s \in \mathrm{S}$, s est accessible depuis 0.
4. $\forall s \in \mathrm{S} \setminus \{0\}$, $d_-(s) = 1$. Tous les sommets différents de la racine possède exactement 1 prédécesseur.
5. $\forall s \in \mathrm{S}$, $d_+(s) \leq 2$. Tous les sommets possèdent au plus deux successeurs.
6. $\forall s \in \mathrm{S}$, $d_+(s) = 0 \iff s \in \mathcal{A}$. Les sommets qui ne possèdent pas de successeurs sont appelés *feuilles* de l'arbre et sont exactement les sommets qui sont des caractères ASCII.

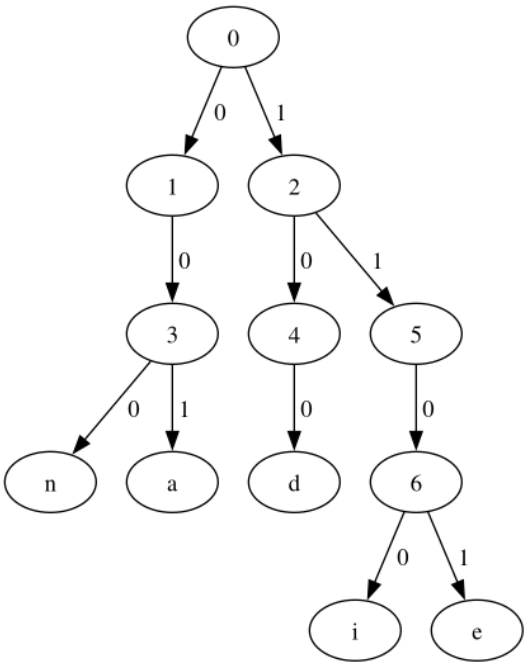
Un arbre de Huffman sera représenté par son dictionnaire d'adjacence.

Pour l'exemple ci-dessus :

```
g = {
    0 : [1,2],
    1 : [3],
    2 : [4,5],
    3 : ['n','a'],
    4 : ['d'],
    5 : [6],
    6 : ['i','e'],
    'n' : [], 'a' : [], 'd' : [], 'i' : [], 'e' : [] }
```

Dans le dictionnaire d'adjacence d'un arbre de Huffman, on appellera pour tout sommet `s` de l'arbre, s'ils existent, *successeur gauche* de `s` le successeur à l'indice 0 de la liste d'adjacence de `s`, `d[s][0]`, et *successeur droit* le successeur à l'indice 1, `d[s][1]`. Par exemple `1` est successeur gauche de `0` et `2` est successeur droit. Aussi `'d'` est successeur gauche de `4`, qui n'a pas de successeur droit.

On peut éventuellement voir l'arbre comme étant étiqueté de la manière suivante :



Question 11. *Accessibilité*

Implémenter une fonction `accessible(g : dict, s) -> bool` qui prend en entrée le dictionnaire d'adjacence d'un graphe orienté quelconque et `s` un sommet de ce graphe et qui renvoie `True` si et seulement si tous les sommets sont accessibles depuis `s` dans `g`.

L'algorithme *devra* utiliser un parcours en *largeur*.

Question 12. *Degrés*

Implémenter une fonction `degres(g : dict) -> dict` qui prend en entrée le dictionnaire d'adjacence d'un graphe orienté quelconque et qui renvoie un dictionnaire dont les clés sont les sommets du graphe et les valeurs sont un couple `(de,ds)` tel que `de` est le degré entrant du sommet et `ds` le degré sortant.

Question 13. *Vérification*

On supposera qu'il existe une fonction `est_ascii(c) -> bool` qui renvoie `True` si et seulement si `c` est un caractère de la table ASCII. La complexité de cette fonction est O(1).

Implémenter une fonction `est_arbre_huffman(g : dict) -> bool` qui prend en entrée le dictionnaire d'adjacence d'un graphe orienté quelconque et qui renvoie vrai si et seulement si le graphe est un arbre de Huffman. On supposera en précondition que les clés du dictionnaire sont bien de la forme $[\![0, n]\!] \uplus \mathcal{F}$ avec $\mathcal{F} \subseteq \mathcal{A}$.

Donner la complexité de l'algorithme en fonction de n et m le nombre de sommets et d'arêtes du graphe, respectivement.

Question 14. *Unicité du chemin*

La distance d'un sommet s à un sommet s' dans un graphe orienté est la longueur du plus petit chemin qui relie s à s'. Elle sera noté $\delta(s, s')$. Par convention on notera $\delta(s, s') = +\infty$ si s' n'est pas accessible depuis s.

Dans un arbre de Huffman, on défini le niveau d'un sommet comme étant la distance de ce sommet à la racine de l'arbre. Dans l'exemple, `2` est au niveau 1, `'d'` au niveau 3 et `'e'` au niveau 4. Par définition, tous les sommets d'un arbre de Huffman ont un niveau différent de $+\infty$.

Montrer que pour tout sommet s dans un arbre de Huffman, H, il existe un unique chemin de source la racine de H et de destination s. On pourra raisonner par récurrence sur le niveau de s.

VI Codage de Huffman

Dans un arbre de Huffman, on peut associer un code binaire de manière unique à chaque chemin dont la source est la racine.

Soit `c = [` s_0`, `s_1`, `s_2`, ..., `s_n` ]` un chemin de longueur `n` dans un arbre `a`. On définie `l = ϕ(c)` la liste de taille `n` telle que `l[i]` vaut 0 si s_i est le successeur gauche de s_{i-1}, 1 s'il est le successeur droit.

Dans l'arbre de Huffman donné précédemment en exemple, nous avons donc :

```
ϕ([0,2,5,6,'i']) = [1, 1, 0, 0]
```

D'après la section précédente, étant donné un arbre de Huffman, H, pour tout sommet s de H, il existe un unique chemin, $\gamma(s)$ de source la racine de H et de destination s. Pour tout sommet s de H appartenant à $\mathcal{A}$ (caractères de la table ASCII), on note $\psi(s) = \phi(\gamma(s))$, le code binaire associé au chemin allant de la racine à s.

Dans l'arbre de Huffman donné précédemment en exemple, nous avons donc :

— ψ(`'i'`) = `[1, 1, 0, 0]`
— ψ(`'n'`) = `[0, 0, 0]`
— ψ(`'d'`) = `[1, 0, 0]`

Enfin soit H un arbre de Huffman, le code de H est le dictionnaire `d` qui associe à chaque feuille f de H le code binaire $\psi(f)$.

Le code de l'arbre de Huffman donné précédemment en exemple est donc :

```
d = { 'n' : [0,0,0],
      'a' : [0,0,1],
      'd' : [1,0,0],
      'i' : [1,1,0,0],
      'e' : [1,1,0,1]
    }
```

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

Implémenter une fonction `code_arbre(a : dict) -> dict` qui prend en entrée le dictionnaire d'adjacence d'un arbre de Huffman `a` et renvoie le code de `a`.

Indication :

1. Pour tout sommet s de `a` différent de la racine. Trouver une relation entre le $\psi(s)$ et $\psi(s')$ avec s' l'unique prédécesseur de s.
2. On pourra utiliser un parcours de graphe et stocker dans un dictionnaire l'ensemble des valeurs de ψ calculées.

Question 16. *Nombre d'occurrences*

Implémenter une fonction `occurrences(s : str) -> dict` qui prend en entrée une chaîne de caractères `s` et qui renvoie un dictionnaire qui associe à chaque caractère de `s` son nombre d'occurrences dans `s`.

Question 17. *Paires minimale de clés*

Implémenter une fonction `paire_min(d : dict)` qui prend en entrée un dictionnaire ayant au moins 2 associations et dont les valeurs sont des entiers positifs et qui renvoie les deux clés associées aux valeurs les plus petites.

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

Pour construire un arbre de Huffman optimal pour la compression d'une chaîne de caractère `s`, contenant au moins deux caractères différents, on utilise l'algorithme suivant :

1. Calculer le dictionnaire `d` du nombre d'occurrences de chaque caractère de `s`.
2. Créer un graphe `g` dont les sommets sont les clés de `d` et n'ayant aucune arête.
3. Initialiser un compteur `i` à 1
4. Trouver les deux clés, `k1` et `k2` ayant la plus petite valeur dans `d`.
5. Enlever les associations `(k1,v1)` et `(k2,v2)` de `d`. On pourra utiliser `v = d.pop(k)` qui enlève l'association `(k,v)` de `d`.
6. Si `d` est vide, ajouter un sommet 0 ayant comme successeur `k1` et `k2` puis renvoyer `g`.
7. Ajouter dans `g` un sommet `i` ayant comme successeur `k1` et `k2`.
8. Ajouter dans `d` une clé `i` associé à la valeur `v1 + v2`.
9. Incrémenter `i`.
10. Reprendre à l'étape 4.

Implémenter une fonction `huffman(s : str) -> dict` qui prend en entrée une chaîne de caractère et renvoie l'arbre de Huffman optimal pour la compression de `s`.

Question 19. *Terminaison*

Montrer la terminaison de l'algorithme précédent.

Question 20. *Codage de Huffman*

On admet pour le moment que de code d'un arbre de Huffman est un code prefixe.

Implémenter une fonction `codage_huffman(s : str) -> array[int8]` qui prend en entrée une chaîne de caractère et renvoie la compression de `s` suivant le code de l'arbre de Huffman optimal pour la compression de `s`.

Question 21. *Décodage de Huffman*

Implémenter une fonction `decodage_huffman(comp : array[int8], h : dict) -> str` qui prend en entrée un tableau d'entiers 8 bits obtenu par compression d'une chaîne suivant le code de `h`, un arbre de Huffman, et renvoie la chaîne `s` compressé suivant le code de `h`.

Question 22. *Code Préfixe*

Montrer que le code associé à un arbre de Huffman est un code préfixe.

VII Annexe

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL