Codages de Shanon-Fano et Huffman

1 Preliminaires

L'objectif est de compresser un flux de caractères (ou symboles) en utilisant le fait que les caractères du flux n'ont pas la même fréquence. C'est typiquement le cas d'un texte dans une langue, par exemple en anglais et en français la lettre 'e' est beaucoup plus fréquente que la lettre 'z'.

Avant de détailler les deux codages proposés, il faut d'abord mesurer la fréquences des caractères dans notre texte.

On fournit trois fichiers :

- `compress_cdls.ml`, le code à compléter, il contient quelques fonctions d'affichage et la lecture du contenu d'un fichier.
- `test.txt`, un fichier pour tester nos fonctions contenant le mot "abracadabra" suivi d'une fin de ligne.
- `mobydick.txt`, le contenu du livre éponyme en anglais pour comparer les deux encodages.

Pour lancer le programme sur le fichier de test, on exécutera la commande : `ocaml compress_cdls.ml test.txt`.

Dans toute la suite on suppose qu'on manipule des textes encodés en ASCII. On pourra ainsi utiliser la fonction `Char.code c`. On utilisera de manière interchangeable les mots « caractère » et « symbole ».

1. Mesure des fréquences

Écrire une fonction `compte_occurrences string -> (int * int) array` qui renvoie un tableau de couple `(nb_occurrences, code_symbole)` triés par ordre de nombre d'occurrences décroissantes de chaque caractères.
On pourra utiliser la fonction `Array.stable_sort`. On rappelle qu'il y a 256 symboles dans la table ASCII (même si en toute rigueur on pourrait se contenter de 128 ici).
Pour la chaine `"abracadabra"`, la fonction renvoie le tableau `[|(97,5); (98,2); (114,2); (10,1); (99,1); (100,1); (0, 0); (1, 0); ...|]` avec de nombreux caractères non présents (le compte vaut 0).

Écrire une fonction `liste_occurrences string -> (int * int) list` qui transforme le tableau précédent en une liste dont les éléments non présents ont été supprimés. Cette étape n'est pas strictement nécessaire mais elle aidera à débugger les fonctions futures. Pour la chaine `"abracadabra"`, la fonction renvoie la liste `[(97,5); (98,2); (114,2); (10,1); (99,1); (100,1)]`.

Les codages de Shanon-Fano et Huffman utilisent ces fréquences pour assigner un code de longueur variable à chaque caractère. Dans la suite on appelera le *poids* d'un caractère son nombre d'occurrences dans le texte.

On s'impose comme contrainte que les codes associés à chaque caractère forment un *codage préfixe*. Un codage est dit préfixe, si aucun des codes de ses caractères n'est le préfixe d'un autre. Par exemple `a: 1, b: 10, c: 0` n'est pas préfixe car le code de `a` est le préfixe du code de `b`. L'usage d'un codage préfixe rend le décodage très facile.

2. Décodage manuel

Décoder au brouillon le flux `101010000011` sachant que le codage préfixe utilisé est `a: 1, b: 01, c: 00`.

2 Codage de Shanon-Fano

L'idée du codage de Shanon-Fano -qui est la version historique (1949)- est d'ordonner les symboles par ordre de poids décroissants puis de les séparer récursivement en deux

ensembles de poids les plus proches possibles. Le code des éléments du premier ensemble seront prolongés d'un 0 et ceux de l'autre d'un 1. On répète cette séparation jusqu'à obtenir des ensembles d'un seul élément. On a alors le code de ce symbole.

3. Separation

Écrire une fonction `separe (int*int) list -> int -> ((int*int) list * int) * ((int*int) list * int)`. La fonction prend en argument une liste de couples `(poids,symbole)` et renvoie deux couples `(sous-ensemble, poids du sous ensemble)`.
Par exemple l'appelle à `separe [(97,5), (98,2), (114,2), (10,1), (99,1), (100,1)] 12` renvoie `(([(97,5)], 5), ([(98,2), (114,2), (10,1), (99,1), (100,1)], 7))`.
On remarque que dans ce cas précis, placer `(98,2)` dans l'ensemble de gauche donne aussi une différence de poids de 2 entre les deux ensembles donc ce résultat est valide aussi.

On prendra garde à renvoyer les deux ensembles dans l'ordre décroissant des poids pour que `separe` puisse être appelée récursivement.

Indication : passer par des nombres flottants pour trouver la taille de la moitié et avoir un calcul qui marche aussi dans le cas où le poids total est impair.

4. Construction de la table de codage

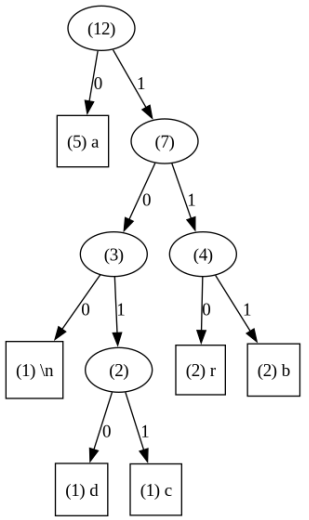
Écrire une fonction `table_shanonfano (int*int) list -> int -> string array` qui renvoie la table de codage de Shanon-Fano pour les occurrences données (et le nombre d'occurrences totales). Si le caractère n'est pas présent dans la liste on laissera une chaine de caractères vide dans la table. Par exemple le code d'un caractère c qui serait la chaine 0011 signifie qu'il y a eu 4 séparations avant que c se retrouve seul, il a été successivement dans les ensembles de gauche, gauche, droite et droite.

Indication : Découper récursivement les ensembles en complétant les codes au fur et à mesure. Quand un ensemble est de taille 1, on peut stocker son code dans la table.

Vérifier visuellement sur l'exemple de test que le codage calculé est bien préfixe et que les découpages ont été bien faits.

3 Codage de Huffman

Il arrive pour certaines distributions que le codage de Shanon-Fano ne trouve pas un nombre de bits moyens par caractère optimal (seulement très proche). Le codage de Huffman résout ce problème. Il procède non pas par séparation mais par regroupement. On trie cette fois les symboles par poids croissants (du moins probable au plus probable). On construit un arbre de codage dont les feuilles sont les symboles. Le code d'un symbole se lit depuis la racine jusqu'au symbole en question.



On donne le type suivant pour les noeuds de cet arbre, les feuilles contiennent un couple `(poids, symbole)`, les noeuds internes représentent un pseudo-symbole (le résultat d'un regroupement) et contiennent le poids du pseudo-symbole et les deux sous-arbres :

```
type noeud =
| Feuille of int * int
| Interne of int * noeud * noeud
```

Le processus de regroupement regroupe les deux sous-arbres de poids les plus faibles et les combine en un nouveau pseudo-symbole. La branche de gauche débute son code par un `0` et celui de droite par un `1`. Ce nouvel arbre est réinséré dans la liste des symboles *en maintenant l'ordre croissant* des poids.

5. Regroupement de deux nœuds

Écrire une fonction `combine noeud -> noeud -> noeud` qui crée le pseudo-symbole correspondant aux deux nœuds donnés.

6. Insertion dans une liste triée

Écrire une fonction `insere_trie noeud -> noeud list -> noeud list` qui prend en argument un nœud et une liste triée par poids croissants. La fonction insère le nouveau nœud dans la liste en maintenant l'ordre sur les poids.

7. Construction de la table de codage

Écrire une fonction `table_huffman(int*int) list -> string array` qui renvoie la table de codage de Huffman pour les fréquences données. Si le caractère n'est pas présent dans la liste on laissera une chaine de caractères vide dans la table. Par exemple le code d'un caractère c qui serait la chaine `0011` signifie qu'il y a eu 4 regroupement avant qu'il ne reste qu'un pseudo-symbole, il a été regroupé depuis les sous-arbres de droite, droite, gauche, gauche (les nœuds sont regroupés depuis les feuilles mais les codes se lisent depuis la racine).

Indication : Convertir les couples `(poids,symbole)` en feuilles. Tant que cette liste contient plus de un élément, on procède à un regroupement qu'on réinsère à la bonne position. Quand l'arbre est terminé on peut remplir la table en le parcourant en profondeur jusqu'à ses feuilles.

Quelques remarques

- On peut montrer que le codage de Huffman est un *codage caractère optimal* (la preuve est trouvable facilement en ligne). Un codage caractère est un codage qui associe à chaque caractère un code.

  Cela semble complètement obligatoire mais on peut remarquer que le nombre de bits moyen pour un caractère est rarement entier (pour plus de précisions voir le concept d'entropie de Shanon). Or il faut produire en sortie un nombre entier de bits par caractère. Il y a alors une redondance (en terme d'entropie) qui correspond à l'écart entre cette partie décimale et l'entier supérieur qui s'accumule tout au long de l'encodage.

  Le codage arithmétique permet d'encoder exactement la fréquence des caractères en encodant directement tout le message en un nombre décimal de précision arbitraire. C'est justement tout l'enjeu de l'implémentation de ce codage, il faut pouvoir représenter un nombre décimal de précision arbitraire en utilisant l'arithmétique entière modulaire du processeur.

- La construction d'un codage de Huffman est un algorithme glouton, et pourtant c'est un codage caractère optimal. Il faut penser au codage de Huffman avant de dire (à tort) qu'un algorithme est glouton car il n'est pas optimal. Ainsi, en pensant au codage de Huffman on dira plutôt qu'un algorithme est glouton car il procède *localement*. Dans l'arbre de Huffman, on regroupe au fur et à mesure les nœuds pendant la construction.