On travaillera depuis la machine virtuelle ClefAgreg2019. On commencera par créer un fichier OCaml `TP19.ml`, initialement vide.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = ocamlc` ou `ocamlopt` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal).

```
$ COMPILATEUR -o TP19.exe TP19.ml
$ ./TP19.exe
```

## Introduction

On s'intéresse dans ce TP à la méthode de compression (et de décompression) dite de Huffman.

Le codage de Huffman (1952) est un codage de type statistique utilisé pour la compression sans perte de données. Il utilise un code à longueur variable pour représenter un symbole $a \in \Sigma$ (un caractère ASCII dans le cas de données de type texte). Le code est déterminé à partir d'une estimation des probabilités d'occurrence des symboles, un code court étant associé aux symboles les plus fréquents. Le codage de Huffman est un code optimal, au sens de la plus courte longueur.

L'algorithme du codage de Huffman, sur un texte `t`, construit un arbre binaire $A_H(\texttt{t})$ appelé arbre de Huffman. Cet arbre permet de définir un codage binaire d'un texte `t` composé de caractères d'un alphabet $\Sigma$ (on s'intéressera uniquement à $\Sigma = [\![0; 255]\!]$ l'alphabet ASCII 256 dans ce TP). Les feuilles de cet arbre $A_H$ correspondent aux caractères $a$ du texte `t` avec leur fréquence d'apparition $f_a \in [0, 1]$ (un nombre rationnel). La structure de l'arbre permet d'organiser les feuilles de manière à ce que les feuilles dont les caractères ont une fréquence élevée soient proches de la racine (voir exemple page suivante).

À chaque nœud interne $n = (A_1, f(n), A_2)$ de l'arbre est associée une valeur numérique $f(n) \in [0, 1]$ qui est la somme des valeurs de son fils gauche $A_1$ et de son fils droit $A_2$.

Enfin, les branches sont étiquetées, par des bits 0 ou 1, de sorte que le codage de chaque caractère `a` de `t` est la concaténation des étiquettes rencontrées sur le chemin menant de la racine de $A_H$ à la feuille portant le caractère `a`.

Les étiquettes sont codées par une fonction appelée clé de codage.

> Définition : Une *clé de codage binaire* de $\Sigma$ est une application injective $c : \Sigma \to \{0, 1\}^*$.
>
> Cette clé de codage binaire est dit un *codage préfixe* si pour tout $a, b \in \Sigma$, $c(a)$ n'est pas un préfixe de $c(b)$ et $c(b)$ n'est pas un préfixe de $c(a)$. Utiliser un codage préfixe permet d'assurer que le décodage soit non ambigu.

Par exemple, si le texte `t` ne contient que les symboles `a`, `b`, `c` et `d`, on peut proposer la clé de codage binaire suivante : $c(\texttt{a}) = 1$, $c(\texttt{b}) = 001$, $c(\texttt{c}) = 01$ et $c(\texttt{d}) = 000$ (voir exemple page suivante). C'est bien un codage préfixe.

Un autre exemple de clé de codage binaire sur ce texte peut être $c(\texttt{a}) = 1$, $c(\texttt{b}) = 10$, $c(\texttt{c}) = 01$ et $c(\texttt{d}) = 00$, qui par contre n'est pas un codage préfixe ($c(\texttt{a}) = 1$ est préfixe de $c(\texttt{b}) = 10$).

 

Définition : Soient `t` un texte et $A$ un arbre binaire dont les feuilles sont étiquetées par des couples $(a, f_a)$, $\forall a \in$ `t` ($f_a \in [0,1]$ est la fréquence de ce symbole $a$ dans le texte `t`), et les arêtes par une valeur 0 (gauche) ou 1 (droite).

On code chaque caractère $a$ par la concaténation des 0 ou 1 obtenus en parcourant le chemin de la racine de $A$ au nœud contenant le caractère $a$. On définit ainsi une clé de codage $c(a)$ pour $a$.

On appelle *coût du codage de* $a \in$ `t` selon la clé de codage $c$ la quantité $q_a = f_a|c(a)|$, où $|c(a)|$ est le nombre de bits utilisés pour coder $a$. On appelle coût de codage de `t` la quantité $Q(\texttt{t}) = \sum_{a \in \texttt{t}} q_a$.

L'algorithme de Huffman, sur un texte $\texttt{t} \in \Sigma^*$, construit en temps polynomial sur `|t|` un tel arbre binaire $A_H(\texttt{t})$ tel que son coût $Q(\texttt{t})$ soit minimal. On ne prouvera pas ici cette propriété.

## Codage de Huffman

L'algorithme suivant réalise la construction de l'arbre de Huffman $A_H(\texttt{t})$ d'un texte `t` défini sur un alphabet $\Sigma$.

À noter que cet arbre n'est pas unique. Il dépend de l'ordre des feuilles dans la liste initiale (par exemple, deux caractères de même fréquence peuvent être permutés).

Définition : Une forêt d'arbres binaires est un ensemble d'arbres binaires.

```
Algorithme : Codage de Huffman, fonction CodageHuffman(t)
Entrées : un texte t sur un alphabet Sigma (ASCII 256)
Sortie : L'arbre de Huffman A_H(t) associé au texte t

Calculer les fréquences f_a de chaque symbole a dans Sigma selon
         son nombre d'apparition dans le texte t ;
Créer une forêt F d'arbres binaires, chacun réduit à un seul nœud
    (f_a, a) pour tout a dans Sigma ;

Tant que F contient au moins deux arbres Faire :
    Extraire de F les deux arbres A1 et A2 avec les fréquences f_A1 et f_A2
             les plus petites à leurs racines ;
    Créer un nouveau nœud n de fréquence f_n = f_A1 + f_A2 ;
    Créer un nouvel arbre binaire A = (f_n, A1, A2) et l'ajouter dans la forêt F ;
Fin Tant que

Soit A_H = l'unique arbre de la forêt F ;
Étiqueter chaque branche de A_H par 0 pour le sous-arbre gauche,
                                par 1 pour le sous-arbre droit ;
Renvoyer A_H.
```

 

L'algorithme de Huffman est un algorithme qui suit le **paradigme glouton** : quand un choix est fait (extraire deux arbres de la forêt pour les fusionner), ce choix minimise localement un critère d'optimalité (prendre les arbres de fréquences minimales).

### Exemple de codage de Huffman

Soit $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ et `t="abacdaca"` sur cet alphabet. Les différentes étapes de Huffman sont représentées ci-dessous.

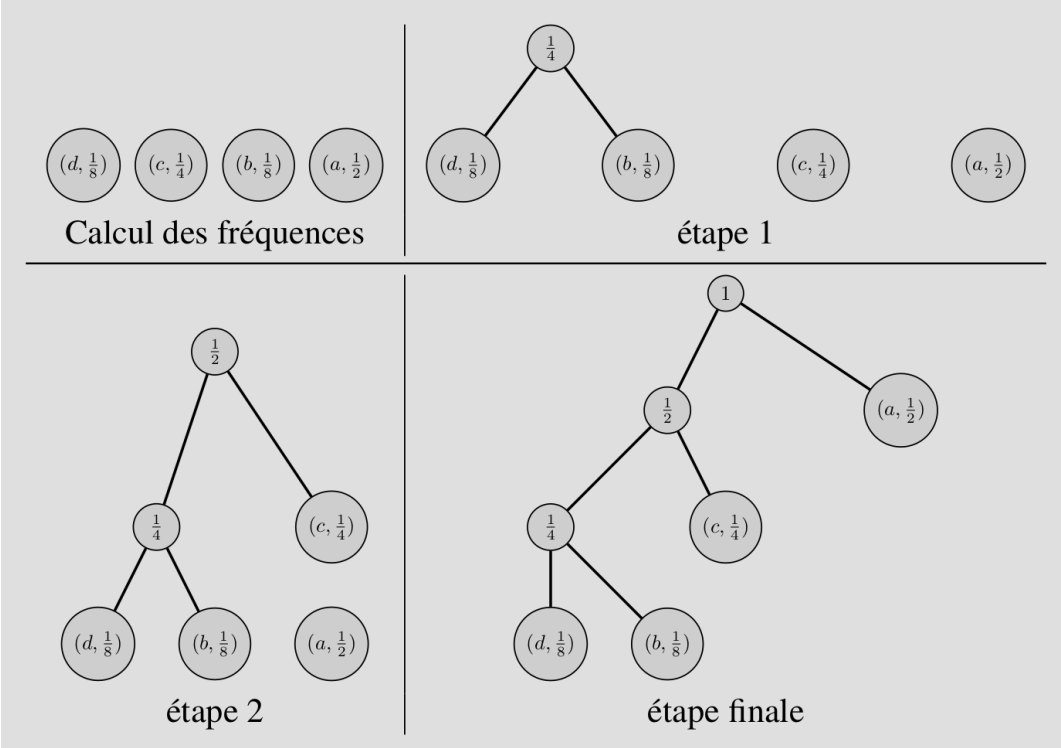


Figure 1 – Illustration des étapes successives de l'algorithme du codage de Huffman, qui produit l'arbre binaire étiqueté $A_H(\texttt{t})$ depuis le texte `t`.

En codant les branches des sous-arbres gauches à 0, et celles des sous-arbres droits à 1, on a donc le codage suivant : $c(\texttt{a}) = 1$, $c(\texttt{b}) = 001$, $c(\texttt{c}) = 01$ et $c(\texttt{d}) = 000$. Ainsi, le mot `t="abacdaca"` est codé en `1 001 1 01 000 1 01 1`, soit 14 bits. Le texte ayant 4 caractères, ces derniers pouvaient être codés sur 2 bits, soit un codage natif optimal en 16 bits. Le codage de Huffman permet donc de gagner 2 bits, soit plus de 12%. Bien sûr, sur des textes plus longs, le gain peut être bien plus appréciable.

### A implémenter pour le codage de Huffman

On utilisera des nombres d'occurrences (entiers) plutôt que des fréquences (rationnelles) car c'est plus simple à manipuler.

Les occurrences de caractères seront donnés dans un tableau `int array` de $|\Sigma| = 256$ valeurs, où la $i$ème valeur donne la fréquence du caractère de numéro $i$ dans le texte d'entrée `t`. On aura alors des valeurs nulles pour les caractères non présents, elles seront simplement ignorées lorsqu'on va construire l'arbre.

 

1. Définir une variable globale `taille_alphabet` valant $|\Sigma| = 256$.
2. Écrire une fonction `calculer_frequences (t:string) : int array` qui renvoie le tableau des occurrences associé au texte `t` donné en argument. On utilisera `Char.code : char -> int` pour obtenir l'entier entre 0 et 255 correspondant à un caractère $a \in \Sigma$ donné.
3. Tester cette fonction sur le message `t="abacdaca"`, qui devrait donner un tableau de taille 256, valant 0 partout sauf en indices 97, 98, 99 et 100 où il vaut respectivement 4, 1, 2 et 1.

On va utiliser le type suivant pour les arbres binaires de Huffman :

```
type arbre = Feuille of int | Noeud of arbre * arbre;;
```

On identifie ainsi complètement un caractère $a$ à l'octet, et donc l'entier entre 0 et 255, qui le représente.

Il n'est pas utile de stocker les fréquences $f(a)$ des caractères $a$ aux feuilles, ni les valeurs $f(n)$ des nœuds, car elles seront utilisées dans le stockage de la forêt $F$. Cette forêt sera implémentée comme une liste `(int * arbre) list`, triée par ordre croissant selon la valeur de cet entier qui représente $f(a)$ aux feuilles et $f(n)$ aux nœuds internes.

```
type foret = (int * arbre) list;;
```

4. Écrire une fonction `creer_foret_pas_triee (t:string) : foret` qui calcule la table des fréquences des symboles $a$ apparaissant dans `t`, puis qui crée la forêt $F$ en y ajoutant un par un les arbres réduits aux feuilles $(f(a), \texttt{Feuille}(a))$ pour chaque lettre $a$ qui apparaît au moins une fois dans le texte `t` (ie. telle que $f(a) > 0$). On ne cherchera *pas* à trier les feuilles selon leurs fréquences d'apparition (questions 6. et 7.).
5. Écrire une fonction `compare_int_arbre (f_x, arbre_x) (f_y, arbre_y)` qui renvoie 0 si `f_x = f_y`, $+1$ si `f_x > f_y`, et $-1$ si `f_x < f_y`. Cette fonction ignore les termes de droite des deux tuples et ne s'intéresse qu'aux entiers `f_x` et `f_y`.
6. En utilisant `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` (cf. `https://ocaml.org/api/List.html#VALsort`) et `compare_int_arbre`, écrire une fonction `trie_foret : foret -> foret` qui trie la forêt par ordre de fréquences croissantes.
7. Écrire une fonction `creer_foret_triee (t:string) : foret` qui crée la forêt $F$ et la trie selon les fréquences d'apparition (par ordre croissant, avec donc les lettres de fréquences les plus faibles en premier).
8. Écrire une fonction `contient_au_moins_deux_arbres : foret -> bool` qui teste si la forêt $F$ contient au moins deux arbres (avec juste un appel à `List.length`). Ce sera la condition d'arrêt de la boucle `while` dans l'algorithme de codage de Huffman.
9. Écrire une fonction `extraire_arbre_freq_minimale : foret -> (int * arbre) * foret` qui extraie de la forêt la paire $(f(A), A)$ telle que $f(A)$ soit minimum dans la forêt (on suppose qu'elle est triée par $f(A)$ croissant) et renvoie la forêt privée de cette paire.

Dans l'algorithme de codage de Huffman, dans la boucle `while`, il faudra utiliser deux fois cette fonction comme suit :

```
let (fA1, a1), foret2 = extraire_arbre_freq_minimale !foret in
foret := foret2;
let (fA2, a2), foret3 = extraire_arbre_freq_minimale !foret in
foret := foret3;
...
```

 

10. Écrire enfin une fonction `calculer_arbre_Huffman (t:string) : arbre` qui implémente l'algorithme de codage de Huffman pour calculer l'arbre $A_H(\texttt{t})$ depuis le texte `t`.
    - On manipulera la forêt $F$ comme une référence sur une `foret` (ie. une `foret ref`).
    - Pour maintenir la forêt $F$ triée on ne cherchera pas à être efficace mais on utilisera la fonction `trie_foret`. (Pour une vraie implémentation on utiliserait une structure de tas binaire)
    - Pour conclure et extraire le dernier couple $(f(A), A)$ de la forêt, après la boucle `while`, on pourra utiliser simplement `let fA, a = List.hd !foret`.
11. Tester `calculer_arbre_Huffman` sur le message `t="abacdaca"`, qui devrait donner `Noeud (Noeud (Noeud (Feuille 100, Feuille 98), Feuille 99), Feuille 97)`, comme dans l'illustration en page précédente.
12. Tester `calculer_arbre_Huffman` sur un autre message de votre choix, qui utilise peu de lettres mais certaines avec beaucoup de répétitions.
    - Dessiner au brouillon l'arbre obtenu avec les valeurs des fréquences associées aux feuilles et aux nœuds internes.
    - Étiqueter l'arbre avec 0 sur les branches à gauche, et 1 sur les branches à droite, et en déduire la clé de codage de chaque lettre présente dans le texte `t` choisi.
    - En conclure l'encodage $\tilde{t}$ correspondant à `t`.
    - Compter le nombre de bits de cet encodage, et le nombre de bits nécessaire à l'encodage naïf du texte `t`, et calculer le taux de compression comme dans l'exemple précédent.

### Utilisation de l'arbre de Huffman pour écrire en binaire le message d'entrée

Un codage $c$ sera représenté en OCaml par le type suivant :

```
type codage = (int list) array;;
```

Le tableau contiendra $|\Sigma| = 256$ listes, vide si le caractère associé n'est pas présent dans le texte `t`, et une liste d'entiers 0 ou 1 correspondant au codage d'un caractère $a$ s'il est présent dans le texte `t`.

Par exemple pour `t="abacdaca"`, le codage associé trouvé par l'algorithme de Huffman sera un grand tableau comme cela :

```
[|[]; []; ...; []; [1]; [0; 0; 1]; [0; 1]; [0; 0; 0]; []; ...; []; []|]
```

Utiliser des `bool list` serait plus économe en mémoire, mais cela serait moins lisible.

13. Écrire une fonction `calculer_codage_symboles (arb:arbre) : codage` qui calcule le codage associé à l'arbre `arb`. On commencera par créer un tableau de taille `taille_alphabet`, rempli de listes vides `[]`, puis on utilisera une fonction récursive auxiliaire (interne) `aux : arbre -> (int list) -> unit`, qui parcourt l'arbre selon l'algorithme suivant :
    - Si `arb = Noeud(a1, a2)`, on applique `aux` à `a1` avec le nouvel accumulateur `0 :: acc` (on est allé à gauche pour lire le sous-arbre `a1`), et on applique `aux` à `a2` avec le nouvel accumulateur `1 :: acc` (on est allé à droite).

 

- Si `arb = Feuille (lettre)`, on stocke dans `codages.(lettres)` le miroir de la liste `acc` (`List.rev acc`).
- Enfin, hors de la fonction `aux`, on a juste à l'appliquer avec `aux arb []`, puis à renvoyer la table `codages` qui a été bien remplie.

14. Vérifier le codage obtenu à partir de l'arbre $A_H$(`t`) sur l'exemple `t = "abacdaca"`.
15. Écrire une fonction `appliquer_codage (t:string) (c:codage) : int list` qui applique le codage `c` sur chaque caractère du texte `t`, pour produire une liste d'entiers 0 ou 1 (des bits).
16. Écrire une fonction `encode_Huffman (t:string) : int list` qui calcule l'arbre $A_H$(`t`) puis qui applique la fonction précédente pour encoder le texte `t`.
17. Vérifier que sur l'exemple `t = "abacdaca"`, on obtient l'encodage `[1; 0; 0; 1; 1; 0; 1; 0; 0; 0; 1; 0; 1; 1]` qui correspond bien à 10011010001011 comme expliqué dans l'exemple.

---

## Décodage de Huffman

Le décodage utilise l'arbre de Huffman $A_H$(`t`) et applique l'algorithme suivant sur un encodage $\tilde{t}$ (une suite de bits 0 ou 1, ou une `int list` dans le cas de notre code OCaml).

En partant de la racine de $A_H$, on suit le chemin indiqué par les booléens successifs de $\tilde{t}$ (en allant dans le sous-arbre gauche si on lit 0, et le sous-arbre droit si on lit 1). Dès que l'on arrive à une feuille `Feuille(a)`, pour un entier $a \in [\![0; 255]\!]$, on ajoute au résultat le caractère associé à cette feuille (obtenu avec `Char.chr a` qui donne le `char` associé à cet entier), et on recommence le décodage à la racine de $A_H$. Quand le code $\tilde{t}$ est parcouru en entier, on a reconstruit le texte `t` initial.

18. Implémenter cet algorithme en une fonction `decode_Huffman (tt : int list) (arb:arbre) : string`. On pourra construire une liste de chaîne de taille 1, et à la fin les concaténer avec `String.concat "" (List.rev !sortie)`.

---

## L'histoire complète

Pour avoir une méthode de compression (et de décompression) complète, en plus de calculer l'arbre $A_H$(`t`) et l'encodage associé au texte `t`, il faudrait un moyen d'écrire cet arbre $A_H$ en binaire dans l'encodage de sortie. En effet, la méthode de décompression implémentée à la partie précédente nécessite de connaître l'arbre $A_H$, qui doit donc être stocké en même temps que l'encodage $\tilde{t}$ du texte `t`.

En général, le codage proposé pour l'arbre $A_H$ est le suivant :

- Une feuille `Feuille(a)` est codée par un octet `1` suivi de l'octet `a`.
- Un nœud `Noeud(x,y)` est codé par un octet `0`, suivi du codage de `x` puis du codage de `y`.

Il s'agit en fait de coder l'ordre préfixe à plat. Comme l'arbre est binaire strict, cela suffit à le reconstruire uniquement.

Cela fera l'objet d'une partie du DM des prochaines vacances.

 