D.S. 03

Arbres binaires

1 Arbres d'intervalles, adapté de CCP 2010

De nombreux algorithmes reposent sur la manipulation d'ensembles d'éléments ordonnés. Lorsque ces ensembles contiennent de nombreux éléments adjacents (aucune valeur n'existe entre les deux éléments), il est plus performant en terme de temps de calcul et d'occupation mémoire de manipuler des intervalles au lieu de valeurs singulières. Cela permet également de manipuler des ensembles infinis de réels ou de rationnels sous la forme d'un ensemble fini d'intervalles contenant un nombre de valeurs infinies.
Nous nous limiterons à des intervalles fermés de $\mathbb{R}$ dont les bornes sont des entiers : $[min; max]$ avec $min \leqslant max$.

1.1 Définitions

1. Deux intervalles sont **disjoints** si leur intersection est vide.
2. La **fusion** de deux intervalles est l'intervalle qui a, comme minimum, le plus petit des minima des deux intervalles, et comme maximum, le plus grand des maxima des deux intervalles. Cette opération correspond à l'union des intervalles si ceux-ci ne sont pas disjoints.
3. Un intervalle est représenté par un couple de deux entiers, son minimum en premier et son maximum en second.

```
type intervalle == int * int;;
```

Question 1
Écrire une fonction `disjoints : intervalle -> intervalle -> bool`
telle que l'appel `disjoints i1 i2` *renvoie la valeur* `true` *si les intervalles* `i1` *et* `i2` *sont disjoints et la valeur* `false` *sinon.*
`disjoints (4, 7) (1, 3)` doit renvoyer `true`.

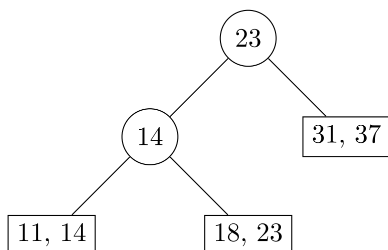
Question 2
Écrire une fonction `fusion : intervalle -> intervalle -> intervalle`
telle que l'appel `fusion i1 i2` *renvoie un intervalle correspondant à la fusion de* `i1` *et* `i2`.

Un arbre binaire d'intervalles a est une structure qui peut être :

- soit vide, notée $\emptyset$,
- soit une feuille qui contient un intervalle $[min; max]$,
- soit un nœud qui contient un entier noté $\text{sep}(a)$, un sous-arbre gauche (noté $G(a)$) et un sous-arbre droit (noté $D(a)$) qui sont tous deux des arbres binaires d'intervalles **non vides**.
- L'union des intervalles d'un arbre a est noté $\mathcal{E}(a)$.

Un arbre binaire d'intervalles est représenté par le type CaML :

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




```
let a0 = Noeud(Noeud(Feuille(11, 14),
                     14,
                     Feuille(18, 23)
                    ),
               23,
               Feuille(31, 37)
              );;
```

Un exemple d'arbre d'intervalle, a_0

Question 3
Écrire une fonction `suite : arbre -> intervalle list`
telle que l'appel `suite a` *renvoie la liste des intervalles des feuilles de* `a` *dans l'ordre obtenu par un parcours (infixe, préfixe ou suffixe) de l'arbre.*
`suite a0` doit renvoyer `[(11, 14); (18, 23); (31, 37)]`.

L'**intervalle englobant** d'un arbre d'intervalles, noté $\mathcal{I}(a)$, est le plus petit intervalle qui contient tous les intervalles contenus dans les feuilles de l'arbre.

Question 4
Écrire une fonction `englobant : arbre -> intervalle`
telle que l'appel `englobant a` *renvoie l'intervalle englobant de* `a` *pour* `a` *non vide.*
`englobant a0` doit renvoyer `11, 37`.

Un arbre d'intervalles **bien formé** est un arbre d'intervalles qui respecte les contraintes suivantes :

1. les intervalles contenus dans les feuilles sont disjoints deux à deux,
2. pour tout nœud n, l'intervalle englobant $\mathcal{G}(n)$, $[min_g; max_g]$, et l'intervalle englobant $\mathcal{D}(n)$, $[min_d; max_d]$ vérifient $max_g = \text{sep}(n) < min_d$.

a_0 est bien formé. Tout arbre vide est bien formé.

Question 5
Écrire une fonction `verifie_bf : arbre -> bool`
telle que l'appel `verifie_bf a` *renvoie* `true` *ou* `false` *selon que* `a` *est bien formé ou non.*
On privilégiera un algorithme qui ne parcourt qu'une seule fois l'arbre.

Question 6
Écrire une fonction `appartient : int -> arbre -> bool`
telle que l'appel `appartient k a` *renvoie* `true` *ou* `false` *selon que* `k` *appartient ou non à l'un des intervalles de* `a` *qu'on suppose bien formé.*
On attend une complexité proportionnelle à la hauteur de l'arbre.
Les entiers de $\mathcal{E}(a)$ sont les entiers k tels que `appartient k a` renvoie `true`.

Question 7
Écrire une fonction `ajoute_simple : intervalle -> arbre -> bool`
telle que l'appel `ajoute_simple (min, max) a` *renvoie un arbre d'intervalle bien formé contenant les intervalles de* `a` *et l'intervalle* `(min, max)`*.*
On suppose que `a` *est bien formé et que* `(min, max)` *est disjoint de tous les intervalles de* `a`*.*
On attend une complexité proportionnelle à la hauteur de l'arbre.

2 Arbres de Huffman, Mines 2006

Le problème est consacré à l'algorithme de Huffman qui permet de coder un texte caractère par caractère à l'aide d'une chaîne binaire en minimisant la longueur totale de la chaîne obtenue ; cet algorithme permet de faire de la compression de données. Les parties 1 et 2 du problème sont des parties préparatoires, le codage d'un texte sera abordé dans la troisième partie.

Dans tout le problème, on utilise des arbres binaires.
Pour un arbre, les termes de nœud et de sommet sont synonymes ; c'est le terme de nœud qui est retenu dans ce problème. Un nœud qui n'a pas de fils est appelé **feuille** alors qu'un nœud qui a au moins un fils est appelé un **nœud interne**.
Chaque nœud n des arbres binaires de ce problème contient, outre les indications concernant ses éventuels fils gauche et droit (voir plus bas), un caractère appelé **lettre du nœud** et noté *lettre(n)* et un entier strictement positif appelé **poids du nœud** et noté *poids(n)*.
Ces arbres binaires sont appelés **H_arbres**.
Une **forêt** est dans ce sujet une collection de H_arbres.
Une forêt est représentée en mémoire par :

- le nombre de H_arbres de la forêt, noté *nb_arbres*,
- le nombre total de nœuds, noté *nb_nœuds*,
- un tableau de nœuds appelé *table*. Dans tout le problème, ce tableau est supposé suffisamment grand pour contenir les nœuds de tous les H_arbres de la forêt considérée, les nœuds sont rangés dans le tableau entre les indices 0 et *nb_nœuds - 1*

ATTENTION : les nœuds qui sont les racines des H_arbres de la forêt se trouvent nécessairement au début du tableau *table*, c'est-à-dire entre les indices 0 et *nb_arbres - 1*.
Pour un nœud donné, les fils gauche et droit sont indiqués par leurs indices dans le tableau table des nœuds ; lorsqu'un fils gauche ou droit n'existe pas, cela est indiqué par une valeur d'indice égale à -1. Le fils gauche d'un nœud n sera noté *fg(n)* et le fils droit sera noté *fd(n)*.

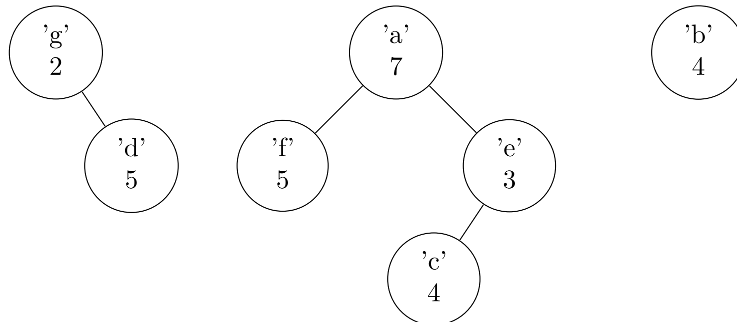


FIGURE I.1 – Un exemple introductif, noté *F_ex*

Pour la forêt *F_ex*, on a : *nb_arbres* $= 3$, *nb_nœuds* $= 7$; une table qui peut la représenter est

Indice du nœud	0	1	2	3	4	5	6
Lettre	'g'	'a'	'b'	'e'	'f'	'c'	'd'
Poids	2	7	4	3	5	4	5
Fils gauche	-1	4	-1	5	-1	-1	-1
Fils droit	6	3	-1	-1	-1	-1	-1

Dans cette table, on voit que le nœud d'indice 3 contient la lettre 'e' et le poids 3, que son fils gauche se trouve à l'indice 5 de la table et qu'il n'a pas de fils droit.

On définit les types suivants

```
type noeud = {lettre : char;  poids : int;
                  fg : int;      fd : int};;

type foret= {mutable nb_arbres : int;
             mutable nb_noeuds : int;
                         table : noeud array};;
```

On définit un nœud vide comme valeur par défaut dans la définition d'un tableau de nœuds :

```
let noeud_vide = {lettre='\000'; poids=0; fg=(-1); fd=(-1)};;
```

La forêt *F_ex* de l'exemple introductif peut être définie par :

```
let f_ex =
  let table_ex = Array.make 100 noeud_vide in
  table_ex.(0) <- {lettre='g'; poids=2; fg=(-1); fd=6};
  table_ex.(1) <- {lettre='a'; poids=7; fg=4; fd=3};
  table_ex.(2) <- {lettre='b'; poids=4; fg=(-1); fd=(-1)};
  table_ex.(3) <- {lettre='e'; poids=3; fg=5; fd=(-1)};
  table_ex.(4) <- {lettre='f'; poids=5; fg=(-1); fd=(-1)};
  table_ex.(5) <- {lettre='c'; poids=4; fg=(-1); fd=(-1)};
  table_ex.(6) <- {lettre='d'; poids=5; fg=(-1); fd=(-1)};
  {nb_arbres = 3; nb_noeuds = 7; table = table_ex};;
```

On supposera que la table est assez grande pour pouvoir contenir les nœuds qui seront ajoutés.

Remarques sur la syntaxe

1. Le poids du nœud d'indice k dans un arbre `a` est lu par `a.table.(0).poids`,
2. La modification d'un champ mutable se fait à l'aide du signe `<-` ; on pourra, par exemple, écrire `a.nb_arbres <- 2` pour indiquer que la forêt `a` contient dorénavant deux H_arbres.

2.1 Fonctions de base pour l'algorithme de Huffman

On considère une forêt contenant *nb_nœuds* nœuds et un entier k vérifiant $1 \leqslant k \leqslant nb_n\oe uds$; il s'agit d'écrire une fonction déterminant l'indice du nœud dont le poids est le plus petit parmi les k premiers nœuds de la table de la forêt, c'est-à-dire parmi les nœuds qui se trouvent entre les indices 0 et $k-1$ de cette table. S'il y a plusieurs nœuds de plus petit poids dans l'intervalle indiqué, la fonction renverra le plus petit indice de ceux-ci.

Question 8
Écrire une fonction `indice_du_min` *telle que si* `f` *est une valeur de type* `foret` *et* `k` *une valeur entière strictement positive ne dépassant pas* `f.nb_noeuds`*, alors* `indice_du_min f k` *renvoie l'indice recherché.*

On considère une forêt F constitué d'au moins deux H_arbres. Il s'agit de faire en sorte que, dans la table de F, les deux racines de plus petits poids soient les deux dernières racines dans la partie de la table consacrée aux racines de la forêt. Autrement dit, il s'agit d'examiner les racines de F (c'est-à-dire les nœuds qui se trouvent entre les indices 0 et $nb_arbres-1$ de la table) pour mettre, par des échanges, les deux racines de plus petits poids aux indices $nb_arbres-2$ et $nb_arbres-1$; plus précisément, on mettra la racine de plus petit poids à l'indice $nb_arbres-1$ et la racine d'avant-dernier plus petit poids à l'indice $nb_arbres-2$.

Par exemple, après ce traitement, la table décrivant *F_ex* devient :

Indice du nœud	0	1	2	3	4	5	6
Lettre	'a'	'b'	'g'	'e'	'f'	'c'	'd'
Poids	7	4	2	3	5	4	5
Fils gauche	4	-1	-1	5	-1	-1	-1
Fils droit	3	-1	6	-1	-1	-1	-1

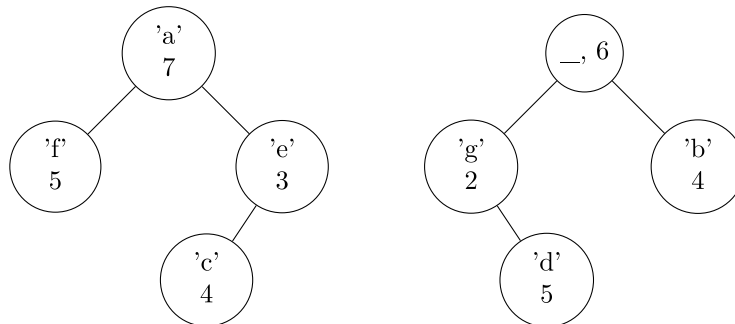
S'il y a des égalités entre les poids qui conduisent à plusieurs couples possibles de racines de plus petits poids, on choisira alors les deux racines de plus petits indices. On utilisera la fonction définie dans la question 8.

Question 9
Écrire une fonction `deux_plus_petits` *telle que si* `f` *est une valeur de type* `foret` *vérifiant* `f. nb_arbres >= 2`*, alors* `deux_plus_petits f` *transforme le champ* `table` *de* `f` *de façon à obtenir l'ordre cherché.*

On considère une forêt F possédant au moins deux H_arbres. On définit une transformation de F nommée assemblage de F. Soient r_1 et r_2 deux racines de plus petits poids ; on suppose que le poids de r_1 est inférieur ou égal à celui de r_2 . L'assemblage de F consiste à ajouter à F un nœud dont :

- le poids est la somme des poids des nœuds r_1 et r_2,
- le fils gauche est r_1,
- le fils droit est r_2.

Le nœud ajouté est donc la racine d'un H_arbre de la forêt obtenue par l'assemblage et le nombre total de H_arbres de la forêt a diminué de 1.
La lettre contenue par ce nouveau nœud n'a pas d'importance et n'est pas spécifiée. On ne cherche pas à ce que, après assemblage, les racines des H_arbres respectent l'ordre de la question 9.
Si on applique cette transformation à la forêt *F_ex*, celle-ci devient la forêt ci-dessous, dans laquelle on n'a pas précisé de valeur pour la lettre du nœud ajouté :



Pour cette forêt, on a $nb_arbres = 2$, $nb_nœuds = 8$ et la table peut être

Indice du nœud	0	1	2	3	4	5	6	7
Lettre	'a'	_	'g'	'e'	'f'	'c'	'd'	'b'
Poids	7	6	2	3	5	4	5	4
Fils gauche	4	2	-1	5	-1	-1	-1	-1
Fils droit	3	7	6	-1	-1	-1	-1	-1

Question 10
Écrire en une fonction assemblage telle que, si `f` *est une valeur de type* `foret` *vérifiant* `f.nb_arbres >= 2`*, alors* `assemblage f` *transforme* `f` *selon les indications fournies ci- dessus. On utilisera la fonction* `deux_plus_petits` *définie dans la question précédente.*

2.2 Propriétés pour l'algorithme de Huffman

Remarque : dans les illustrations de cette partie, lorsqu'un champ n'est pas précisé dans un nœud, cela signifie que sa valeur n'intervient pas.
La hauteur d'un nœud n d'un arbre, notée $h(n)$, est définie de la façon suivante :

- la hauteur de la racine vaut 0,
- la hauteur d'un nœud autre que la racine vaut un de plus que la hauteur de son père.

On dit, dans un arbre, qu'un nœud n est de **hauteur maximum** s'il n'existe pas de nœud de hauteur strictement plus grande que $h(n)$ dans cet arbre ; un nœud de hauteur maximum est nécessairement une feuille.
Deux feuilles d'un arbre sont dites **sœurs** si elles ont le même nœud pour père.
Étant donné un H_arbre A, on appelle **évaluation** de A la quantité $e(A) = \sum_{f \text{ feuille de } A} h(f).poids(f)$.
Pour l'arbre A_ex ci-dessous, on a $e(A_ex) = 1 \times 5 + 3 \times 4 + 3 \times 7 = 38$.

Question 11
Écrire une fonction `eval` *telle que, si* `f` *est une valeur de type* `foret`*, alors* `eval f` *calcule la somme des évaluations des arbres qui composent la forêt..*

On rappelle que les poids des nœuds sont des entiers strictement positifs.
On dit que deux H_arbres ont mêmes feuilles s'ils ont le même nombre de feuilles et que les contenus des feuilles de l'un sont les mêmes que les contenus des feuilles de l'autre.
On dira par exemple que les deux H_arbres A_ex et A'_ex ci-dessous ont mêmes feuilles.

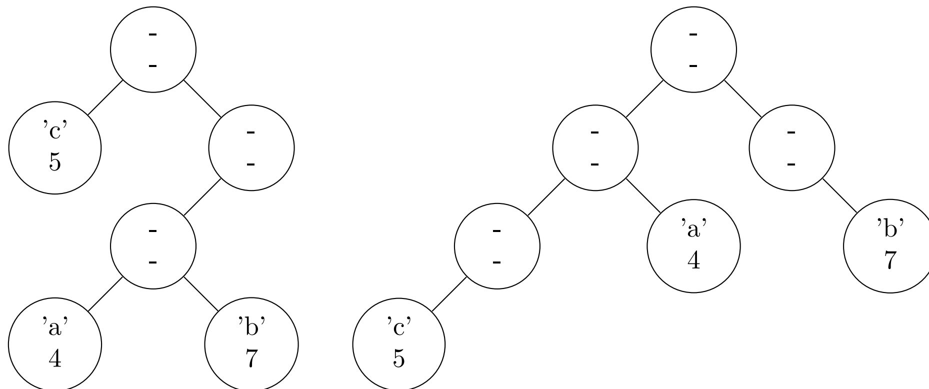


FIGURE I.2 – Deux arbres, A_ex et A'_ex

Question 12
Montrer que, si un H_arbre A est optimal, alors tout nœud interne de A admet deux fils. Après avoir montré ce résultat, transformer le H_arbre A_ex en un H_arbre B_ex de mêmes feuilles que A_ex et d'évaluation inférieure en s'appuyant sur l'argument de la preuve ; on indiquera pour cet exemple le gain d'évaluation obtenu.

Question 13
Soient f_1 et f_2 deux feuilles d'un H_arbre A optimal vérifiant la relation $h(f_1) > h(f_2)$; montrer qu'alors on a $poids(f_1) \leqslant poids(f_2)$. Après avoir montré ce résultat, transformer le H_arbre B_ex en un H_arbre C_ex de mêmes feuilles et d'évaluation inférieure à celle de C_ex ; on indiquera le gain d'évaluation obtenu.

Question 14
On considère un H_arbre A et deux feuilles f_1 et f_2 de A de plus petits poids. Montrer qu'il existe un H_arbre optimal de mêmes feuilles que A dans lequel f_1 et f_2 sont de hauteur maximum et sont sœurs.

On considère un H_arbre A et deux feuilles f_1 et f_2 qui sont sœurs dans A ; on note p_1 et p_2 les poids respectifs de ces feuilles. On note n le père (commun) de f_1 et f_2 et on considère la transformation suivante :

- *poids(n)* devient égal à $p_1 + p_2$,
- on supprime f_1 et f_2, n devient donc une feuille.

Le champ *lettre(n)* n'a pas d'importance et n'est donc pas spécifié.
On note B le H_arbre ainsi obtenu.
On dit qu'on a obtenu B en simplifiant A **par coupe** de f_1 et f_2 .

Question 15
Établir une relation entre $e(A)$, $e(B)$, p_1 et p_2 .

Question 16
On considère maintenant un H_arbre A et deux feuilles f_1 et f_2 qui sont sœurs dans A et de plus petits poids. On simplifie A par coupe de f_1 et f_2 et on obtient ainsi un H_arbre B. Montrer que A est optimal si et seulement si B est optimal.

2.3 Algorithme de Huffman

On appelle **chaîne binaire** une suite composée de 0 et de 1.
On considère dans toute cette partie un texte, c'est-à-dire une chaîne de caractères, T.
On souhaite coder ce texte avec une chaîne binaire. Pour cela, on décide d'associer une chaîne binaire à chaque caractère : son **mot de code** ; l'ensemble des mots de code est le **codage**. On peut alors coder le texte caractère par caractère en mettant bout à bout successivement les mots de code de tous les caractères de T ; on obtient alors le texte codé qui sera noté $C(T)$.
L'objectif est de choisir les mots de code pour que le décodage du texte soit possible et que la chaîne binaire $C(T)$ soit la plus courte possible.
Dans les exemples on utilisera des textes écrits uniquement avec les 6 lettres 'a', 'b', 'c', 'd', 'e' et 'f' dont on dira qu'elles forment un **alphabet réduit**.

Question 17
On considère un codage de l'alphabet réduit

caractère	*'a'*	*'b'*	*'c'*	*'d'*	*'e'*	*'f'*
mot de code	*00*	*101*	*11*	*0101*	*011*	*100*

On suppose que le texte codé $C(T)$ est 1000011011, déterminer le texte T.

On dit qu'un mot de code u est **préfixe** d'un autre mot de code v si la chaîne v commence par u ; par exemple, $u = 01$ est préfixe de $v = 0110$. On dit que le codage d'un alphabet est **préfixe** si aucun mot de code n'est préfixe d'un autre mot de code.

Question 18
Indiquer si le codage donné en exemple dans la question ci-dessus est ou non préfixe.
Dire pourquoi il est utile que le codage de l'alphabet soit préfixe.

On considérera par la suite un texte *T_ex* écrit dans l'alphabet réduit.
Les nombres d'occurrences des 6 caractères dans *T_ex* sont donnés par le tableau

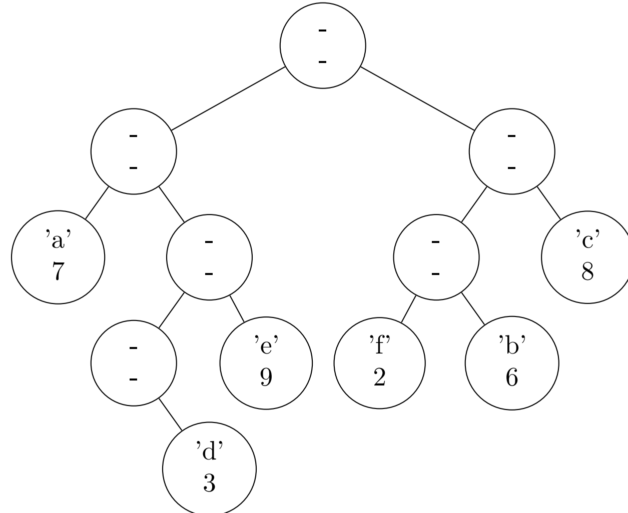
caractère	'a'	'b'	'c'	'd'	'e'	'f'
nombre d'occurrences	7	6	8	3	9	2

On représente un codage préfixe l'alphabet réduit par un H_arbre dont les feuilles contiennent les caractères de l'alphabet comme champ lettre et le nombre d'occurrences de ce caractère dans le texte T comme champ poids.
Les champs lettre et poids des nœuds internes n'ont pas d'importance.
Ce H_arbre est construit de sorte que le mot de code d'un caractère c corresponde au chemin de la racine du H_arbre à la feuille contenant c.

Plus précisément, pour retrouver un caractère connaissant son mot de code,

- on part de la racine du H_arbre,
- on lit le mot de code et, au fur et à mesure, on descend dans le H_arbre : lorsqu'on rencontre un 0, on descend à gauche, lorsqu'on rencontre un 1, on descend à droite.

Le H_arbre associé au codage de la question 17 est représenté par l'arbre ci-contre. Par exemple, pour aller de la racine au nœud correspondant au mot de code 011 on descend une fois à gauche puis deux fois à droite pour trouver la lettre 'e'.



Question 19
Indiquer une relation entre l'évaluation du H_arbre représentant un codage préfixe de l'alphabet et la longueur du texte codé $C(T)$ si T est codé en utilisant ce codage préfixe.

On appelle **codage optimal** pour T un codage préfixe minimisant la longueur de $C(T)$.
Pour chercher un codage optimal, on cherche un H_arbre d'évaluation minimum et dont les feuilles correspondent aux caractères munis de leurs nombres d'occurrences.
Pour cela, on utilise l'**algorithme de Huffman**.
Cet algorithme est initialisé avec une forêt F de H_arbres tous réduits à un nœud et contenant chacun, comme lettre, un caractère et, comme poids, le nombre d'occurrences dans T de ce caractère. Tous les caractères figurent une fois et une seule parmi ces nœuds ; le nombre de nœuds de cette forêt initiale F est donc égal au nombre de H_arbres et encore égal au nombre de caractères.
On applique ensuite à F la boucle suivante :

```
tant que F possède au moins deux H_arbres, faire assemblage(F)
```

ce qui termine l'algorithme de Huffman.

Question 20
Prouver que, lorsque l'algorithme décrit ci-dessus est terminé, l'unique H_arbre de la forêt correspond à un codage optimal pour T.

Question 21
Déterminer, en utilisant l'algorithme de Huffman, un codage optimal pour T_ex.
On dessinera le H_arbre obtenu par l'algorithme et on exploitera ce H_arbre pour donner explicitement les mots de code des 6 caractères.

3 Solutions

Solution de la question 1 - Il suffit de vérifier que la borne supérieure d'un intervalle est strictement inférieure à la borne inférieure de l'autre intervalle.

```
let disjoints (min1, max1) (min2, max2) =
  max1 < min2 || max2 < min2;;
```

Solution de la question 2 -

```
let fusion (min1, max1) (min2, max2) =
  (min min1 min2), (max max1 max2);;
```

Solution de la question 3 -

```
let rec suite arbre =
  match arbre with
  |Vide -> []
  |Feuille(a, b) = [(a, b)]
  |Noeud(g, _, d) -> (suite g) @ (suite d);;
```

Solution de la question 4 -

```
let rec englobant a =
  match a with
  |Vide -> failwith "Erreur arbre vide"
  |Feuille(min, max) -> min, max
  |Noeud(g, n, d) -> fusion (englobant fg) (englobant fd);;
```

Solution de la question 5 - On écrit une fonction auxiliaire récursive qui retourne l'intervalle englobant en même temps que la réponse booléenne.

```
let verifie a =
  let rec aux a =
    match a with
    |Vide -> true, 0, 0
    |Feuille(min, max) -> true, min, max
    |Noeud(g, n, d) -> let rep1, min1, max1 = aux g in
                       let rep2, min2, max2 = aux d in
                       let rep = rep1 && rep2 && (max1 = k) &&
                            (k < min2) in
                       rep, min1, max2 in
  let rep, _, _ = aux a in rep;;
```

Solution de la question 6 -

```
let rec appartient k a =
  match a with
  |Vide -> false
  |Feuille(min, max) -> (min <= k) && (k <= max)
  |Noeud(g, n, d) when n < k -> appartient k d
  |Noeud(g, n, d) -> appartient k g;;
```

Solution de la question 7 - Dans le cas simple de l'énoncé, comme la valeur d'un nœud appartient à l'un des intervalles du fils gauche, on n'aura jamais cette valeur dans l'intervalle à ajouter. On ne pourra donc rencontrer que le cas $n < min$ ou $max < n$. De même quand on arrive à une feuille, comme les intervalles sont disjoints il y en aura un à droite de l'autre.

```
let rec ajouter (min, max) a =
  match a with
  |Vide -> Feuille( min, max)
  |Feuille(min1, max1) when max1 < min ->
          Noeud(a, max1, Feuille(min, max))
  |Feuille(min1, max1) -> Noeud( Feuille(min, max), max, a)
  |Noeud(g, n, d) when n < min -> Noeud(g, n, ajouter (min,
     max) d)
  |Noeud(g, n, d) -> Noeud(ajouter (min, max) g, n, d);;
```

Solution de la question 8 -

```
let indice_du_min foret k =
  let mini = ref foret.table.(0).poids in
  let ind_mini = ref 0 in
  for i = 1 to k-1 do
    if foret.table.(i).poids < !mini
    then (ind_mini := i;
          mini := foret.table.(i).poids) done;
  !ind_mini;;
```

Solution de la question 9 - Je note k la valeur de *nb_arbres*. On opère en deux temps,

1. on cherche le minimum parmi les n premiers arbres,
2. on le place en position n,
3. on cherche le minimum parmi les $n-1$ premiers arbres,
4. on le place en position $n-1$.

On commence par une fonction d'échange :

```
let echange i j foret =
  let noeud = foret.table.(i) in
  foret.table.(i) <- foret.table.(j);
  foret.table.(j) <- noeud;;
```

On peut alors traduire l'algorithme.

```
let deux_plus_petits foret =
  let k = foret.nb_arbres in
  if k >= 2
  then begin let i = indice_du_min foret k
             in echange i (k-1) foret;
             let j = indice_du_min foret (k-1)
             in echange j (k-2) foret end;;
```

Solution de la question 10 - Le nouvel arbre devra être à la $k-1$-ième position, celle occupée par l'avant-dernière racine.

On note k le nombre d'arbres et n le nombre de nœuds :

- on place les deux arbres de poids minimal à la fin des arbres (question ci-dessus)
- on copie le nœud d'indice $k-2$ (l'avant dernier arbre) à la position n,
- on place à la position $k-2$ un nouveau nœud, sa somme est la somme des deux poids, son fils gauche a l'indice $k-1$ et son fils droit a l'indice n,
- on augmente n de 1, on ajoute un nouvel nœud
- on diminue k de 1, le nouveau nœud remplace deux arbres.

```
let assemblage f =
  let n = f.nb_noeuds in
  let k = f.nb_arbres in
  if k >= 2
  then begin
    deux_plus_petits f;
    f.table.(n) <- f.table.(k-2);
    f.table.(k-2) <-
      {lettre = ‘\000‘; fg = (k-1); fd = n;
       poids = f.table.(n).poids + f.table.(k-1).poids};
    f.nb_noeuds <- (n+1);
    f.nb_arbres <- (k-1)
  end;;
```

Solution de la question 11 -

On calcule l'évaluation d'un arbre en calculant la somme des évaluations de ses fils. On a besoin d'envoyer un paramètre hauteur pour calculer l'évaluation des feuilles.

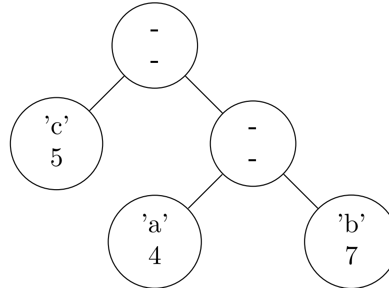
```
let eval f =
  let rec eval_arbre arbre haut =
    match arbre.fg, arbre.fd with
    |-1, -1 -> arbre.poids*h
    |-1, k -> eval_arbre f.(k) (haut + 1)
    |k, -1 -> eval_arbre f.(k) (haut + 1)
    |k, l -> eval_arbre f.(k) (haut + 1) + eval_arbre f.(l) (
       haut + 1)
  in let e = ref 0 in
     for i = 0 to (nb_arbres -1)
       do e := !e +eval_arbre f.(i) 0 done;
     !e;;
```

Solution de la question 12 - Si a est un arbre dont un nœud interne n n'a un seul fils, par exemple le fils gauche g, on remplace le nœud n par son fils gauche g. L'arbre obtenu, noté a', a les mêmes feuilles mais les feuilles qui figuraient dans g ont maintenant une hauteur diminuée de 1. Ainsi, si p est la somme des poids des feuilles dans g, on a $e(a') = e(a) - p$.

Les poids sont strictement positifs et tout nœud interne contient au moins une feuille donc $p > 0$. On obtient donc $e(a') < e(a)$ et a' contient les mêmes feuilles que a donc a n'est pas optimal.

Par contraposition dans un arbre optimal tout nœud interne a exactement deux fils.

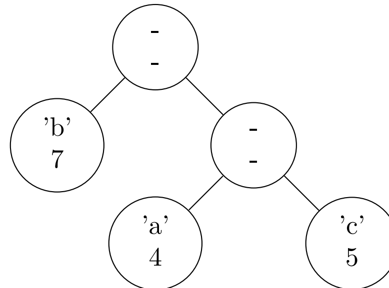
Cette procédure appliquée à A_ex donne l'arbre B_ex suivant.



L'évaluation de B_ex vaut $1.5 + 2.4 + 2.7 = 27 = 38 - 11$.

Solution de la question 13 - On considère deux feuilles, f_1 et f_2, d'un arbre optimal a telles que $h(f_1) = h_1 > h_2 = h(f_2)$.
Si on avait poids$(f_1) = p_1 > p_2 =$ poids(f_2) on pourrait construire un arbre a' égal à a à l'exception de l'échange des feuilles f_1 et f_2.
L'évaluation due à la feuille f_1 passerait de h_1p_1 à h_2p_1 et celle de f_1 passerait de h_2p_2 à h_1p_2 d'où $e(a') = e(a) - h_1p_1 + h_2p_1 - h_2p_2 + h_1p_2 = e(a) + (p_1 - p_2)(h_2 - h_1) < e(a)$ ce qui est impossible en raison du caractère optimal de a. On doit donc avoir poids$(f_1) \leqslant$ poids(f_2).

On peut appliquer cette méthode à B_ex pour obtenir C_ex d'évaluation 25 en échangeant les feuilles 'c' et 'b'.



Solution de la question 14 -

- on suppose que f_1 est de poids minimal. Si f_1 n'est pas à la hauteur maximale, pour toute feuille f de hauteur maximale on a $h(f) > h(f_1)$ donc, d'après la question précédente, $poids(f) \leqslant poids(f_1)$. On en déduit , en raison de la minimalité que $poids(f) = poids(f_1)$.
  On peut alors échanger les feuille f et f_1 sans changer l'évaluation.
  Il existe un arbre optimal A_1, de mêmes feuilles que A, dans lequel f_1 est de hauteur maximale.
- On a vu que, dans un arbre optimal, tout nœud interne admet deux fils.
  Une feuille de hauteur maximale admet un père qui est un nœud interne donc admet donc deux fils. Le second fils est aussi une feuille car, sinon, il y aurait des feuille à une hauteur supérieure, ce qui est impossible.
  On considère la feuille f', sœur de f_1 dans A_1.
  On a $poids(f') \geqslant poids(f_2)$ car f_1 a le deuxième poids minimal et $f' \neq f_1$.
- Si $poids(f') = poids(f_2)$ alors on peut échanger f' et f_2 sans changer l'évaluation : on obtient un arbre optimal de mêmes feuilles que A_1 donc de même feuilles que A dans lequel f_1 et f_2 sont de hauteur maximum et sœurs.
- Si $poids(f') > poids(f_2)$ f_2 est aussi à la hauteur maximale car sinon on aurait $h(f') > h(f_2)$ et la question précédente prouverait qu'alors on a $poids(f') \leqslant poids(f_2)$, ce qui est exclu. f' et f_2 sont donc à la hauteur maximale et on peut échanger f' et f_2 sans changer l'évaluation : on obtient un arbre optimal de mêmes feuilles que A_1 donc de même feuilles que A dans lequel f_1 et f_2 sont de hauteur maximum et sœurs.

Solution de la question 15 - Avec les notations de l'énoncé on a $h(f_1) = h(f_2) = h(n) + 1$ donc $e(B) = e(A) - p_1h(f_1) - p_2h(f_2) + (p_1 + p_2)h(n) = e(A) - (p_1 + p_2)$

Solution de la question 16 - On garde les notations de la question précédente.

- On suppose que A est optimal.
  Soit B' un arbre de mêmes feuilles que B. On peut remplacer la feuille de poids $p_1 + p_2$ par un nœud interne qui admet deux fils dont les feuilles ont les poids p_1 et p_2 par la construction inverse de celle ci-dessus. On obtint un arbre A' qui a les mêmes feuilles que A donc $e(A') \geqslant e(A)$ car A est optimal. De plus, comme ci-dessus, $e(B') = e(A') - (p_1 + p_2)$
  On a donc $e(B') = e(A') - (p_1 + p_2) \geqslant e(A) - (p_1 + p_2) = e(B)$ pour tout arbre B' de même feuilles que B donc B est optimal.
- On suppose que B est optimal.
  On choisit un arbre optimal A' de même feuilles que A.
  D'après la question 14 on peut supposer que f_1 et f_2 sont sœurs dans A'. En regroupant f_1 et f_2 on peut associer à A' un arbre B' qui admet les mêmes feuilles que B.
  On a $e(A') \leqslant e(A)$ car A" est optimal et, en raison de l'optimalité de B,
  $e(A') = e(B') + (p_1 + p_2) \geqslant e(B) + (p_1 + p_2) = e(A)$ donc $e(A') = e(A)$.
  Ainsi A est optimal.

Solution de la question 17 -
Le code de la première lettre commence par un 1 mais 1 n'est pas un code.
Le code de la première lettre commence donc par 10, qui n'est pas un code.
Ainsi le code de la première lettre commence par 100 qui est le code de 'f'.
Comme aucun autre code commence par 100 la première lettre est 'f'.

On peut poursuivre et on trouve le mot 'face'.

Solution de la question 18 -
On vérifie qu'aucun code n'est le préfixe d'un autre : le codage est préfixe.

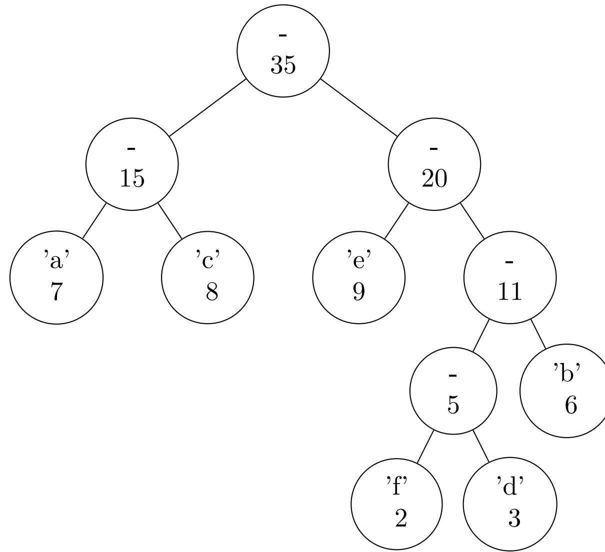
Si le codage n'est pas préfixe alors un même code pourrait signifier plusieurs mots.
Par exemple si 1 code 'a' et 11 code 'b' alors 111 pourrait coder 'aaa', 'ab' ou 'ba'.

Solution de la question 19 -
Chaque caractère est codé par un code de longueur égale à la profondeur de la feuille correspondante. Comme le nombre d'occurrences du caractère dans le texte est le poids de la feuille la longueur totale du texte codé est donc égale à l'évaluation de l'arbre.

Solution de la question 20 - On prouve que l'arbre obtenu par des applications successives de la fonction assemblage est optimal par récurrence sur le nombre n de caractères de l'alphabet.

- Pour $n = 1$ il n'y a rien à prouver car il n'existe qu'un seul arbre avec une seule feuille ; il est donc optimal.
- On suppose que l'algorithme de Huffman conduit à un arbre optimal pour tout ensemble de $n - 1$ caractères avec $n \geqslant 2$.
  On considère un ensemble de n caractères dans lequel c_1 et c_2 sont deux caractères de poids minimum notés p_1 et p_2 respectivement.
  L'algorithme de Huffman commence par regrouper les deux feuilles f_1 et f_2 correspondant à ces caractères. Si on effectue la coupe de f_1 et f_2, on obtient une feuille. On est alors ramené au cas de $n - 1$ caractères.
  D'après l'hypothèse de récurrence, l'algorithme de Huffman va construire un arbre optimal B à partir de ces $n - 1$ caractères.
  Cet arbre correspond à l'arbre obtenu par la méthode de Huffman à partir des n caractères initiaux, A, auquel on applique la coupe de f_1 et f_2 qui sont sœurs de plus petit poids.
  D'après la question 16 on déduit de l'optimalité de B que A est optimal.

Solution de la question 21 - On applique 5 fois l'algorithme `assemblage` à la forêt des feuille simples.



$C(T)$ vaut 86.
La table de codage est

'a'	'b'	'c'	'd'	'e'	'f'
00	111	01	1101	10	1100