CORRIGE DE L'EPREUVE D'INFORMATIQUE ESIM MP 97

I - L'arbre de Huffman

1)On parcourt l'arbre de Huffman selon l'algorithme suivant :

```
ALGO decode(texte_code) → texte décodé
DEBUT
   texte_decode ← mot vide
   TANT QUE texte_code ≠ ∅
   BOUCLER
      se placer à la racine de l'arbre de Huffman
      TANT QUE on n'est pas sur une feuille
      BOUCLER
         SI premier_chiffre(texte_code) = 0 ALORS parcourir la branche gauche
         SINON parcourir la branche droite
         FIN SI
      FIN BOUCLE
      écrire le caractère contenu dans la feuille à la suite de texte_decode
   FIN BOUCLE
   texte_decode
FIN
```

Avec le message $1111100101111110 1$, on obtient les caractères $(11111)(00)(101)(11111)(01) \rightarrow$ HACHE

2)Supposons que tous les caractères aient la même fréquence d'appartition, et montrons par récurrence sur n la propriété suivante : si $2^{n-1} < nbcar \leq 2^k$, alors la profondeur de l'arbre de Huffman est n.

- Pour $n = 0$, on a $nbcar = 1$, et l'arbre de Huffman ne contient qu'un noeud, donc sa profondeur est bien nulle.
- Soit $n \in N$. Supposons la propriété vraie pour tous les entiers inférieurs ou égaux à n. Alors tous les arbres de Huffman intermédiaires construits jusqu'à ce qu'on accède à l'arbre-racine n° $1+2^n$ d'une liste contenant au départ $2^n < nbcar \leq 2^{k+1}$ arbres-racines ont un poids strictement supérieur à celui de chaque arbre-racine, donc se trouvent systématiquement placés en fin de liste. Il en résulte que lorsqu'on accède à l'arbre-racine n° $1+2^n$, la liste est dans l'état suivant :

$lst \rightarrow \left(x_{1+2^n}; f\right) \rightarrow \ldots \rightarrow \left(x_{nbcar}; f\right) \rightarrow h$, où f désigne la fréquence commune à tous les caractères, et $h = \left(x_1 .. x_{2^n}; 2^n f\right)$ est l'arbre de Huffman formé à partir de 2^n premiers arbres-racines. Par hypothèse de récurrence, h a une profondeur égale à n. De plus, le poids total des $nbcar - 2^n \leq 2^n$ arbres-racines restant à gauche de h étant inférieur ou égal à celui de h, tous les arbres de Huffman intermédiaires construits en accédant à ces arbres-racines ne modifieront pas le statut de h en fin de liste. Et comme il y en a entre 0 (au sens strict) et 2^n (au sens large), l'hypothèse de récurrence forte entraîne que l'arbre de Huffman h' construit à partir de ces arbres-racines sera de profondeur $n' \leq n$.

La situation pré-finale sera donc $lst \rightarrow h' \rightarrow h$ (si $nbcar < 2^{n+1}$) ou $lst \rightarrow h \rightarrow h'$ (si $nbcar = 2^{n+1}$). L'arbre final aura donc une profondeur de $1 + \max(n', n) = 1 + n$, ce qui achève la récurrence. Or $n = E\left(\log_2 nbcar\right) + 1$, d'où :

la profondeur d'un arbre de Huffman de taille $nbcar$ est $E\left(\log_2 nbcar\right) + 1$

3)
```
let initialisations () =
  let texte = !txt
  in
  for k = 0 to string_length texte - 1
  do
    let tk = texte.[k]
    in
    try
      for j = 0 to !nbcar - 1
      do if tk = car.(j) then (frequence.(j) <- frequence.(j) + 1; raise Exit) done;
      incr nbcar;
      car.(!nbcar - 1) <- tk;
      frequence.(!nbcar - 1) <- 1
    with Exit -> ()
  done;;
```

4)Version récursive

```
let rec appartient c = function
   "" -> false
  |ch -> if nth_char ch 0 = c then true
         else appartient c (sub_string ch 1 (string_length ch - 1));;
```

Version itérative

```
let appartient c ch =
  try
    for k = 0 to string_length ch - 1
    do if nth_char ch k = c then raise Exit done;
    false (* cas où toute la chaîne est parcourue sans rencontrer c *)
  with Exit -> true
       (* le parcours de la chaîne a déclenché l'exception Exit, donc c a été trouvé
*);;
```

5)Au vu de l'exemple proposé dans l'énoncé, lorsque l'un des arbres de la liste a le même poids que celui à insérer, l'insertion s'effectue après tous les arbres ayant ce poids dans la liste. De plus, la liste semble ne devoir jamais comporter d'arbre vide : la fonction récursive `insère` ne prévoira donc pas le cas de vacuité, ce qui entraînera un message d'avertissement de la part de Caml parce-que la reconnaissance par motif n'est pas exhaustive :

```
let rec insere a liste =
  match liste with
     [] -> [a]
    |t::q -> match (a,t) with
               (noeud (g,x,d),noeud (g',x',d')) -> if x.poids < x'.poids then
a::liste
                                                   else t::(insere a q);;
```

6)La remarque précédente est confirmée ! La création de la liste demandée consiste donc en l'algorithme suivant :

ALGO **`creer_liste`**() → liste ordonnée d'arbres-racines
DEBUT
  liste ← []
  POUR *k* DE 0 A (nombre pointé par `nbcar`) – 1
  BOUCLER
    paire ← {`poids` = cellule n° *k* de `frequence` ; `chaine` = chaîne formée du caractère de la cellule n° *k* de `car`} *arbre_racine* ← `noeud` (`nil`,*poids*,`nil`)
    liste ← **`insère`**(*arbre_racine*,*liste*)
  FIN BOUCLE
  pointeur sur *liste*
FIN

```
let creer_liste () =
  for k = 0 to !nbcar - 1
  do
    liste := insere
             (noeud (nil,{poids = frequence.(k) ; chaine = char_for_read
car.(k)},nil))
             !liste
  done;
  liste
  where liste = ref [];;
```

7)A nouveau, on ne se préoccupera pas des arbres vides, puisque **`fusion`** ne sera pas amené à en manipuler :

```
let fusion a1 a2 =
  match (a1,a2) with
    (noeud (_,x1,_),noeud (_,x2,_)) ->
       noeud (a1,{poids = x1.poids + x2.poids ; chaine = x1.chaine^x2.chaine},a2);;
```

8)L'algorithme consiste simplement à faire la fusion des deux arbres de tête par **`fusion`**, puis à insérer le résultat dans la queue à l'aide de **`insere`**. L'algorithme s'arrête lorsqu'il ne reste plus qu'un sel arbre dans la liste, et c'est alors l'arbre de Huffman. Un algorithme sur les listes étant naturellement récursif, on obtient :

```
let creer_arbre liste =
  let rec f l = match l with [a] -> a |a1::a2::q -> f (insere (fusion a1 a2) q)
  in f !liste;;
```

Remarquons que le fait d'imposer d'utiliser une référence sur liste au lieu d'une liste est une maladresse du concepteur du sujet, prouvant que le sujet a probablement été d'abord « pensé » en Pascal...

9)L'algorithme **codage** est un algorithme essentiellement itératif, consistant à traiter séquentiellement tous les caractères de la chaîne `ch` à l'aide du sous-algorithme récursif suivant : on détermine grâce à **`appartient`** lequel des fils de l'arbre de Huffman courant contient le caractère à coder, et on appelle le sous-algorithme sur ce fils. La récursion s'arrête lorsque l'arbre est une feuille. Au cours des appels récursifs du sous-algorithme, on tient à jour une liste contenant des 0 ou des 1 selon la direction qu'on prend dans le parcours de l'arbre, et il suffit d'en juxtaposer les cellules à la fin de l'exécution du sous-algorithme pour récupérer la chaîne de 0 et de 1 formant le code de Huffman du caractère. On peut d'ailleurs immédiatement les juxtaposer à la chaîne déjà construite par concaténation des résultats du traitement des caractères précédents :

```
let codage ch =
  let rec petit_poucet c a =
    match a with
      (* les deux fils d'un arbre de Huffman sont simultanément vides ou non vides *)
       noeud (nil,_,_) -> ""
      |noeud (a1,_,a2) -> match a1 with
                            noeud (_,paire,_) when appartient c paire.chaine ->
                                 (string_of_int 0)^(petit_poucet c a1)
                           |_ -> (string_of_int 1)^(petit_poucet c a2)
  in
  let ch_codé = ref ""
  in
  for k = 0 to string_length ch - 1 do ch_codé := !ch_codé^(petit_poucet ch.[k] !huff)
done;
  !ch_codé;;
```

10)L'algorithme **decodage** consiste à traiter de manière itérative la chaîne `ch` donnée en argument de la manière suivante : on utilise un sous-algorithme récursif, prenant en argument une chaîne et un arbre, et s'appelant sur le reste de la chaîne et le fils gauche (respectivement droit) de l'arbre selon que le premier caractère de la chaîne est un 0 (respectivement un 1). Le sous-algorithme s'arrête lorsqu'on parvient à une feuille, et il ne reste plus qu'à récupérer (sous forme de chaîne) le caractère stocké dans cette feuille. Ce sous-algorithme est appelé initialement avec une certaine chaîne et l'arbre `!huff`, et retourne une paire contenant d'une part la chaîne représentant le caractère codé correspondant à la partie traitée de la chaîne donnée en argumentet, et d'autre part le reste (non traité) de cette chaîne.
Il suffit alors d'appeler itérativement ce sous-algorithme, d'abord sur la chaîne `ch`, puis sur les restes successifs retournés dans la deuxième partie de la paire retournée par le sous-algorithme, et ce jusqu'à ce que le reste soit vide, tout en concaténant au fur et à mesure les caractères décodés, qui se trouvent dans la première partie de cette paire :

```
let decodage ch =
  let rec jeu_de_piste ch' a =
    match a with
       noeud (nil,paire,_) -> (paire.chaine,ch')
      |noeud (a1,_,a2) ->
        if ch'.[0] = `0` then jeu_de_piste (sub_string ch' 1 (string_length ch' - 1))
a1
        else jeu_de_piste (sub_string ch' 1 (string_length ch' - 1)) a2
  in
  let ch_codé = ref ch and ch_décodé = ref ""
  in
  while !ch_codé <> ""
  do
    let (s,s') = jeu_de_piste !ch_codé !huff
    in
    ch_décodé := !ch_décodé^s;
    ch_codé := s'
  done;
  !ch_décodé;;
```

Exemple 1 : un exemple d'emploi de l'algorithme de Huffman

```
#txt := "Bonjour à tous les élèves de MP de partout et d'ailleurs !";
initialisations ();
huff := creer_arbre (creer_liste ());;
```

```
- : unit = ()

#for k = 0 to !nbcar - 1 do print_char car.(k); print_char ` ` done;
for k = 0 to !nbcar - 1 do print_int frequence.(k); print_char ` ` done;
print_newline ();;
B o n j u r   à t s l e é è v d M P p a ' i !
1 4 1 1 4 3 11 1 4 4 4 6 1 1 1 3 1 1 1 2 1 1 1
- : unit = ()

#let s = codage !txt
in print_string s; print_newline (); print_string (decodage s);;
00100001000000111000100110011010100001011010001001101100010111111101100010101001111101
01110101011101100010101111110010100110101101010101111110010101100100101011110100010011
01101010111011010101011111001001001011001111111111111111000110011110001010011 10
Bonjour à tous les élèves de MP de partout et d'ailleurs !- : unit = ()
```

11)L'algorithme de recherche dichotomique dans un tableau trié (par ordre croissant) consiste à comparer l'élément cherché à l'élément médian du tableau (ou l'un des deux éléments médians si le nombre de cellules du tableau est pair). Si les deux éléments sont égaux, la recherche est terminée ; si l'élément cherché est strictement inférieur (resp. supérieur) à celui contenu dans la cellule médiane, on relance la recherche sur le sous-tableau composé des cellules situées strictement à gauche (resp. à droite) de la cellule médiane.
Dans le contexte de l'algorithme de Huffman, on sait que l'élément cherché est présent dans le tableau, donc l'algorithme s'arrête lorsque cet élément est trouvé ; mais comme on demande de programmer l'algorithme de rcherche dichotomique dans le cas général, nous allons prévoir un message d'erreur lorsque le sous-tableau sur lequel on doit relancer la recherche est vide.
Plutôt que de gérer des sous-tableaux, et de générer des recopies inutiles, nous allons travailler sur le tableau en place, en représentant un sous-tableau par une plage de cellules, délimitée par une borne gauche g et une borne droite d, initialement égales respectivement à 0 et à l'indice maximal des cellules du tableau :

```
let recherche c t =
  (* au début, le champ de recherche est constitué de toutes les cellules du tableau
*)
  let g = ref 0 and d = ref (vect_length t - 1)
  (* on définit une référence prête à accueillir le rang de l'élément cherché dans t
*)
  and en_attente = ref 0
  in
  try
    while !g <= !d (* condition d'arrêt *)
    do
      let milieu = (!g + !d) / 2
      in
      if t.(milieu) = c then (en_attente := milieu; raise Exit); (* c est trouvé! *)
      (* sinon on délimite la partie du tableau sur laquelle poursuivre la recherche
*)
      if t.(milieu) < c then g := milieu + 1 else d := milieu - 1;
    done;
    raise Not_found
  with Exit -> !en_attente;;
```

12)Soit n la longueur du tableau, et supposons $n \geq 4$.
L'algorithme est essentiellement constitué d'une boucle qui, dans le pire des cas (élément trouvé lorsque le sous-tableau à examiner n'a qu'une cellule, ou élément absent du tableau) s'arrête lorsque `!g > !d`. Or la suite des différences δ_k formée des différences (`!d - !g`) au tour de boucle n° k vérifie la relation de récurrence $\delta_{k+1} = E\left(\frac{\delta_k}{2}\right) - 1$, avec $\delta_0 = n-1$ (valeur avant l'entrée dans la boucle). Donc elle forme une suite strictement décroissante d'entiers, de premier terme strictement positif, et par conséquent cette suite devient strictement négative à partir d'un certain rang, ce qui assure que le programme finit bien. Même lorsque le programme est interrompu par la rencontre fortuite de x dans une cellule médiane, il comporte donc au plus p tours de boucle, chacun de complexité constante, où p est défini comme le premier entier k tel que $\delta_k < 0$.

Or $\delta_k \leq \frac{\delta_{k-1}}{2} - 1$. En posant $u_k = \delta_k - 2$, on a $u_k \leq \frac{u_{k-1}}{2} \leq \frac{n-3}{2^k}$. Donc $\delta_k < 0 \Leftrightarrow u_k < 2$ se produit dès que $k \geq \log_2(n-3) - 1$. Il en résulte que $p \leq \log_2(n-3) \leq \frac{\ln n}{\ln 2}$.

Soit $T(n)$ la complexité de **recherche** dans le pire des cas. On a donc $T(n) \leq \frac{\ln n}{\ln 2}\Theta(1)$, soit $T(n) = O(\ln n)$.

Conclusion
La recherche dichotomique sur un tableau est un algorithme de complexité logarithmique dans le pire des cas.

13)Nous allons réaliser un parcours récursif préfixe de l'arbre de Huffman :
on examine d'abord la racine, pour voir si la chaîne qui l'étiquette est composé d'un unique caractère :

- si tel est le cas, on est sur une feuille, et la variable globale `code` pointe vers le code de Huffman du caractère. Il ne reste plus alors qu'à chercher, grâce à **`recherche`**, le rang de ce caractère dans le tableau `car`, qu'on suppose classé, puis à placer la chaîne référencée par `code` dans la cellule de rang correspondant du tableau `huff_code` (ici, il y a une petite difficulté technique due à une maladresse de l'énoncé qui initialise `car` à l'aide de caractères `` ` ``, ce qui oblige par précaution à ne considérer que les cellules « significatives » de `car`, c'est-à-dire les `!nbcar` premières ; il vaudrait mieux en fait définir un nouveau tableau de la bonne longueur au moment où l'on trie les éléments de `car`...). Enfin, on élimine le dernier chiffre dans `!code`, pour mettre cette variable globale à jour pour la suite du parcours.
- sinon, on parcourt récursivement les branches gauche puis droite de l'arbre, non sans avoir au préalable ajouté le chiffre correspondant dans `!code` : 0 pour la branche gauche, 1 pour la branche droite. On doit bien entendu réaliser l'opération inverse à la sortie...

```
let rec parcours = function
   noeud (_,p,_) when string_length p.chaine = 1 ->
     huff_code.(recherche (nth_char p.chaine 0) car) <- !code
  |noeud (g,_,d) ->
     code := !code^"0";
     parcours g;
     code := sub_string !code 0 (string_length !code - 1);
     code := !code^"1";
     parcours d;
     code := sub_string !code 0 (string_length !code - 1);;
```

II - Code de SHANNON-FANO

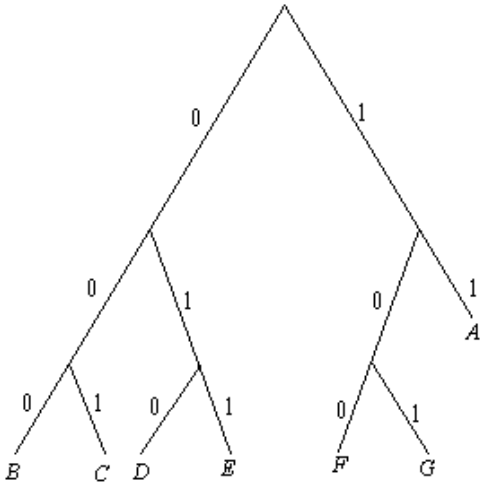
14)L'algorithme `shannon_fano` comporte principalement un sous-algorithme récursif `ajoute_bit`, dont les arguments sont les deux indices d et f des cellules extrêmes de la plage de cellules G_{d_f} du tableau `frequence` qui doit être exploitée.
Ce sous-algorithme en appelle lui-même un autre, `cherche_k`, dont la fonction est de chercher un indice k correspondant à la définition de l'énoncé, c'est-à-dire coupant la plage G_{d_f} en deux sous-plages G_{d_k} et G_{k+1_f} telles que $\left|S_{d\_k} - S_{k+1\_f}\right|$ soit minimum. Il n'y a a pas forcément unicité de k, mais il y a au plus deux valeurs possibles, et elles sont alors consécutives (exemple : `frequence = [|1 ;1 ;1|]` $\rightarrow k = 0$ ou $k = 1$) : la stratégie choisie a été de privilégier la plus grande des deux valeurs.
Une fois k déterminé, on concatène le bit 0 (en fait la chaîne `"0"`) derrière toutes les chaînes contenues dans les cellules d'indices k à d du tableau `sf_code` et le bit 1 derrière toutes les chaînes contenues dans les cellules d'indice k à d de ce tableau.
Enfin, on appelle récursivement `ajoute_bit` sur chacune des deux sous-plages.
Le sous-algorithme `ajoute_bit` s'arrête lorsque la plage à traiter est monocellulaire ($d = f$), tandis qu'i est appelé initialement avec les valeurs indiciales extrêmes du tableau `frequence`. Quant à `sf_code`, il est initialisé avec des chaînes vides.
Pour compléter cette explication, détaillons encore l'algorithme `cherche_k d f` : il calcule d'abord la somme des contenus des cellules de la plage G_{d_f}, puis remplace dans cette somme tous les signes + par des – en partant de la gauche (en fait, ceci est simulé avec la méthode $a+b \rightarrow a+b-2b = a-b$) jusquà obtenir un résultat négatif ou nul, auquel cas on compare les valeurs absolues de ce résultat avec le dernier résultat positif précédent pour choisir k :

```
let shannon_fano () =
  let cherche_k d f =
    let s = ref 0 and i = ref (d - 1)
    in
    for j = d to f do s := !s + frequence.(j) done;
    while !s > 0 do incr i; s := !s - 2 * frequence.(!i) done;
    if !s + frequence.(!i) >= 0 then !i else !i - 1
  in
  let rec ajoute_bit d f =
    if d = f then ()
    else
      begin
        let k = cherche_k d f
        in
        for j = d to k do sf_code.(j) <- sf_code.(j)^"0" done;
        for j = k + 1 to f do sf_code.(j) <- sf_code.(j)^"1" done;
```

```
        ajoute_bit d k;
        ajoute_bit (k + 1) f
      end
  in ajoute_bit 0 (!nbcar - 1);;
```

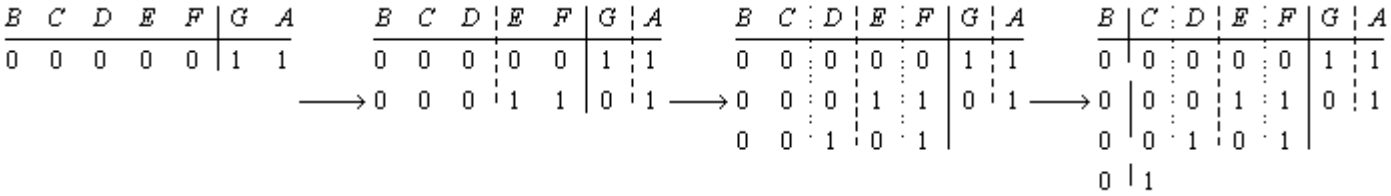
15)

Si le texte à compresser est *AAAABCDEFG*, alors `!nbcar` vaut 7, et les sous-tableaux significatifs de `car` et `frequence` sont respectivement `[|`A`;`B`;`C`;`D`;`E`;`F`;`G`|]` et `[|4;1;1;1;1;1;1|]`, ce qui conduit à l'arbre de Huffman ci-contre : on en déduit le code de Huffman des 7 caractères :



A	*B*	*C*	*D*	*E*	*F*	*G*
11	000	001	010	011	100	11

Pour construire le code Shannon-Fano de ces caractères, on part du tableau `car =[|`B`;`C`;`D`;`E`;`F`;`G`;`A`|]` trié dans l'ordre croissant des fréquences, et donc du tableau `frequence = [|1;1;1;1;1;1;4|]`. On obtient successivement :



Ce qui donne le tableau des codes suivant :

A	*B*	*C*	*D*	*E*	*F*	*G*
11	0000	0001	001	010	011	10

Remarque

Dans le fichier Huffmanc.ml, on peut tester les algorithmes des questions 11) à 14) sur l'exemple de cette question, mais il faut au préalable relancer Camlwin, car les variables globales `car` et `frequence` sont directement utilisées, et avec des définitions différentes, dans les fonctions `parcours` et `shannon_fano`.
Pour procéder à ce test :

- arrêter Camlwin sans arrêter le frontal.
- évaluer l'exemple 2 uniquement dans Huffmanc.ml.

16)Le nombre moyen de bits nécessaire au codage d'un caractère pour le codage de *AAAABCDEFG* est :

a)avec l'algorithme de Huffman : $\frac{20}{7} \approx \underline{2,9}$.

b)avec l'algorithme de Shannon-Fano : $\frac{21}{7} = \underline{3,0}$.

On ne peut en tirer aucune conclusion, d'abord parce que la différence n'est pas significative, et ensuite parce que même si elle l'était, le texte à coder est trop court et trop particulier pour être représentatif des textes habituellement codés avec ce type de codage.

Remarque générale sur le problème

Ce problème est intéressant parce qu'il traite d'un algorithme très connu et très performant de codage de fichier. Il est seulement dommage que son auteur n'ait pas voulu faire deux sujets différents pour les programmations en Caml et en Pascal, comme cela s'est généralement fait pour les énoncés des autres concours de la session 97. Cela entraîne l'utilisation abusive de pointeurs (références en Caml), normale en Pascal, mais particulièrement inélégante et lourde en Caml.
L'auteur n'a visiblement pas cherché à ce que les programmes demandés puissent être effectivement utilisés pour coder un texte. D'un autre côté, les algorithmes demandés le sont directement en Caml, et une certaine connaissance du langage, qui dépasse les fonctions rappelées en page 7, est indispensable, pénalisant le candidat capable de concevoir les algorithmes demandés, mais qui aurait une pratique insuffisante de Caml : à vous d'en tirer

les conclusions, à savoir qu'il est indispensable de suer sur les feuilles de TD devant la machine, et que la compréhension théorique n'est pas suffisante si elle n'est pas confrontée à la programmation réelle...