```
(*
   Compression de Huffman
   Projet d'IPF
   Alexandre Darcherif 2013/2014
*)


(* I/ Outils *)

(* Question 1 *)

type arbre=
|Feuille of char
|Noeud of arbre * arbre

let arbre =  Noeud(Feuille 't', Noeud(Feuille 'a',Feuille 'o'));;

(* Question 2 *)
(*
  Interface nb_noeuds
  type: arbre -> int
  args: a: Arg. de type arbre
  pre: -
  postcondition: resultat = nombre de noeuds de l'arbre
  test:
        - nb_noeuds arbre (i.e Noeud(Feuille 't', Noeud(Feuille 'a',Feuille 'o'))) -> - : int 2
*)

let rec nb_noeuds a = match a with
|Feuille c -> 0
|Noeud (b,d) -> 1 + nb_noeuds b + nb_noeuds d;;

nb_noeuds arbre;;

(* Question 3 *)

(*
  Interface nb_feuilles
  type: arbre -> int
  args: a: Arg. de type arbre
  pre: -
  postcondition: resultat = nombre de feuilles de l'arbre
  test:
        - nb_feuilles Noeud(Feuille 't', Noeud(Feuille 'a',Feuille 'o')) -> - : int 3
*)

let rec nb_feuilles a = match a with
|Feuille c -> 1
|Noeud (b,d) -> nb_feuilles b + nb_feuilles d;;

nb_feuilles arbre;;

(* Question 4 *)

(*
  Interface trouve
  type: char * (char * 'a) list -> 'a
  args: x: Arg. de type string
        l: Arg. de type ('b * 'a) list
  pre: -
  postcondition: resultat = valeur associee a x
  test: Pour l = [("e",1);("a",1);("i",1);("s",2)]
       - trouve  l 'a';; -> - : int = 1
       - trouve  l 'f';; -> Exception: Failure "La liste de couple est vide ou le caractère recherché
n'est pas présent dans la liste !!"
       - trouve  l 'e';; -> - : int = 3
*)

let rec trouve = function
  |([],x) -> failwith "La liste de couple est vide ou le caractère recherché n'est pas présent dans la
liste !!"
  |((a,b)::q,x)->if x=a then b else trouve (q,x);;
```

```
trouve  ([('b',1);('a',1);('d',15);('e',3)],'e');;

(* Question 5 *)


(*
  Interface insertion
  type: 'a -> 'a list -> 'a list
  args: x : element à ajouter dans la liste
        l : liste étudiée
  pre: La liste doit être triée dans le sens decroissant
  post: resultat = insertion de l'élément dans la liste
  test:
        - insertion (11,[10;5;4;3;2;1;0]);; -> - : int list = [11; 10; 5; 4; 3; 2; 1; 0]
        - insertion (5,[10;5;4;3;2;1;0]);; -> - : int list = [10; 5; 5; 4; 3; 2; 1; 0]
        - insertion 10 [];; -> - : int list = [10]

On pourrait améliorer cette fonction en prenant en paramètre une fonction qui compare l'élément x et
ceux de l et qui renvoie true ou false suivant les cas.
*)
let rec insertion (x,l) = match (x,l) with
|(x,[]) -> [x]
|(x,t::q) -> if x>t then x::t::q else t::insertion (x,q);;

insertion (5,[10;5;4;3;2;1;0]);;

(* II/ Decompression *)

(* Question 1 *)

(*
  Interface cherche_car
  type: 'a arbre -> bool list -> char * bool list
  args: x: arbre
        l: code
  pre: Un arbre en paramètre
  post: resultat =  char * bool list ( le caractère , et la liste des booléens true/false )
  test: cherche_car arbre [false;true;false;false;true;true];; -> - : char * bool list = ('t', [true;
false; false; true; true])

*)
let rec cherche_car (x,l) = match (x,l) with
|(Noeud(g,d),[]) -> failwith " le code n'est pas optimal ( ou inexistant ) "
|(Feuille g,q) -> (g,q)
|(Noeud (d,g), t::q ) -> if t then cherche_car(g,q) else cherche_car (d,q);;

cherche_car (arbre,[false;true;false;false;true;true]);;

(* Question 2 *)

(*
  Interface decode
  type: 'a arbre -> bool list -> char list
  args: x: arbre
        l: code
  pre: Un arbre en paramètre
  post: Une liste de caracteres ( char )
  test: decode (arbre,6,[false;true;false;false;true;true]);; Retourne ['t';'a';'t';'o']

*)
let rec decode (x,n,l) = match (x,n,l) with
|(x,n,[]) -> []
|(x,n,q) -> if n>0 then let (a,z) = cherche_car (x,l) in [a]@decode(x,n-1,z)
                else [];;
decode (arbre,6,[false;true;false;false;true;true]);;

(* III/ Compression *)

(* a/ Construction de l'arbre *)

(*
```

```
   Interface frequences
   type: 'a list -> ('a * int) list
   args: l : liste de caractère dont on cherche la frequence
   pre: -
   post: Liste de couple (lettre, frequence)
   test: frequences ['t';'e';'x';'t';'e'];; - : (int * char) list = [(1, 'x'); (2, 'e'); (2, 't')]

*)

let cmp (a,b) (c,d) = if (a>c) then 1 else if a=c then 0 else (-1);;
let rec frequences l =
  let rec f1 = function
    |(x,[])->[(x,1)]
    |(x,(a,b)::q2)-> if x=a then (a,b+1)::q2 else (a,b)::(f1(x,q2)) in
  let rec f2 = function
    |[]->[]
    |t1::q1 -> let r = f2 q1 in f1 (t1,r) in
  let cmp (a,b) (c,d) = if (b>d) then 1 else if b=d then 0 else (-1) in
  List.sort cmp (f2 l);;

frequences ['t';'e';'x';'t';'e'];;

let rec compteurfrequence l = match l with
|[]-> 0
|(b,c)::q -> c + (compteurfrequence q);;

(* Question 2 *)


(*
   Interface initaliste_liste_arbre
   type: (char * 'a) list -> ('a * 'b arbre) list
   args: l: Liste renvoyée par la fonction frequences
   pre: -
   post: Liste de couples (frequence, Feuille lettre)
   test: initialiste_liste_arbre (frequences ['t';'e';'x';'t';'e']);; - : (int * arbre) list = [(1,
Feuille 'x'); (2, Feuille 'e'); (2, Feuille 't')]
*)


let rec initialise_liste_arbre = function
  |[]->[]
  |(a,b)::q->(b,Feuille a)::(initialise_liste_arbre q);;

initialise_liste_arbre (frequences ['t';'e';'x';'t';'e']);;

(* Question 3 *)


let rec creer_arbre = function
  |[]-> failwith"pb"
  |[(p,a)]->a
  |(p1,a1)::(p2,a2)::q-> creer_arbre (List.sort cmp ((p1+p2,Noeud (a1,a2))::q));;

(* b/ Encodage *)

(*
  Interface creer_traduction
  type: 'a arbre -> (char * bool list) list
  args: a: Arbre de Huffman
        l: Liste vide qui sera remplit lors de la récursion
  pre: -
  post: Liste de couples de forme (char, liste de booleens)
*)

let creer_traduction a =
  let rec aux (arbre,l) = match (arbre,l) with
    |(Feuille a,l) -> [(a,l)]
    |(Noeud (a,b),l) -> aux (a,l@[false]) @ aux (b,l@[true])
  in aux (a,[]);;

(*
```

```
   Interface encode_liste
   type: 'a list * ('a * 'b list) list -> 'b list
   args: a: texte a encoder (Liste de char)
         b: Liste des chemins de chaque char (Liste de couple (char, liste de booleens))
   pre: -
   post: Liste de booleens representant le texte encode

*)

let rec encode_liste (a,b) = match (a,b) with
  |([],_)->[]
  |((t::q),l)-> trouve (l,t) @ encode_liste (q,l);;

(* IV/ Extension : gestion des entrées et sorties *)

(* Lecture et ecriture *)

(*
   Interface charge
   type:  val lire : in_channel -> char list = <fun>
          val charge : string -> char list = <fun>
   args: st: texte à charger (en liste de char )
         ii: variable contenant le nom du fichier à ouvrir
   pre: -
   post: Liste de char qui correspond au texte chargé

*)
let rec lire = function(ii) ->
  try
    let i = input_char(ii) in
      i::lire(ii)
  with End_of_file -> [];;

let charge = function(st) ->
  let ii = open_in st in
    lire(ii);;
charge("test.txt");;

(*
   Interface sauve
   type:  val ecrire : out_channel * string list -> unit = <fun>
          val sauve : string * string list -> unit = <fun>
   args: st: texte à charger (en liste de char )
         oo: variable contenant le nom du fichier sur lequel ecrire
         li: liste à ecrire
   pre: -
   post: Liste de char qui correspond au texte chargé

*)
let rec ecrire = function
    oo, [] -> close_out oo
  | oo, x::li ->  begin output_string oo x; ecrire(oo,li) end;;

let sauve = function(st, li) ->
  let oo = open_out st in
    ecrire(oo,li);;

(* idem *)
let rec ecrira = function
    oo, [] -> close_out oo
  | oo, x::li ->  begin output_char oo x; ecrira(oo,li) end;;

let sauva = function(st, li) ->
  let oo = open_out st in
    ecrira(oo,li);;

(* Compression des données *)

        (* stockage du code *)

                (* passage de base 2 à base 10 *)
```

```
let rec puissance a =
if a = 0 then 1
else 2*(puissance (a-1));;

                (* conversion 8 caractère = > 1 caractère & compression*)

(*
   Interface compresse_code
   type:  val compresse_code : bool list -> string = <fun>
   args: code: code à compresser ( liste de booléens )
         l: liste à compresser
         a: compteur ( de 7 à 0 )
         n: entier => on passe de 8 octets à 1 seul ( sauf peut-être pour le dernier )
   pre: -
   post: Liste de caractère dont le code ascii en base 2 correspond à 8 booléens (ou moins, pour le
dernier cas)

*)

let string_of_char c = String.make 1 c;;
let compresse_code code =
let rec c8c1 (l,a,n) = match l with
|[]-> string_of_char(char_of_int (n))
|t::q -> if t then if a=0 then (string_of_char(char_of_int(n+1)))^(c8c1 (q,7,0))
                  else c8c1 (q,(a-1),(n + puissance a))
        else if a=0 then (string_of_char(char_of_int(n)))^(c8c1 (q,7,0))
        else c8c1 (q,a-1,n) in c8c1 (code,7,0);;

(*
   Interface compresse_arbre
   val compresse_arbre : out_channel * (char * int) list -> unit = <fun>
   args: oo: variable contenant le nom du fichier sur lequel ecrire
         liste:  frequences à compresser (liste de couple char*int)
   pre: -
   post: compression des frequences , les nombres correspondant aux frequences sont codé sur 4 octets

*)
let rec compresse_arbre (oo,liste) = match (oo,liste) with
    oo,[] -> close_out oo
    | oo,(a,b)::q -> begin
        output_string oo (string_of_char a);
        output_binary_int oo b;
        compresse_arbre (oo,q)
        end;;

let compression = function(st, li) ->
  let oo = open_out st in
    compresse_arbre (oo,li);;



                (* recuperation du code *)
(*
   Interface recuperation_code
   val recuperation_code : char list -> bool list = <fun>
   args: t: premier argument d'encode_liste
   pre: -
   post: recuperation du premier argument d'encode_liste ( liste de booléens )

*)
let recuperation_code t = let a=creer_arbre (initialise_liste_arbre (frequences t)) in encode_liste
(t,creer_traduction a);;
                (*stockage du code dans un fichier txt *)

sauve("test2.txt", [compresse_code (recuperation_code (charge("test.txt")))]);;
compression("test3.txt",frequences(charge("test.txt")));;

(*decompression des frequences*)

(*
   Interface decompresse_arbre
   val decompresse_arbre : in_channel -> (char * int) list = <fun>
```

```
   args: st: texte à charger (en liste de couple de char & int )
         ii: variable contenant le nom du fichier à ouvrir
   pre: -
   post: Liste de couple de char & int correspondant à la frequence
*)
let rec decompresse_arbre = function(ii) ->
  try
    let i = input_char(ii) and j= input_binary_int (ii) in
      (i,j)::decompresse_arbre(ii)
  with End_of_file -> [];;

let decompresse = function(st) ->
  let ii = open_in st in
    decompresse_arbre(ii);;
decompresse("test3.txt");;

                (* conversion base 10 -> base 2 *)


let c10c2 c =
let rec aux (c,b) =
if b = 0 then [c = 1]
else if (c-(puissance b) >= 0) then (true)::(aux (c-puissance b,b-1))
     else (false)::(aux (c,b-1))
in aux (c,7);;

(* decompression du code *)

let rec decompresse_code l = match l with
|[]->[]
|t::q -> (c10c2 (int_of_char t))@(decompresse_code q);;

decompresse_code (charge("test2.txt"));;

(*création de l'arbre*)
creer_arbre (initialise_liste_arbre (decompresse("test3.txt")));;

(*restitution du texte*)

sauva("test4.txt",decode(creer_arbre (initialise_liste_arbre (decompresse
("test3.txt"))),compteurfrequence (frequences (charge("test.txt"))),decompresse_code (charge
("test2.txt"))));;
```