TP 36

Algorithme de Lempel-Ziv-Welsh Compression

L'algorithme d'Huffman est efficace, mais il présente un désavantage majeur : dans sa version classique, il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal.
Dans ce TP, nous allons étudier une autre technique de compression qui, bien que moins efficace en pratique que Huffman, se programme assez facilement et permet de compresser des flux plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.
Il s'agit de l'*algorithme de Lempel-Ziv-Welch*, appelé communément compression *LZW*, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

1 Principe de la compression

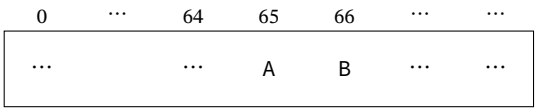
L'idée de l'algorithme LZW est de faire avancer une fenetre sur le texte en maintenant à jour une table des motifs déjà rencontrés. Quand on rencontre un motif déjà vu, on le code avec une référence vers la table et quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.
On se fixe une longueur de code d : on peut alors avoir une table t de longueur 2^d (ou alors un tableau dynamique dont la taille ne pourra dépasser 2^d). Ainsi, on pourra référencer chaque motif avec un mot de d bits [1]. Afin de pouvoir retrouver efficacement l'indice associé à un motif, on utilise une table de hachage réalisant l'inverse de la table.
Cette table de hachage (un dictionnaire donc) que l'on se constitue au fur et à mesure de la compression (le tableau de motifs) n'est pas transmis : comme les règles pour l'ajout et le référencement de motifs sont non ambiguës, il pourra être reconstitué lors de la décompression.
L'algorithme procède alors ainsi pour compresser :

- on initialise la table avec une entrée pour chaque caractère (donc chaque octet, en considérant typiquement des caractères sur huit bits) ;
- on maintient une variable contenant le plus long suffixe m du texte lu qui soit présent dans la table, il est initialisé avec la première lettre du texte ;
- on lit alors chaque caractère x :
  - soit mx est dans la table, et alors on remplace le motif courant par mx (*i.e.* $m \leftarrow mx$),
  - Soit mx n'est pas dans la table, par construction m y est nécessairement on produit alors le code correspondant à m, on rajoute une entrée dans la table pour mx si elle contient moins de 2^d éléments et le suffixe étudié devient x (*i.e.* $m \leftarrow x$) ;
- quand tous les caractères ont été lus, on produit le code correspondant à m.

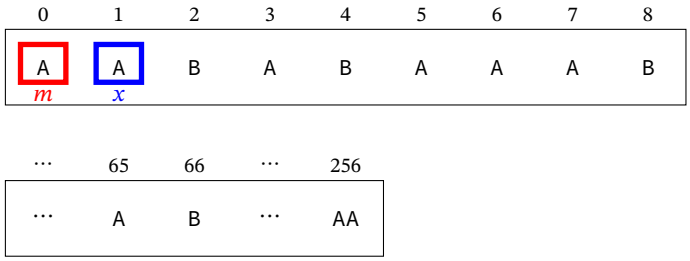
1. il existe une variante de l'algorithme avec une longueur variable, mais elle est plus subtile à mettre en œuvre

Voici les différentes étapes pour la compression de la chaîne AABABAAAB qui produit la suite d'entiers 65, 256, 66, 65, 258, 257 qui seront alors codés dans un fichier sur d bits.

- À l'initialisation, une entrée pour chaque caractère :

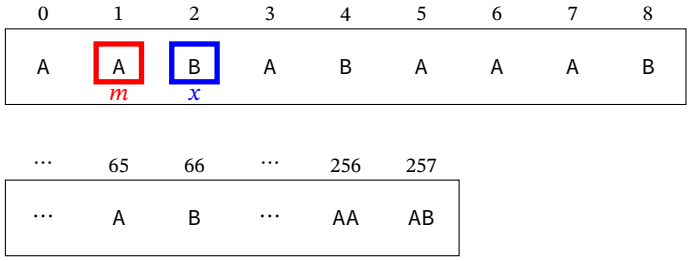


- Première étape : mx =AA n'est pas dans la table, on le rajoute à l'indice 256, m prend la valeur A et on emet le code 65 en sortie :



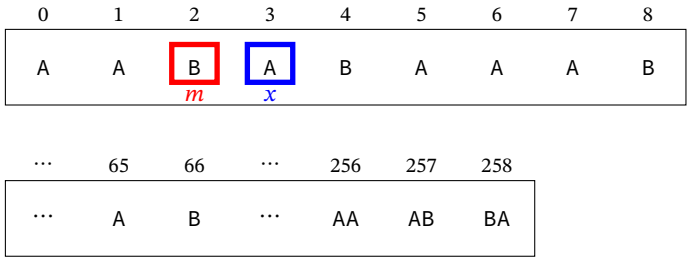
Sortie :
65

- Deuxième étape : mx =AB n'est pas dans la table, on le rajoute à l'indice 257, m prend la valeur B et on emet le code 65 en sortie :



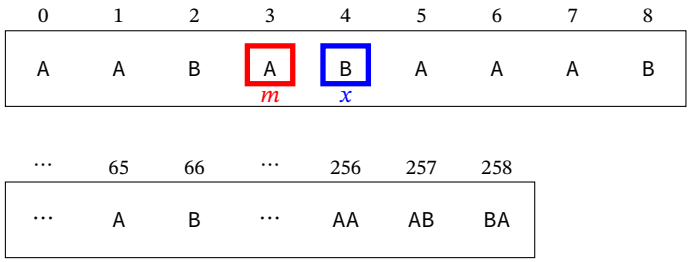
Sortie :
65
65

- Troisième étape : mx =BA n'est pas dans la table, on le rajoute à l'indice 258, m prend la valeur A et on emet le code 66 en sortie :



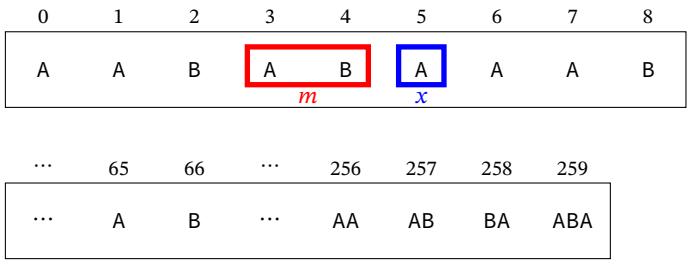
Sortie :
65
65
66

- Quatrième étape : mx =AB est dans la table ! m prend la valeur AB et on n'emet aucun code en sortie :



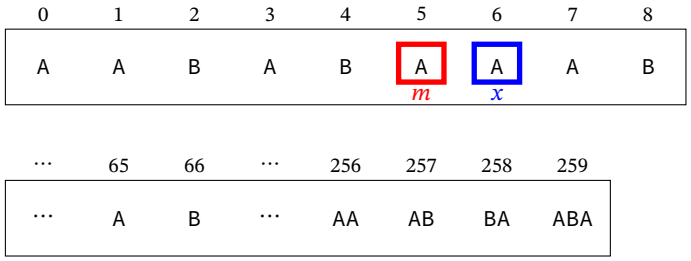
Sortie :
65
65
66

- Cinquième étape :



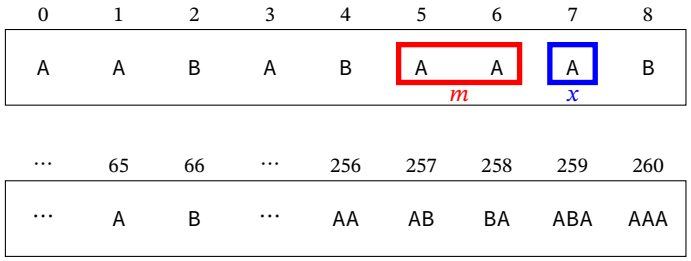
Sortie :
65
65
66
257

- Sixième étape :



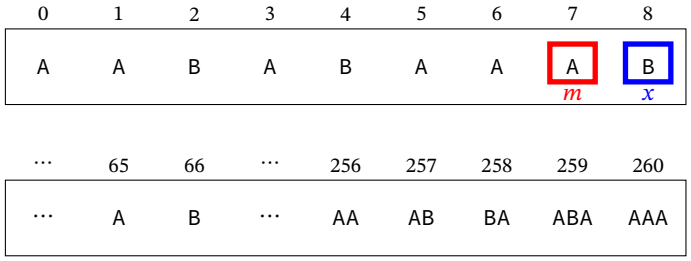
Sortie :
65
65
66
257

- Septième étape :



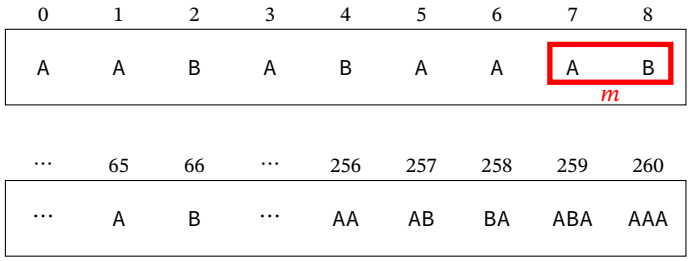
Sortie :
65
65
66
257
256

- Huitième étape :



Sortie :
65
65
66
257
256

- Neuvième étape : on a épuisé les caractères en entrée, on emet le code correspondant à m :



Sortie :
65
65
66
257
256
257

Exercice 36.1 Simuler la compression de la séquence ABABABA.

II Lecture/écriture bit à bit dans un fichier

Rappel. On utilise les commandes suivantes pour ouvrir un fichier en mode binaire :

```
let f_in = open_in_bin "nom_du_fichier" in ...
let f_out = open_out_bin "nom_du_fichier" in ...
```

La fermeture se fera avec les commandes `close_in f_in` et `close_out f_out`.

L'unité atomique de lecture ou d'écriture est l'octet (*byte* en anglais). Comme pour l'algorithme d'Huffman, on veut écrire des codes d'une longueur qui n'est pas nécessairement égale à 8 bits. Il faut donc travailler un peu plus.
Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit ainsi que le nombre de bits qui ont été accumulés. Dès qu'on accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis réinitialiser ces variables.
Si l'accumulateur vaut $acc = b_0 \dots b_{i-1}$ et qu'on lui ajoute un bit b, on veut obtenir $acc = b_0 \dots b_{i-1}b$. Cela revient à faire l'opération $acc \leftarrow acc + b2^i$.
On considère le type suivant :

```
type out_channel_bits =
  { o_fichier : out_channel
  ; mutable o_accumulateur : int
  ; mutable o_bits_accumules : int
  }
```

Le champ `o_accumulateur` correspond au *acc* ci-dessus, et le champ `o_bits_accumules` au nombre de bits contenus dans l'accumulateur (autrement dit, au i ci-dessus). La fonction suivante correspond à l'ouverture d'un fichier :

```
let open_out_bits fn =
  { o_fichier = open_out_bin fn
  ; o_accumulateur = 0
  ; o_bits_accumules = 0
  }
```

Exercice 36.2 Écrire une fonction `output_bit : out_channel_bits -> bool -> unit` qui traite l'écriture d'un bit [2] en mettant à jour l'accumulateur et le nombre de bits accumulés, et en écrivant un octet dans le canal de sortie le cas échéant.

Il faut encore s'occuper du dernier octet potentiellement incomplet. Si l'on dispose de k bits à la fin de l'écriture, il faut les compléter avec $8 - k$ zéros pour écrire un octet. Avec notre ordre d'écriture des bits, cela ne change en fait pas la valeur de l'accumulateur. Cependant, ces zéros ne sont pas significatifs et il faudra tenir compte de ce fait à la lecture. Pour cela, on ajoute un dernier octet indiquant le nombre de zéros de *padding* présents dans l'avant-dernier octet du flux.

Exercice 36.3 Écrire une fonction `close_out_bits : out_channel_bits -> unit` permettant de fermer un fichier.

Pour la lecture, on maintient de même un accumulateur et un nombre de bits accumulés. Quand on nous demande un bit :

- si l'on ne dispose d'aucun bit accumulé, on lit un octet du fichier (et l'on dispose maintenant de 8 bits, sauf cas particulier traité plus bas) ;
- ensuite, on récupère le bit de poids faible de l'accumulateur, et l'on décale ensuite les autres bits. Autrement dit, si $acc = b_0 \dots b_{i-1}$, on veut renvoyer b_0 et remplacer *acc* par $b_1 \dots b_{i-1}$.

2. le `bool` du type correspond au bit à écrire : `false = 0` et `true = 1`

Il faut traiter correctement les zéros de *padding* ajoutés en fin de fichier. Pour cela, il est nécessaire de savoir si l'on est en train de lire l'avant-dernier octet (celui qui a été « rembourré ») : on calcule la taille du fichier à l'ouverture et l'on vérifie à chaque lecture d'octet. Si l'on est en train de lire cet octet rembourré, il suffit de lire l'entier suivant pour savoir combien de bits sont à ignorer.
On donne le type et les fonctions d'ouverture et de fermeture :

```
type in_channel_bits =
  { i_fichier : in_channel
  ; mutable i_accumulateur : int
  ; mutable i_bits_accumules : int
  ; i_taille : int
  }

let open_in_bits fn =
  let fichier = open_in_bin fn in
  { i_fichier = fichier
  ; i_accumulateur = 0
  ; i_bits_accumules = 0
  ; i_taille = in_channel_length fichier
  }

let close_in_bits f = close_in f.i_fichier
```

Exercice 36.4 Écrire la fonction `input_bit : in_channel_bits -> bool` qui permet de lire un bit du flux.

III Implémentation

On redonne ici les fonctions classiques de la bibliothèque `Hashtbl` pour gérer automatiquement un dictionnaire mutable :

- `Hashtbl.create` crée une table de hachage vide (l'entier passé en argument donne la taille initiale de la table, il est souhaitable qu'il soit de l'ordre du nombre d'éléments que l'on compte stocker dans la table) ;

  ```
  Hashtbl.create : int -> ('a, 'b) Hashtbl.t
  ```

- `Hashtbl.mem` telle que `Hashtbl.mem t k` renvoie `true` si `t` a une valeur associée à `k`, `false` sinon ;

  ```
  Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool
  ```

- `Hashtbl.find` prend en argument une table et une clé et renvoie la valeur associée à la clé (ou une exception `Not_found` s'il n'y a pas de telle valeur) ;

  ```
  Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b
  ```

- `Hashtbl.find_opt`, la même chose avec une option plutôt qu'une exception ;

  ```
  Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option
  ```

- `Hashtbl.add` telle que `Hashtbl.add t k v` rajoute l'association (k, v) à la table t.

  ```
  Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
  ```

Exercice 36.5 Écrire une fonction `output_code` telle que l'appel `output_code f code d` va écrire les d bits du code dans le fichier f.

```
output_code : open_out_bits -> code -> int
```

Exercice 36.6 Écrire une fonction `lzw_comp` qui prend en argument un entier et deux noms de fichiers tel que l'appel `lzw_comp d entrée sortie` compresse le fichier d'entrée vers le fichier de sortie en utilisant un code de longueur d.

```
lzw_comp : int -> string -> string -> unit
```

Pour tester, vous pouvez récupérer les fichiers `ex1.txt` (qui contient uniquement le texte de l'exemple détaillé ci-dessus) et `ex1.comp` (qui est le résultat de la compression pour un code sur douze bits). Vous devez obtenir exactement le même fichier que moi après compression. Vous pouvez tester s'il y a une différence avec la commande `diff` (qui ne doit rien afficher) :

```
diff ex1.comp votre_fichier
```

Exercice 36.7 Vous pouvez aussi télécharger sur mon site le texte complet de *La guerre des boutons* de Louis Pergaud (qui est dans le domaine public).
Tester l'influence de la longueur du code choisie sur la taille du fichier compressé.