Devoir 1. Compression d'une séquence d'entiers

Le sujet de ce travail pratique est le développement d'un outil de compression[1]. L'entrée de la procédure de compression est une séquence de nombres naturels $\{x_i \colon\ = 0, 1, 2, \ldots\}$ (abstraction d'un fichier d'octets, par exemple). Notre procédure encode l'entrée à l'aide d'une transformation qui produit une séquence $\{y_i \colon i = 0, 1, 2, \ldots\}$ de *petits* entiers quand l'entrée est compressible. Ensuite, on encode les entiers en une séquence de chaînes de bits $\{z_i \colon i = 0, 1, 2, \ldots\}$, en utilisant moindre de bits pour les petits entiers. Il est important que les deux étapes sont invertibles instantanément : on peut calculer $x_i \leftrightarrow y_i$ et $y_i \leftrightarrow z_i$ à chaque $i = 0, 1, 2, \ldots$. En conséquence, la procédure est une méthode d'encodage déchiffrable.

[1] Voir §5.5 de Sedgewick et Wayne pour une discussion détaillée de méthodes de compression.

1.1 Move-to-front

La transformation **move-to-front** (MTF) sert à encoder une séquence d'entiers par une autre. Pour encoder la séquence $x_0, x_1, x_2, \ldots$ d'entiers non-négatifs, on maintient une permutation π. Initialement, on a identité : $\pi[n] = n$ pour tout n.

Pour transmettre le prochain entier n, on identifie son unique position i dans la permutation ($\pi[i] = n$), et on émet i. En même temps, on performe une rotation circulaire de $\pi[0..i]$ qui place n en position 0. Dans la permutation résultante π', on a $\pi'[0] = \pi[i]$, $\pi'[j] = \pi[j-1]$ pour $j = 1, 2, \ldots, i$ et $\pi'[j] = \pi[j]$ pour tout $j > i$.

Pour décoder, on fait la même rotation sauf que l'argument est l'indice i, et on doit retourner $\pi[i]$ (avant la rotation).

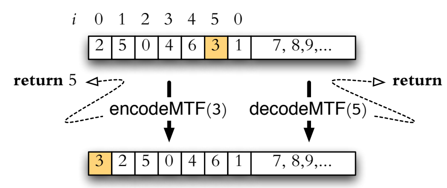

FIG. 1: La transformation MTF

Exemple : la transformation MTF de la séquence $(1, 2, 5, 2, 1, 4, 3, 0)$ donne $(1, 2, 5, 1, 2, 5, 5, 5)$:

i	x_i	encodeMTF(x_i)	π (après rotation)
0	1	1	$\begin{pmatrix}1 & 0 & 2\cdots\infty\end{pmatrix}$
1	2	2	$\begin{pmatrix}2 & 1 & 0 & 3\cdots\infty\end{pmatrix}$
2	5	5	$\begin{pmatrix}5 & 2 & 1 & 0 & 3 & 4 & 6\cdots\infty\end{pmatrix}$
3	2	1	$\begin{pmatrix}2 & 5 & 1 & 0 & 3 & 4 & 6\cdots\infty\end{pmatrix}$
4	1	2	$\begin{pmatrix}1 & 2 & 5 & 0 & 3 & 4 & 6\cdots\infty\end{pmatrix}$
5	4	5	$\begin{pmatrix}4 & 1 & 2 & 5 & 0 & 3 & 6\cdots\infty\end{pmatrix}$
6	3	5	$\begin{pmatrix}3 & 4 & 1 & 2 & 5 & 0 & 6\cdots\infty\end{pmatrix}$
7	0	5	$\begin{pmatrix}0 & 3 & 4 & 1 & 2 & 5 & 6\cdots\infty\end{pmatrix}$

1.2 Comment encoder les entiers ?

On veut transmettre une séquence de nombres naturels $y_0, y_1, y_2, \ldots$ sur une ligne de communication. La difficulté est qu'on ne peut utiliser que deux symboles, comme en code Morse : $\boxed{0}$ et $\boxed{1}$. (Il s'agit de l'abstraction d'un fichier binaire.)

On définit donc un code $\rho(y)$ (séquence de symboles de $\boxed{0}$ et $\boxed{1}$) pour tout y, et on transmet les codes $\rho(y_0)$, $\rho(y_1)$, $\ldots$, l'un après l'autre. En plus d'être *déchiffrable* (on peut récupérer la séquence y_0, y_1, $\ldots$ sans ambiguïté à partir de la chaîne concaténée), ρ devrait être *instantané* : on veut déchiffrer y_i dès qu'on reçoit tous les bits de $\rho(y_i)$. Et, enfin, on désire aussi que le code soit économique.

Encodage unaire. L'exemple le plus simple est le *code unaire* $\alpha(n)$:

$$\alpha(n) = \underbrace{\boxed{1}\boxed{1}\ldots\boxed{1}}_{n \text{ fois}}\boxed{0} \tag{1.1}$$

Clairement, il est possible de décoder une séquence de codes concaténés $\alpha(y_0)$, $\alpha(y_1)$, $\ldots$, car il suffit de compter les $\boxed{1}$s jusqu'au premier $\boxed{0}$. Mais ce code n'est pas économique de tout ...

Encodage binaire. L'encodage binaire standard $\beta(n)$ est beaucoup plus efficace, parce qu'il utilise seulement d symboles où d est le plus petit nombre naturel tel que $n < 2^d$. Pour $n = \sum_{i=0}^{d-1} b_i \cdot 2^i$ (avec $b_i \in \{0, 1\}$ et $b_{d-1} = 1$) on transmet la séquence $b_{d-1}, b_{d-2}, \ldots, b_0$, en commençant par le bit de poids fort. En particulier $\beta(0) = \lambda$ (chaîne vide), $\beta(1) = \boxed{1}$, $\beta(2) = \boxed{1}\boxed{0}$, etc. On définit le code par récurrence

$$\beta(n) = \begin{cases} \lambda & \text{vide si } n = 0 \\ \boxed{1} & \{n = 1\} \\ \beta\left(\frac{n}{2}\right) \oplus \boxed{0} & \{n = 2, 4, 6, 8, \ldots\} \\ \beta\left(\frac{n-1}{2}\right) \oplus \boxed{1} & \{n = 3, 5, 7, \ldots\} \end{cases}$$

où $\oplus$ dénote la concaténation. Hélas, β n'est pas déchiffrable : $(1, 2, 1)$ et $(6, 1)$ mènent à la même séquence de bits $\boxed{1}\boxed{1}\boxed{0}\boxed{1}$.

Encodage γ. On peut construire un code instantané en préfixant $\beta(n)$ par sa longueur encodé en unaire : si $\beta(n)$ comprend d symboles, alors on écrit d fois $\boxed{1}$, suivi par un seul $\boxed{0}$, et l'encodage binaire sur d bits.

$$\gamma(n) = \alpha\Big(\big|\beta(n)\big|\Big) \oplus \beta(n), \tag{1.2}$$

où $\oplus$ dénote la concaténation et $|\ldots|$ la longueur en bits. Le code γ est instantanément déchiffrable : compter les $\boxed{1}$s au début ($= d$), jusqu'au premier $\boxed{0}$, et décoder les d bits suivants contenant $\beta(y_i)$.

Encodage ω. On peut améliorer le code γ en spécifiant le nombre de bits avec un encodage plus efficace que l'unaire. On se sert de la récursivité :

$$\omega_0(n) = \begin{cases} \lambda & \text{chaîne de longueur } 0 \text{ quand } n = 0 \\ \omega_0\Big(\big|\beta(n)\big| - 1\Big) \oplus \beta(n) & \{n > 0\} \end{cases} \tag{1.3}$$

On rend le code déchiffrable par un bit $\boxed{0}$ apposé : on transmet $\omega_1(n) = \omega_0(n) \oplus \boxed{0}$. Par exemple, $\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}$ encode la séquence $7, 2, 0$.

n	$\beta(n)$	$\gamma(n)$	$\omega_0(n)$
0	λ	0	λ
1	1	$\underbrace{10}_{\alpha(1)}\ \underbrace{1}_{\beta(1)}$	1
2	10	$\underbrace{110}_{\alpha(2)}\ \underbrace{10}_{\beta(2)}$	$\underbrace{1}_{\omega_0(1)}\ \underbrace{10}_{\beta(2)}$
3	11	$\underbrace{110}_{\alpha(2)}\ \underbrace{11}_{\beta(3)}$	$\underbrace{1}_{\omega_0(1)}\ \underbrace{11}_{\beta(3)}$
4	100	$1110\ 100$	$\underbrace{1\ 10}_{\omega_0(2)}\ \underbrace{100}_{\beta(4)}$
5	101	$1110\ 101$	$1\ 10\ 101$
6	110	$1110\ 110$	$1\ 10\ 110$
7	111	$1110\ 111$	$1\ 10\ 111$
8	1000	$11110\ 1000$	$\underbrace{1\ 11}_{\omega_0(3)}\ \underbrace{1000}_{\beta(8)}$
16	10000	$111110\ 10000$	$\underbrace{1\ 10\ 100}_{\omega_0(4)}\ \underbrace{10000}_{\beta(16)}$
2015	11111011111	[23 symboles]	$1\ 11\ 1010\ 11111011111$

1.3 Travail à faire

Développement algorithmique (30 points)

Move-to-front. ▶ Développer une structure de données[2] qui implante le type abstrait avec opérations encodeMTF et decodeMTF. La structure est initialisée avec le maximum M tel que tout entier à l'entrée sera entre 0 et $(M-1)$. Montrez l'implantation des opérations avec tous les détails.

Encodage universel. On écrit les chaînes de bits à l'aide d'une interface de sortie : la méthode writeBit(b) écrit le bit b dans le fichier. ▶ Donner un algorithme *récursif* encodeOmega(n) pour écrire[3] l'encodage $\omega_1(n)$. (Noter que l'algorithme doit aussi calculer $|\beta(n)|$ et écrire $\beta(n)$.)

Implémentation Java (30 points)

▶ Implémenter un outil de compression en Java qui emploit la transformation MTF et l'encodage universel ω.

- ⋆ L'exécutable s'appelle `MTFOmega`, et il prend 1 argument de la ligne de commande qui est le nom d'un fichier à lire
- ⋆ Le fichier d'entrée est ouvert par `FileReader` (comme un flot de caractères), et lu à l'aide de la méthode `read()` (qui retourne une valeur de 16 bits, entre 0 et 65535)
- ⋆ Chaque entier x_i lu est écrit comme encodeOmega(encodeMTF(x_i))
- ⋆ La sortie est un fichier binaire, écrit sur `System.out` (on écrit un octet par la méthode `write`). Ici, il faut encoder la séquence de bits en une séquence d'octets. Utilisez le principe (gros-boutiste) (*big-endian*), et implémentez[4] une structure de buffer/tampon pour empaqueter les bits.

Exemple d'application à la ligne de commande (code dans `tp1.jar`) :

```
% java -cp tp1.jar MTFOmega fichier.txt > fichier.zzz
```

▶ Tester l'efficacité de la méthode développée sur quelques exemples : choisir quelques fichiers (5–10) sur le site https://introcs.cs.princeton.edu/java/data/, et comparer la taille du fichier comprimé avec celle d'un autre logiciel de compression au choix (par exemple, `gzip`). Compiler les résultats dans un tableau : pour chaque jeu de donnés, donner en octets (1) la taille originale, (2) la taille comprimée selon `MTFOmega`, et (3) la taille comprimée selon l'autre outil.

[2] **Indice:** Pour stocker la permutation actuelle π, on peut utiliser une liste chaînée ou un tableau. Initialiser avec la permutation d'identité.

[3] **Indice:** Appelez writeBit(0) pour $\boxed{0}$ et writeBit(1) pour $\boxed{1}$.

[4] Vous avez le droit d'adapter le code de `BinaryOut.java` de Sedgewick et Wayne, avec mention de la source.