Compression BZip

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `bzip` contenant plusieurs fichiers dont une source OCaml :

```
curl cdn.sci-phy.org/mp2i/tp11-bzip.tgz | tar xvz
```

Le fichier à compléter est le fichier `bzip.ml`. On y trouvera quelques fonctions utiles qui seront présentées au fur et à mesure de nos besoins. On y trouve également quatre fichiers de tests, `kanagawa.txt`, `adn.dat`, `declaration.txt` et `ocaml.ppm`.

1.2 Objectifs

Dans ce sujet, on s'intéresse à la compression d'une chaîne de caractères (et à sa décompression), selon une variante de la méthode utilisée par l'outil de compression bzip. Cet outil tente de diminuer la taille occupée en mémoire par une chaîne de caractères en détectant les occurrences successives d'un même caractère, et en les enregistrant plus efficacement. L'utilisation de la transformation de Burrows-Wheeler pemettra d'augmenter le nombre de répétitions de caractères, et donc l'efficacité de la méthode.

Il vous est fourni une fonction « `read` » prenant en argument un chemin vers un fichier et retournant une chaîne de caractères correspondant au contenu de ce fichier.

1.3 Soucis pratiques

Travailler avec des chaînes de caractères pose quelques problèmes pratiques. Parmi ceux-ci, une limitation de l'outil `WinCaml` peut vous causer des soucis : les chaînes de caractères, notamment celle obtenue en chargeant le fichier `ocaml.ppm`, peuvent contenir le caractère de code « 0 », et si l'application tente d'afficher une telle chaîne, alors l'affichage risque fort de se bloquer.

Il est, de toute façon, plus simple de travailler avec des entiers qu'avec des caractères. Les chaînes de caractères en OCaml sont de simples successions de caractères, supposés encodés dans un format de type ASCII étendu. Chaque caractère peut donc être associé à un entier entre 0 et 255, et il est possible de passer d'un caractère à l'entier qui lui est associé et inversement grâce aux fonctions `int_of_char` et `char_of_int`.

Pour simplifier les choses, on travaillera donc avec des tableaux d'entiers entre 0 et 255. On dispose donc d'une fonction `to_array` permettant de convertir une chaîne de caractères en tableau d'entiers (une seconde fonction `to_string` est également disponible si l'on souhaite effectuer la conversion inverse, par exemple pour enregistrer le résultat dans un fichier, mais attention à ne pas provoquer un affichage de chaînes contenant des caractères de code « 0 »!).

Pour obtenir un tableau d'entiers correspondant au contenu d'un fichier, on pourra donc simplement écrire (en adaptant le chemin au besoin)

```
let data = to_array (read "U:/bzip/adn.dat")
```

On cherche ici à écrire une fonction de *compression* prenant en argument un tableau d'entiers entre 0 et 255 et retournant un tableau d'entiers entre 0 et 255 de taille si possible plus petite, et tel qu'il soit possible, au travers d'une autre fonction, de *décompression*, de retrouver le tableau original.

Bien évidemment, il n'est pas possible d'obtenir *à tous les coups* un tableau plus petit! Mais la très grande majorité des fichiers contenant des données qui ne sont pas déjà compressées devraient, on le verra, voir leur taille réduite par l'algorithme de compression étudié.

2 Compression par redondance

Avant de commencer, on détermine, dans le tableau en entrée, l'entier k qui apparaît le moins fréquemment (voire jamais). En cas d'égalité, on choisira arbitrairement parmi les possibilités. Cet entier k est placé en tête du tableau qui contiendra le résultat. Puis la compression par redondance s'effectue de la façon suivante :

- un entier k est encodé par la succession de deux entiers k et 0
- un entier différent de k qui n'est pas répété est simplement codé par lui-même ;
- deux entiers identiques consécutifs, différents de k, sont également codés par eux-mêmes ;
- un entier c répété p fois avec $3 \leqslant p \leqslant 255$ est codé par une séquence de trois entiers : l'entier k, l'entier p, et enfin l'entier c lui-même.
- un entier c répété p fois avec $p > 255$ est codé comme $\lfloor p/255 \rfloor$ répétitions de 255 fois l'entier c, suivi ce qu'il convient pour coder $p \mod 255$ fois l'entier c (par exemple, 1000 occurrences du caractère de code 42 seront encodées par la séquence « k, 255, c, k, 255, c, k, 255, c, k, 235, c »).

Prenons un exemple, et construisons un tableau pour une chaîne simple :

```
# let data = to_array "abbaaaacabbbbcccc";;
val data : int array = [|97; 98; 98; 97; 97; 97; 97; 99;
                         97; 98; 98; 98; 98; 99; 99; 99; 99|]
```

On choisira par exemple $k = 0$, ce qui donnera le tableau suivant après compression :

```
# encode_RLE data;;
- : int array = [|0; 97; 98; 98; 0; 4; 97; 99; 97; 0; 4; 98; 0; 4; 99|]
```

Le tableau de 17 caractères a donc été ici compressé en un tableau de 15 caractères. Le facteur de compression est ici faible, mais nous verrons que sur d'autres exemples, le gain peut être important.

1. Proposer une fonction `frequencies` de signature `int array -> int array` prenant en argument un tableau d'entiers `t` et retournant un tableau de 256 entiers tel que la case d'index `i` de ce tableau corresponde au nombre d'occurrences de l'entier `i` dans le tableau `t`.

2. En déduire une fonction `less_frequent` de signature `int array -> int` prenant en argument un tableau d'entiers `t` et retournant l'entier entre 0 et 255 apparaissant le moins fréquemment dans le tableau `t` (en cas d'égalité dans le nombre d'apparitions, on prendra le plus petit de ces entiers).

3. Écrire une fonction `count_repeats` de signature `int array -> int -> int` qui prend en argument un tableau d'entiers et un entier indiquant une position dans le tableau, et retourne le nombre d'entiers successifs identiques à compter de cette position. **Si ce nombre dépasse 255, on retournera 255**.

Par exemple, en prenant pour argument le tableau d'exemple et 0, la fonction devra retourner 1. En lui passant ce même tableau et 3, elle devra retourner 4, et en lui passant toujours le même tableau et 5, elle devra retourner 2. Enfin, en lui passant le tableau et 14, elle devra retourner 3 (il est vivement recommandé d'effectuer ces tests, en particulier le dernier, car un mauvais fonctionnement de cette fonction posera des difficultés ultérieurement).

4. Écrire une fonction `size_encoded` de signature `int array -> int` qui prend en entrée un tableau d'entiers et retourne un entier indiquant la longueur qu'aura le tableau résultat de la compression (15 sur notre exemple).

5. Déterminer la taille avant et après compression RLE des quatre fichiers fournis en exemple.

	avant compression	après compression
`adn.dat`		
`kanagawa.txt`		
`declaration.txt`		
`ocaml.ppm`		

Les tailles après compression modulo 42 devraient être 9, 33, 17 et 33.

6. Interpréter les résultats obtenus (notamment l'efficacité ou non de la compression) sur les différents fichiers.

7. Écrire une fonction `encode_RLE` de signature `int array -> int array` qui prend en entrée un tableau de caractères et retourne un tableau de caractères compressé par la méthode proposée au-dessus. On pourra commencer par créer un tableau de la bonne longueur avant d'y placer les bons codes.

Note : RLE signifie « Run Length Encoding », et désigne l'encodage par redondance que l'on implémente ici.

Pour vérifier que le contenu d'un tableau `arr` encodé avec `encode_RLE` est correct, vous pouvez utiliser la commande « `Hashtbl.hash arr` » et comparer l'entier obtenu avec ceux ci-dessous.

fichier	hash original	hash RLE
`adn.dat`	542983240	786551765
`kanagawa.txt`	399184142	950838657
`declaration.txt`	934097618	365614891
`ocaml.ppm`	875403533	97021462

3 Décompression RLE

8. Écrire une fonction `decode_RLE` de signature `int array -> int array` réalise l'opération inverse. On pourra d'abord créer une fonction qui détermine la taille du tableau décompressé, avant de procéder à la décompression proprement dite.

9. Vérifier que le tableau décompressé correspond bien au tableau initial pour chacun des fichiers d'exemples. On pourra par exemple faire le test suivant :

```
let data = to_array (read "U:/bzip/adn.dat") in
  data = decode_RLE (encode_RLE data);;
```

4 Transformée de Burrows-Wheeler et compression BZip

4.1 Principe et objectifs

L'ennui est, comme on l'a vu, que cette méthode n'est pas très efficace sur un texte normal, car on a rarement un très grand nombre de caractères identiques qui se succèdent, autrement dit, si la chaîne compressée n'est généralement pas beaucoup plus longue, elle est très rarement raccourcie.

La transformée de Burrows-Wheeler est une technique proposée en 1983 par Michael Burrows et David John Wheeler. Elle réorganise des données afin d'augmenter les chances que des données identiques se retrouvent côte à côte.

Pour illustrer le principe, considérons la chaîne de caractères [1] « `concours` » (on continue à travailler sur des tableaux d'entiers, mais il est plus simple, pour présenter la transformation, de visualiser les choses sous la forme de caractères. La chaîne « `concours` » correspond en fait au tableau d'entiers `[|99; 111; 110; 99; 111; 117; 114; 115|]`).

On note $\mathcal{R}[i]$ la permutation circulaire vers la gauche de la chaîne de i rangs. Ainsi, $\mathcal{R}[0]$ correspond à « `concours` », $\mathcal{R}[1]$ à « `oncoursc` » ou bien $\mathcal{R}[3]$ à « `courscon` ».

Il y a autant de permutations que de caractères présents dans la chaîne (certaines pouvant être identiques). Dans le cas du mot « `concours` », cela donne 8 permutations, que l'on classe par ordre lexicographique :

```
concours
courscon
ncoursco
oncoursc
oursconc
rsconcou
sconcour
ursconco
```

Le résultat de la transformation de Burrows-Wheeler sur la chaîne de caractères « `concours` » correspond aux caractères présents dans la dernière colonne ci-dessus, soit la chaîne de caractères « `snoccuro` ».

On mémorise également l'indice de la lettre, dans la chaîne obtenue, qui correspond à la première lettre de la chaîne originale. Ce sera la *clé de la transformation*. Elle sera indispensable pour réaliser la transformation inverse, mais c'est la seule information supplémentaire que l'on ait besoin. Dans le cas présent, la clé sera 3 (attention, il y a également un `c` en position 4, mais il ne s'agit pas du premier caractère de « `concours` »).

Si l'on en revient à nos tableaux d'entiers, cela donnera :

```
# let data = to_array "concours";;
val data : int array = [|99; 111; 110; 99; 111; 117; 114; 115|]

# encode_BW data;;
- : int array * int = ([|115; 110; 111; 99; 99; 117; 114; 111|], 3)

# let data = to_array "concours"
    in let arr, key = encode_BW data
      in to_string arr, key;;
- : string * int = ("snoccuro", 3)
```

1. Cette séance de travaux pratique se base largement sur une épreuve écrite du concours Polytechnique, et les chaînes de caractères prises en exemple ont été conservées.

Dans la chaîne transformée, les deux `c` se sont retrouvés côte à côte. Pour des chaines plus longues, grâce au tri, cela arrive encore plus fréquemment, ce qui augmente l'efficacité des algorithmes de compression utilisant les redondances.

Par exemple, la chaîne « `concours de l'ecole polytechnique` » sera transformée par cet algorithme en « `seeleeen dlt'ucn ooohcpcc iuryqol` » avec pour clé 23.

```
# let data = to_array "concours de l'ecole polytechnique"
    in let arr, key = encode_BW data
      in to_string arr, key;;
- : string * int = ("seeleeen dlt'ucn ooohcpcc iuryqol", 23)
```

On voit ici apparaître des séquences de trois lettres identiques successives, ce qui laisse espérer de meilleurs résultats lors d'une compression par redondance sur des textes plus longs.

10. Écrire une fonction `compare_rotations` dont la signature sera compatible avec `int array -> int -> int -> int`, qui accepte en argument un tableau d'entiers et deux entiers i et j (positifs et strictement inférieurs à la longueur du tableau) et retourne :

- `1` si $\mathcal{R}[i] > \mathcal{R}[j]$ pour l'ordre lexicographique ;
- `-1` si $\mathcal{R}[i] < \mathcal{R}[j]$ pour l'ordre lexicographique ;
- `0` si $\mathcal{R}[i] = \mathcal{R}[j]$.

On déterminera le résultat **sans calculer explicitement les rotations**, car le tableau fourni en paramètre peut être très long !

On fournit, pour la suite, une fonction `sort_rotations` qui trie les rotations du tableau de caractères `s` par ordre croissant, et retourne un tableau `r` d'entiers représentant les numéros, ordonnés, des rotations ($\mathcal{R}[r[0]] \leqslant \mathcal{R}[r[1]] \leqslant \ldots \leqslant \mathcal{R}[r[n-1]]$). Cette fonction effectue, dans la majorité des cas, $O(n\ln(n))$ appels à la fonction de comparaison. Il s'agit d'un tri rapide, qui sera étudié un peu plus tard.

```
let sort_rotations t =
  let n = Array.length t in
  let order = Array.init n (fun i -> i) in
  Array.sort (compare_rotations t) order;
  order;;
```

11. Écrire une fonction `encode_BW` de signature `int array -> int array * int` qui prend en argument un tableau de caractères et retourne un tuple constitué du tableau après transformation et de la clé.

12. Écrire une fonction `bzip` de signature `int array -> int array * int` qui applique la méthode de Burrows-Wheeler à un tableau de caractères puis le compresse en exploitant les redondances, et retourne le tableau de caractères correspondant et la clé de la transformation de Burrows-Wheeler.

13. Analyser les résultats obtenus avec les quatre fichiers test.

	avant compression bzip	après compression bzip
`adn.dat`		
`kanagawa.txt`		
`declaration.txt`		
`ocaml.ppm`		

Bien évidemment, d'autres subtilités entrent en jeu dans le format de compression bzip pour obtenir le meilleur facteur de compression possible, quelle que soit la source, mais les idées essentielles sont là! Vous pouvez cette fois encore utiliser la commande « `Hashtbl.hash arr` » et comparer l'entier obtenu avec ceux ci-dessous pour vérifier le bon fonctionnement de vos fonctions. Le tableau contient aussi lés clés.

fichier	hash original	hash BZip	clé
`adn.dat`	542983240	230887398	19693
`kanagawa.txt`	399184142	701580960	15152
`declaration.txt`	934097618	437203177	3903
`ocaml.ppm`	875403533	802555628	118668

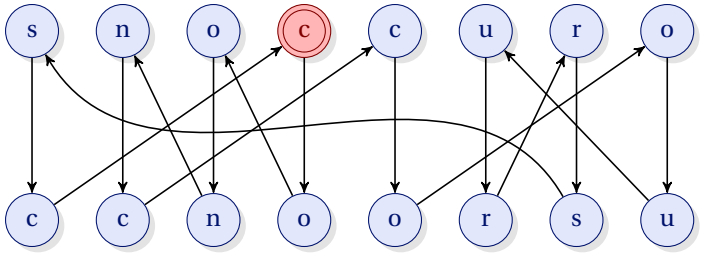
5 Décompression BZip

Pour que cette transformation puisse être utilisée pour de la compression de données, il est indispensable de pouvoir réaliser l'opération inverse inverse.

Pour ce faire, nous allons montrer que la transformation de Burrows-Wheeler peut être inversée. On s'intéresse donc à une chaîne contenant les mêmes caractères que notre chaîne codée, mais réordonnés selon l'entier qui leur est associé. Reprenons par exemple le cas de la chaîne « `concours` ».

À chaque caractère de la chaîne codée, on associe le caractère placé au même endroit dans la chaîne triée. À chaque caractère de la chaîne triée, on associe le même caractère de même rang dans la première.

La figure suivante montre ces deux correspondances :



On retrouve le texte de départ en partant de la clé (ici 3, mise en évidence dans le schéma ci-dessus) et en suivant simplement les flèches.

Il faut donc contruire un tableau `t` d'indices entiers tel que `t.(i)` soit la position, dans le tableau représentant la chaîne encodée, correspondant à la i^e lettre de la chaîne triée. Le tableau `t` donne donc la correspondance représentée par les flèches de la seconde ligne vers la première. Sur l'exemple, ce serait `[| 3; 4; 1; 2; 7; 6; 0; 5 |]`.

14. Écrire la fonction `find_indices` de signature `int array -> int array` prenant en paramètre un tableau tel qu'obtenu lors d'un appel à `codege_BW` et retournant le tableau d'entiers précédemment décrit. Pour ce faire, on réfléchira à la façon dont le résultat fourni par la fonction `frequencies` peut nous renseigner sur la position de la première occurrence d'un caractère de code i dans la chaîne triée.

15. Écrire une fonction `decode_BW` de signature `int array * int -> int array` qui prend en argument un tableau de caractères et un entier (clé) obtenus par l'application de l'algorithme de Burrows-Wheeler, et retourne le tableau original.

16. Écrire une fonction `bunzip` de signature `int array * int -> int array` qui effectue l'opération inverse de la fonction `bzip`.