Algorithmique Avancée

TD Compression de Données

1 Méthodes élémentaires

Exercice* 1.1 *: Suppression des blancs

Principe : lorsqu'une séquence de blancs est rencontrée, elle est remplacée par un couple de caractères : le premier est un caractère spécial indiquant qu'il s'agit d'une répétition, le second est un compteur du nombre de blancs.

Question 1.1.1 Écrire un algorithme de compression.

Question 1.1.2 Écrire un algorithme de décompression.

Exercice* 1.2 *: Répétitions dans un texte

Principe : lorsqu'une séquence d'occurrences d'un même caractère est rencontrée, elle est remplacée par une suite de trois caractères : un caractère spécial indiquant qu'il s'agit d'une répétition, puis le nombre de répétitions, puis le caractère répété.

Question 1.2.1 Écrire un algorithme de compression.

Question 1.2.2 Écrire un algorithme de décompression.

Exercice* 1.3 *: Image et Run length Encoding (RLE)

Principe : dans le cas particulier d'une image en noir et blanc, représentée par une suite de bits "0" (pixel éteint = noir) et "1" (pixel allumé = blanc), on a souvent des suites de plus de 8 pixels noirs (ou de blancs). On peut utiliser un compteur codé sur un octet pour compter alternativement le nombre de "0" et le nombre de "1".

Par exemple : 000000000000000000011111111111111111000000000000111111111000000000000
se compresse en 18 16 12 9 10.

Question 1.3.1 Comment prendre en compte les suites d'au moins 256 pixels de même couleur ?

Question 1.3.2 Écrire un algorithme de compression.

Question 1.3.3 Écrire un algorithme de décompression.

2 Compression sans dictionnaire

Exercice* 2.4 *: Propriétés des codes

Rappel : soient w_1 et w_2 des mots sur l'alphabet S, on dit que w_1 est un *préfixe* strict de w_2, et l'on note $w_1 < w_2$ ssi il existe un mot non vide u tel que $w_2 = w_1 u$.

Soit A un alphabet. Un code $C : A \to \{0,1\}^+$ est dit *préfixe* (en fait on devrait dire "sans-préfixe" !) ssi aucun mot du code (on appelle mot du code un élément de l'ensemble $C(A)$) n'est préfixe d'un autre mot du code.

Un code préfixe $C : A \to \{0,1\}^+$ est dit *complet* ssi, lorsque u est un mot du code et $x < u$, alors les mots $x0$ et $x1$ sont soit des mots du code, soit des préfixes de mots du code.

Par la suite, on identifiera souvent C et $C(A)$ et on appliquera l'expression "code préfixe" aussi bien à $C(A)$ qu'à C.

Question 2.4.1 $\{00, 001, 10\}$ est-il un code préfixe ? un code préfixe complet ?
Et $\{00, 010, 101, 110\}$? Et $\{0, 100, 101, 111, 1100, 1101\}$?

Question 2.4.2 On considère le code $C : a \mapsto 0$, $b \mapsto 100$, $c \mapsto 101$, $d \mapsto 111$, $e \mapsto 1100$, $f \mapsto 1101$. Décoder le message : 10101011011101011000101010100101010100.

Question 2.4.3 Soit $A = \{a_1, \ldots, a_n\}$ un alphabet d'au moins deux lettres. On suppose que chaque lettre a_i a une probabilité $p_i = \frac{1}{2^{k_i}}$ avec $p_1 \geq p_2 \geq \ldots \geq p_n$

1. Montrer que $p_{n-1} = p_n$.
2. Montrer, par récurrence sur n, qu'il existe un codage préfixe complet C tel que le nombre de bits de $C(a_i)$ est égal à k_i, pour tout $i \in \{1, \ldots, n\}$.
3. Soit T un texte écrit sur $A = \{a_1, \ldots, a_n\}$, dans lequel chaque lettre a_i apparaît avec la probabilité $p_i = \frac{1}{2^i}$ si $1 \leq i \leq n-1$ et $p_n = \frac{1}{2^{n-1}}$. Soit C un codage préfixe complet tel $C(a_i)$ a i bits, si $i \in \{1, \ldots, n-1\}$ et $C(a_n)$ a $n-1$ bits. Quel est le nombre moyen de bits par caractère dans ce codage ?

Exercice 2.5 ***: Algorithme de Shannon-Fano***

Principe : Soit A un alphabet de n lettres. On suppose que chaque lettre a_i a la probabilité p_i d'apparaître dans tous les textes formés sur A, et que $\sum_{i=1}^{n} p_i = 1$.
On suppose que $p_1 \geq p_2 \geq \ldots \geq p_n$, et on considère la liste $L = (a_1, a_2, \ldots, a_n)$ des lettres, rangées par ordre décroissant de leurs probabilités.
La méthode suivante décrit un codage de compression $C : A \to \{0,1\}^+$.

- Couper L en 2 sous-listes $L_1 = (a_1, a_2, \ldots, a_k)$ et $L_2 = (a_{k+1}, a_{k+2}, \ldots, a_n)$, telles que les sommes cumulées des probabilités de L_1 et de L_2 soient le plus voisines possibles, c'est-à-dire choisir k qui minimise la différence $|\sum_{i=1}^{k} p_i - \sum_{i=k+1}^{n} p_i|$.
- Chaque lettre de L_1 reçoit 0 comme premier bit de codage et chaque lettre de L_2 reçoit 1 comme premier bit de codage.
- Recommencer récursivement le traitement sur L_1et L_2 pour déterminer les bits suivants.
- Le processus s'arrête lorsqu'une liste ne contient plus qu'un seul caractère.

Par exemple, si A contient les cinq lettres a, b, c, d et e, de probabilités respectives 0.35, 0.17, 0.17, 0.16 et 0.15, les codes sont 00 pour a, 01 pour b, 10 pour c, 110 pour d et 111 pour e.

Question 2.5.1 Donner la suite des codes générés pour la suite des lettres $(a_1, \ldots, a_9)$, de probabilités respectives $(1/4, 1/4, 1/8, 1/8, 1/8, 1/32, 1/32, 1/32, 1/32)$.

Question 2.5.2 Montrer que le code engendré par la méthode de Shanon-Fano est préfixe complet.

Question 2.5.3 Pour une lettre a_i on note p_i sa probabilité et l_i la longueur du code associé. On appelle *longueur moyenne des mots du code* la quantité $\sum_{i=1}^{n} l_i p_i$. La longueur moyenne des mots du code obtenu par la méthode précédente est-elle toujours minimale ?

Question 2.5.4 Décrire en quelques lignes le principe des algorithmes de compression et de décompression d'un texte T en utilisant cette méthode. Évaluer la complexité de ces algorithmes en fonction de la taille N du texte.

Exercice 2.6 ***: Algorithme de Huffman statique***

Le principe a été vu en cours et les codes préfixes et complets sont définis dans l'Exercice 3.7.

Question 2.6.1 On suppose que A contient les cinq lettres a, b, c, d et e, de probabilités respectives 0.35, 0.17, 0.17, 0.16 et 0.15. Quel est le code généré par l'algorithme de Huffman ? Construire l'arbre de Huffman associé.

Question 2.6.2 Mêmes questions pour A contenant les neuf lettres $a_1, \ldots, a_9$, de probabilités respectives $1/4, 1/4, 1/8, 1/8, 1/8, 1/32, 1/32, 1/32, 1/32$.

Supposons que A contient n lettres $a_0, \ldots, a_{n-1}$. On compresse un texte T écrit sur l'alphabet A au moyen de l'algorithme de Huffman.

Question 2.6.3 Donner la forme de l'arbre de Huffman et le taux de compression lorsque toutes les lettres $a_0, \ldots, a_{n-1}$ ont la même probabilité d'apparition dans le texte T.

Question 2.6.4 Même question lorsque le nombre d'apparitions de la lettre a_i $(i = 0, \ldots, n-1)$ dans le texte T est égal à 2^i.

Exercice* 2.7 *: Mot de Dyck-ABCIF

Soit A un alphabet. Dans cet exercice, on dira qu'un mot w écrit sur $A \cup \{0, 1\}$ est un *mot de Dyck-ABCIF* sur A (ABCIF pour *a*rbre *b*inaire *c*omplet avec *i*nformations aux *f*euilles) s'il possède les propriétés suivantes :
- le nombre de "1" de w est égal à son nombre de "0" augmenté de 1
- dans tout préfixe strict de w, le nombre de "1" est inférieur ou égal au nombre de "0"
- dans w, chaque "1" est immédiatement suivi d'une lettre de A et chaque lettre de A est immédiatement précédée d'un "1".

On rappelle que tout arbre binaire complet avec informations aux feuilles et étiquettes dans A peut être représenté par un mot de Dyck-ABCIF sur A (*cf.* transparents du cours).

Question 2.7.1 Énumérer les primitives sur les arbres et les mots qui permettent de répondre aux deux questions ci-dessous.

Question 2.7.2 Écrire l'algorithme qui code un arbre binaire complet avec informations aux feuilles et étiquettes dans A au moyen d'un mot de Dyck-ABCIF sur A.

Question 2.7.3 Écrire l'algorithme qui, étant donné un mot de Dyck-ABCIF w, construit l'arbre binaire complet codé par w.

Question 2.7.4 Appliquer l'algorithme précédent au mot $01a001b1c001e1f1d$.

Exercice* 2.8 *: Algorithme de Huffman dynamique

Principe : La méthode de Huffman compresse un texte de façon optimale en codant les caractères les plus fréquents par les codes les plus courts. L'algorithme de Huffman dynamique construit un arbre de Huffman adaptatif qui évolue au fur et à mesure de la lecture et du traitement (compression ou décompression) des symboles. L'algorithme de compression (compresseur) et l'algorithme de décompression (décompresseur) modifient l'arbre de la même façon, de telle sorte qu' à chaque instant ils utilisent les mêmes codes (mais ces codes changent au cours du temps).
Initialement le compresseur démarre avec un arbre vide : aucun symbole n'a encore reçu de code. Le premier symbole lu est transmis avec son codage initial (ASCII, ou codage sur 5 bits si le texte ne contient que les 26 lettres de l'alphabet). Ce symbole est alors ajouté à l'arbre et reçoit ainsi un code (de longueur variable). La prochaine fois que ce symbole sera rencontré, le compresseur transmettra son code actuel et incrémentera sa fréquence. Il se peut alors que l'arbre ne soit plus un arbre de Huffman adaptatif (AHA) et si c'est le cas, le compresseur le modifie, ce qui implique une modification des codes.
Le décompresseur travaille de la même façon : lorsqu'il lit le code initial d'un symbole, il l'ajoute à l'arbre et lui affecte ainsi un code ; lorsqu'il lit un code de longueur variable, il utilise l'arbre courant pour déterminer le symbole correspondant, puis il modifie l'arbre de la même manière que le compresseur.
Il faut bien sûr que le décompresseur sache si le code qu'il lit est un codage initial ou un code de longueur variable : pour cela chaque codage initial est précédé d'un code spécial d'échappement, de longueur variable. Ainsi lorsque le décompresseur lit ce code d'échappement, il sait que les k bits suivants ($k = 8$ pour l'ASCII, $k = 5$ pour un codage initial sur 5 bits) correspondent au codage initial d'un symbole qui n'est pas encore apparu dans le texte compressé lu jusque là.
La valeur du code spécial d'échappement évolue aussi au cours du traitement : pour ce faire, on ajoute à l'arbre une feuille spéciale #, de fréquence 0, dont la position dans l'arbre varie au cours du temps.

Modification de l'arbre : On rappelle les propriétés suivantes :
Propriété 1 *Soit H un AHA avec n feuilles et $n-1$ nœuds internes. Dans la numérotation hiérarchique GDBH* GaucheDroiteBasHaut *$x_1, x_2, \ldots, x_{2n-1}$ des nœuds, on a :*

1. *$W(x_1) \leq W(x_2) \leq \ldots \leq W(x_{2n-1})$, où $W(x_i)$ est le poids du nœud x_i.*
2. *Pour $1 \leq i \leq n-1$, les nœuds x_{2i-1} et x_{2i} sont frères (i.e. ont le même père dans l'arbre).*

Propriété 2 *Etant donné un AHA et une feuille f, de numéro x_{i_0}, dont le chemin à la racine est $\Gamma_f = x_{i_0}, x_{i_1}, \ldots, x_{i_k}$ ($i_k = 2n-1$), alors l'arbre résultant de l'incrémentation de $W(f)$ sera encore un AHA ssi $W(x_{i_j}) < W(x_{i_j+1})$, pour $0 \leq j \leq k-1$. On dira dans ce cas que tous les nœuds du chemin Γ_f sont incrémentables.*

L'algorithme ci-dessous effectue la modification de l'AHA H après lecture du symbole s. Il produit en résultat un AHA H' dans lequel la fréquence de s a été augmentée de 1, et les autres poids ont été modifiés en conséquence.
En plus des primitives classiques sur les arbres, on dispose d'une fonction `finBloc(H,m)` qui, étant donné un nœud m numéroté x_m dans un AHA H, renvoie le nœud b tel que $W(x_m) = W(x_{m+1}), \ldots = W(x_b)$ et $W(x_b) < W(x_{b+1})$.

Les deux algorithmes suivants sont présentés dans le cours.
*Modification : AHA * Symbole → AHA*

*Traitement : AHA * noeud → AHA*

Question 2.8.1 Montrer que l'algorithme n'échange jamais un nœud avec un de ses ancêtres.

Question 2.8.2 Quel est (en fonction de la taille de l'alphabet des symboles) le nombre maximal d'échanges réalisés lors d'une modification de l'arbre ?

Compression et Décompression : La compression se fait selon l'algorithme ci-dessous.

Procédure Compression (T :Texte)
var H : Huffman, s : Symbole
. H=feuille spéciale #
. **Répéter**
. s= symbole suivant de T
. **Si** $s \in H$ **Alors**
. transmettre le code de s dans H
. **Sinon**
. transmettre le code de # dans H puis le code initial de s
. **Fin Si**
. H=Modification (H, s)
. **Jusqu'à** T terminé
FinProcédure `Compression`

Question 2.8.3 Quelle est la complexité de cet algorithme, en temps et en place ?

Question 2.8.4 Pour $T = abracadabra$, avec le code initial sur 5 bits :
$a = 00000, b = 00001, c = 00010, d = 00011, \ldots, r = 10001, \ldots$
le texte compressé est 00000 000001 0010001 0 10000010 0 110000011 0 110 110 0 (les espaces servent à séparer les différentes transmissions). Expliciter les différentes étapes de la production du texte compressé, en construisant au fur et à mesure l'arbre de Huffman.

Question 2.8.5 Ecrire l'algorithme de décompression. Quelle est sa complexité ?
Vérifier qu'en suivant cet algorithme, la décompression de l'exemple précédent produit bien le texte *abracadabra*.

Exercice* 2.9 *: Exemple pour l'algorithme de Huffman dynamique

On reprend ici l'exemple présenté en cours. Les lettres minuscules sont codées sur 5 bits. `a:00000`, `b:00001`, `c:00010`, etc.

Les états successifs de l'arbre obtenu par l'algorithme de Huffman dynamique appliqué au texte `carambarbcm` sont présentés dans le cours.

Question 2.9.1 Quel est le texte compressé de `carambarbcm` ?

Question 2.9.2 En appliquant l'algorithme de Huffman dynamique aux textes `t1` et `t2`, on obtient les textes compressés suivants :

`c1 : 00001000000001000101110001111111010`

`c2 : 000101101000000101000000010010101010010011111011011101100100000000100000011101`0

Quels sont les textes `t1` et `t2` ?

Exercice* 2.10 *: Codage arithmétique

Principe : On ne code pas chaque lettre séparément, mais on code le texte entier par un nombre réel compris entre 0 et 1.
Soit A un alphabet de n lettres. On suppose que chaque lettre x_i a la probabilité p_i d'apparaître dans tous les textes formés sur A, et que $\sum_{i=1}^{n} p_i = 1$. On découpe l'intervalle $[0,1[$ en n intervalles de longueurs $p_1, \ldots, p_n$. Pour cela, on définit une suite croissante $l_0 = 0 < l_1 < \ldots < l_{n-1} < l_n = 1$, telle que $l_i - l_{i-1} = p_i$, pour $i = 1, \ldots, n$. L'intervalle $[l_{i-1}, l_i[$ est associé à la lettre x_i.
Au texte T que l'on veut coder, on associe un intervalle construit en emboîtant une suite d'intervalles traduisant les probabilités de la suite des lettres de T. Pour coder un texte $T = xyz\ldots$ écrit sur A :

- à la première lettre, x, est associé un certain intervalle $[l_{i-1}, l_i[$, que l'on note $[a, b[$; au mot formé de la seule lettre x, on associe cet intervalle $[a, b[$
- au mot xy formé des deux premières lettres de T, on associe un intervalle $[a', b'[$ contenu dans $[a, b[$ et construit de la façon suivante : on "rapporte" (par une homothétie et une translation) l'intervalle $[l_{j-1}, l_j[$ associé à la lettre y à l'intervalle $[a, b[$; par exemple, si $[a, b[= [0.1, 0.5[$ et si $[l_{j-1}, l_j[= [0.7, 0.9[$ alors l'intervalle $[a', b'[$ obtenu en "rapportant" l'intervalle $[l_{j-1}, l_j[$ à l'intervalle $[a, b[$ est égal à $[0.1 + 0.4 * 0.7, 0.1 + 0.4 * 0.9[= [0.38, 0.46[$
- on réitère cette construction jusqu'à la fin du texte T (on rapporte l'intervalle associé à la lettre z à l'intervalle $[a', b'[$, et ainsi de suite)
- par ce procédé, on associe au texte T un intervalle $[\alpha, \beta[$
- le code associé au texte T est le réel α.

Remarque : Si on prolonge le texte T, on affinera l'intervalle $[\alpha, \beta[$.

Question 2.10.1 Soit a et b deux réels tels que $a < b$. Déterminer la transformation (composée d'une homothétie et d'une translation) qui envoie l'intervalle $[0, 1[$ sur l'intervalle $[a, b[$. Quelle est l'image d'un intervalle $[c, d[$ (contenu dans $[0, 1[$) par cette transformation ?

On suppose que A contient les cinq lettres D, E, P, R et S, de probabilités respectives 0.1, 0.4, 0.1, 0.2 et 0.2.

Question 2.10.2 Coder le texte "DESESPERER".

Question 2.10.3 Décoder le réel 0.8534215.

Question 2.10.4 Écrire un algorithme qui réalise le codage : on en donnera deux versions (l'une itérative et l'autre récursive).

Question 2.10.5 Écrire un algorithme qui réalise le décodage : on en donnera deux versions (l'une itérative et l'autre récursive).

3 Compression avec dictionnaire

Exercice 3.11 ***: Lempel-Ziv-Welch (LZW), compression***

Principe : On rappelle le principe de l'algorithme de Lempel-Ziv-Welch. Soit T le texte sur un alphabet A qu'on veut compresser. On construit une table F de facteurs de T contenant à l'origine A. À chaque mot f dans F est associé un code $h(f)$. Ensuite, tant que T n'est pas vide, on réitère les quatre étapes suivantes :

- chercher le plus long préfixe f de T présent dans F,
- transmettre $h(f)$,
- si a est la lettre qui suit f dans T alors ajouter fa dans F,
- remplacer T par T privé de son préfixe.

À la i-ème étape de cet algorithme, on obtient un mot f_i de F. Le texte T initial est égal à $f_1 f_2 \dots f_k$ (et le texte compressé est égal à $h(f_1)\ h(f_2)\ \dots\ h(f_k)$). On dit qu'il y a un *chevauchement* dans T s'il existe un indice i tel que $f_{i-1} = xv$ et $f_i = xvx$, où x est une lettre de A et v un mot écrit sur A.

Pour les questions ci-dessous, on suppose que l'alphabet A contient n lettres $a_0, a_1, \dots, a_{n-1}$ et que la fonction h est construite comme suit :

- $h(a_0) = 0, h(a_1) = 1, \dots, h(a_{n-1}) = n-1$
- pour un mot fx qui n'est pas dans F et tel que $f \in F$ et $t = fxg$ on définira $h(fx)$ comme le plus petit entier qui n'est pas encore associé à un élément de F.

On considère $A = \{r, a, e\}$ avec $h(r) = 0, h(a) = 1$, $h(e) = 2$.

Question 3.11.1 Détailler le déroulement de l'algorithme sur le texte "rarearreeerearrrareeeeee" et déterminer le texte compressé.
Y a-t-il un chevauchement dans le texte initial ?

Question 3.11.2 On prolonge le texte précédent par : "rareeeerarrrerarreerareeeee" et on passe en phase stationnaire dans l'algorithme de compression. Continuer le déroulement de l'algorithme et déterminer la suite du texte compressé.

Question 3.11.3 On suppose maintenant que $A = \{e, r, u\}$ avec $h(e) = 0$, $h(r) = 1$ et $h(u) = 2$. Détailler le déroulement de l'algorithme sur le texte "errerrerrerrerreur" et déterminer le texte compressé.

Question 3.11.4 Il y a un chevauchement dans le texte initial, à quel endroit ?
Que se passe-t-il à cet endroit lors du déroulement de l'algorithme de compression ?

Question 3.11.5 Écrire un algorithme de compression.

Exercice 3.12 ***: LZW, décompression***

Principe : À partir du texte compressé et du dictionnaire initial (codage des lettres), on construit la table F des facteurs au fur et à mesure que l'on décompresse.

On fait les mêmes hypothèses sur l'alphabet A et sur la fonction h que dans l'exercice précédent.

Question 3.12.1 On suppose que $A = \{m, e, l, a, n, i, d\}$ et que que le dictionnaire initial est $h(m) = 0, h(e) = 1, h(l) = 2, h(a) = 3, h(n) = 4, h(i) = 5, h(d) = 6$.
On reçoit le message compressé suivant :
0 1 2 3 4 5 8 3 0 12 6 14 8 12 18 2 13 7 2 20 22 15 22.
Décoder ce message.

Question 3.12.2 On suppose que $A = \{a, b, r, t\}$ et que que le dictionnaire initial est $h(a) = 0, h(b) = 1, h(r) = 2, h(t) = 3$.
On reçoit le message compressé suivant :
2 0 3 5 7 3 1 0 1 4 9 5 10 12 5.
Décoder ce message.

Question 3.12.3 Soit $T_C = c_1 \dots c_k$ le texte compressé obtenu à partir d'un texte T. Montrer qu'il existe $f_1, \dots, f_k$ tels que :
(i) $T = f_1 \dots f_k$
(ii) pour tout $i = 1, \dots, k$, $c_i = h(f_i)$
(iii) pour tout $i = 2, \dots, k$, ou bien f_i est une lettre de A, ou bien il existe $j < i$ tel que $f_i = f_j a$, avec a première lettre de f_{j+1}.

Question 3.12.4 Quelles sont les conséquences du (iii) si $j < i - 1$? $j = i - 1$?

Question 3.12.5 Écrire un algorithme de décompression.

4 Compression arborescente

Exercice 4.13 ***: Isomorphisme d'arbres***

L'exercice est décomposé en deux parties : la première rappelant deux méthodes de tri de liste d'entiers (qui ne sont pas des tris par comparaisons !) et la deuxième partie consistant à définir un algorithme quasi linéaire permettant de résoudre le problème du test d'isomorphisme d'arbres.

Question 4.13.1 Étant donnée une liste de n entiers positifs, rappeler le fonctionnement du tri par dénombrement. Indiquer la contrainte nécessaire afin d'obtenir un algorithme de complexité temporelle $O(n)$.

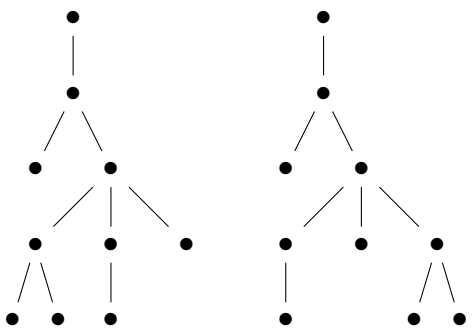
Voilà sur un exemple le fonctionnement du tri par base depuis les unités (nommé en anglais radix sort by least significant digit). Le but consiste à trier la liste d'entiers suivante $[765, 994, 23, 435, 888, 18, 524, 770]$. À chaque étape, c'est le chiffre souligné qui est pris en considération.

765		77$\underline{0}$		$\underline{1}$8		18
994		2$\underline{3}$		$\underline{2}$3		23
23		99$\underline{4}$		5$\underline{2}$4		$\underline{4}$35
435	$\rightarrow$	52$\underline{4}$	$\rightarrow$	4$\underline{3}$5	$\rightarrow$	$\underline{5}$24
888		76$\underline{5}$		7$\underline{6}$5		$\underline{7}$65
18		43$\underline{5}$		7$\underline{7}$0		$\underline{7}$70
524		88$\underline{8}$		8$\underline{8}$8		$\underline{8}$88
770		1$\underline{8}$		9$\underline{9}$4		$\underline{9}$94

Question 4.13.2 Écrire l'algorithme de tri par base en supposant qu'on accède au chiffre des unités de l'entier x via l'écriture $x[0]$, au chiffre des dizaines via l'écriture $x[1]$, ...
En utilisant la structure de données adéquate, on prouvera avoir une complexité temporelle linéaire en nombre d'écriture mémoire, dans le pire des cas.

On pourra remarquer que le tri par base peut s'appliquer au tri de chaînes de caractères sur un alphabet fini. En particulier si l'on considère que des entiers sont stockés sous forme de chaînes de caractères sur l'alphabet binaire, par exemple "1011".

Dans la suite de l'exercice, on considère l'ensemble des arbres enracinés et d'arité quelconque, i.e. les degrés sortants sont de valeurs arbitraires. Dans la partie gauche de la figure ci-dessous, nous proposons un exemple :



```
function Tree2Parenthesis(T a tree)
    res := ""
    for every child C of the root of T do
        res := Concatenate(res, "(", Tree2Parenthesis(C), ")")
    return res
```

Question 4.13.3 Donner une description en quelques lignes de la fonction Tree2Parenthesis.

Question 4.13.4 Appliquer la fonction Tree2Parenthesis aux deux arbres présentés ci-dessus.

Par la suite, on remplace les parenthèses ouvrantes "(" par "1" et les fermantes ")" par "0".

Question 4.13.5 Quelles sont les chaînes de caractères (binaires) encodant chacun des arbres ?

Par la suite, on considère les arbres enracinés d'arité quelconque et non plans, c'est-à-dire que les sous-arbres d'un nœud ne sont pas ordonnés. Remarquons que pour ce modèle, les deux arbres ci-dessus sont identiques.

Dans le cours, on a donné une version simplifiée (mais adaptée aux BDDs) de la résolution de l'isomorphisme d'arbres c'est-à-dire de l'algorithme qui teste si deux arbres sont identiques. En effet, on a uniquement traité le cas des arbres plans. On généralise la résolution ici. Voilà l'algorithme de Aho, Hopcroft et Ullman dans une version simple, mais avec une efficacité amoindrie (nous verrons l'algorithme optimal plus loin dans le TD).

Algorithme 1 Tree Isomorphism Problem

```
function NodeName(v a node)
    name := ""
    L = []
    for every child w of v do
        L.append(Concatenate("1", NodeName(w), "0"))
    L := sort(L)
    for ℓ in L do
        name := Concatenate(name, ℓ)
    return name
```

```
function TestIso(T1, T2 two trees)
    n1 := NodeName(Root(T1))
    n2 := NodeName(Root(T2))
    return n1 = n2
```

Lemme 1 *Deux arbres sont isomorphes si les noms affectés à leur racine par* NodeName *sont identiques.*

Question 4.13.6 Quelle est la complexité temporelle (au pire cas) de l'Algorithme 1 (indiquer la mesure de complexité pertinente) ? Exhiber le pire cas.

Nous introduisons ici l'algorithme de AHU sous sa version optimale. On rappelle que la profondeur d'un nœud est sa distance à la racine de l'arbre.

AHU Algorithm :
T_1 et T_2 sont les deux arbres en entrée.

1. Associer l'étiquette 0 à chacune des feuilles de T_1 et T_2.
2. Inductivement, supposons que chaque nœud du profondeur $i+1$ ait été associé à un entier. Soit L_1 la liste des nœuds du niveau $i+1$ de T_1, rangés par ordre croissant des entiers associés. Soit L_2 l'analogue pour T_2.
3. À chaque nœud interne w de profondeur i, on construit une liste L_w, en parcourant de gauche à droite la liste L_1, et dès qu'on tombe sur un nœud v fils de w, on ajoute à L_w l'entier associé à v. La liste L_w est directement triée car L_1 est triée, puis on concatène les éléments de L_w en sens inverse, en mettant les entiers bout-à-bout [a]. Soit S_1 la liste des entiers associés aux nœuds internes de niveau i.
4. Répéter l'étape 3. pour construire l'analogue S_2 pour T_2.
5. Trier les listes S_1 et S_2 en ordre croissant et les renommer S'_1 et S'_2.
6. Si S'_1 et S'_2 sont différentes alors T_1 et T_2 ne sont pas isomorphes. Sinon, chaque entier est remplacé par un nouvel entier (en partant de 1 et dans l'ordre gauche-droite de S'_1) et les entiers identiques de S'_1 sont remplacés par la même valeur. La suite de nombres, précédée par les feuilles à 0 constituent la nouvelle liste L_1 associée au niveau i.
   On reprend à l'étape 3. pour le niveau $i-1$.
7. Si les racines de T_1 et T_2 sont associées au même entier, alors les arbres sont isomorphes.

a. Si $L_w = [0, 0, 1, 1, 3, 10]$ alors la concaténation donne 1031100. De plus 00 est différent de 0.

Lemme 2 *L'algorithme AHU est correct.*

L'algorithme AHU est annoncé de complexité linéaire, mais c'est en prenant comme hypothèse que le nombre de fils d'un nœud est borné, indépendamment du nombre total n de l'arbre. Nous ne considérons pas cette hypothèse

Question 4.13.7 Écrire le pseudo-code de l'algorithme AHU.

Question 4.13.8 Calculer la complexité de l'algorithme pour la mesure de complexité adéquate.

Exercice 4.14 ***: Compaction d'arbre binaire de décision***

Question 4.14.1 En utilisant un trie binaire en tant que structure de données annexe, afin de reconnaître efficacement les arbres déjà vus, écrire un algorithme de compression d'arbres prenant en entrée un arbre de décision binaire et renvoyant le ROBDD associé.

Question 4.14.2 Calculer la complexité de l'algorithme, en supposant que l'arbre de décision binaire contient 2^k feuilles.

Question 4.14.3 Si la donnée en entrée n'est pas un arbre de décision binaire, mais un OBDD non réduit, comment adapter l'algorithme précédent afin de tirer parti du fait que l'entrée est partiellement compressée ?

Exercice 4.15 ***: Comptage dans un BDD***

Soit A un ROBDD dépendant de k variables ordonnées de x_1 à x_k. Pour un BDD B enraciné en un nœud interne x_i, on obtient son sous-BDD, lorsque $x_i = 0$, via $B.false$ et l'autre BDD, lorsque $x_i = 1$, via $B.true$.

Question 4.15.1 Donner un algorithme permettant de construire la table de vérité de la fonction associée à A. Quelle est sa complexité ?

Question 4.15.2 Donner un algorithme comptant le nombre d'affectations rendant la fonction associée à A vraie. Quelle est sa complexité ?

Question 4.15.3 Donner un algorithme construisant l'affectation $\mathbf{a}$
REPRENDRE : en regardant l'affectation comme un mot sur 0-1
la plus petite, dans l'ordre lexicographique, des variables $x_1, \ldots, x_k$ telle que $f(\mathbf{a}) = 1$.

Exercice 4.16 ***: Fusion de ROBDDs***

Question 4.16.1 Étant donnés deux ROBDDs A et B et (a, G_a, D_a) et (b, G_b, D_b) des nœuds de A et B. Rappeler quelle est la structure F fusion de A et B.

Question 4.16.2 Soient $f(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$ et $g(x_1, x_2, x_3, x_4) = (x_1 \texttt{ xor } x_2) \vee (x_3 \texttt{ xor } x_4)$ Représenter les ROBDDs de chacune des fonctions (en prenant l'ordre $x_1, \ldots, x_4$, inverse par rapport à celui vu en cours).

Question 4.16.3 Via l'opération de fusion, construire les BDD de $f \wedge g$ et $f \texttt{ xor } g$.