ALGO1 : Algorithmique – DM

Pour le 12 novembre 2019

Consignes

- Nous attendons des réponses argumentées et bien rédigées. N'hésitez pas à illustrer vos explications avec des schémas et des exemples.
- Inutile de perdre du temps à taper votre devoir sous LaTeX, si cela doit se faire au détriment des explications et des schémas illustratifs.
- Le sujet est long, et sera noté sur 25 points : il est possible d'atteindre une excellente note sans traiter tout parfaitement, mais faites au mieux et le plus possible !
- Le problème 3 demande que vous programmiez l'algorithme et que vous l'exécutiez sur différents fichiers textes. Nous vous encourageons fortement à utiliser Python (version 3.4 ou plus récente), et nous mettons à disposition un fichier squelette à remplir. Vous êtes néanmoins libre d'utiliser un des langages suivants : OCaml, C ou C++, Java, Javascript, Julia, mais nous insistons : utilisez Python, ce sera plus simple et plus formateur pour tout le monde.
- Le travail devra être effectué par **binôme** : vous êtes libres de choisir qui vous voulez. Il devra être rendu **au plus tard le 12 novembre 2019 avant 8h** à Lilian Besson **et** Raphaël Truffet (`lilian.besson`, `raphael.truffet@inria.fr`).
- Le plagiat (depuis Internet ou vos camarades) sera sévèrement puni. Nous voulons voir **votre implémentation, avec vos idées**. Nous serons stricts : jusqu'à 0/7 sur le problème 3, en cas de plagiat évident.
- Que vous choisissiez de rédiger sur papier libre et de numériser vos feuilles, ou de rédiger par LaTeX (ou un autre outil informatique), respectez la convention de nommage suivante :
  - vos réponses aux problèmes 1 à 3 (sauf code) doivent être **un seul document PDF**, nommé `DM_ALGO1__PRENOM1_NOM1__PRENOM2_NOM2.pdf`, contenant toutes les pages *dans l'ordre*,
  - votre code Python doit être un seul document Python, ou bien un notebook Jupyter (cf. `www.Jupyter.org`) (auquel cas, incluez le fichier `.ipynb` **et** le fichier Python exporté depuis le notebook), nommé `DM_ALGO1__PRENOM1_NOM1__PRENOM2_NOM2.py` (inutile de modifier ou d'inclure `heap_operations.py`),
  - votre code doit s'exécuter sans erreur ni avertissement, en moins de deux minutes pour un fichier texte brut de moins de 1 Mo, et accepter des noms de fichiers en argument donné par la ligne de commande (cf. problème 3 plus bas pour les détails),
  - tous les fichiers doivent être inclus dans une archive au format zip, nommée `DM_ALGO1__PRENOM1_NOM1__PRENOM2_NOM2.zip`.
  - Nous serons stricts : -2/20 de pénalité pour tout manquement à ces consignes simples. Et -2/20 de pénalité par jour de retard.

---

1 Problème 1 : Arbres non binaires

Dans ce devoir, nous nous intéressons à une famille particulière d'arbres, la famille $\mathcal{A}$. Les nœuds des arbres de $\mathcal{A}$ sont d'arité 2, 3 ou 4. Un nœud d'arité i ($i = 2..4$) comporte $i - 1$ clés et i arbres fils. Les feuilles ne contiennent pas de valeurs et seront notées $\emptyset$. La propriété d'arbre de recherche étudiée sur les arbres binaires s'étend naturellement à de tels arbres.

Par exemple, pour les nœuds d'arité 3, la première clé doit être comprise entre les valeurs des clés des sous-arbres gauches et du centre alors que la deuxième clé doit être comprise entre celles des sous-arbres du centre et de droite. Nous imposons de plus que la profondeur de chaque feuille soit la même. La figure 1 présente un exemple d'arbre de $\mathcal{A}$. Nous prendrons pour convention de ne pas faire figurer les feuilles.

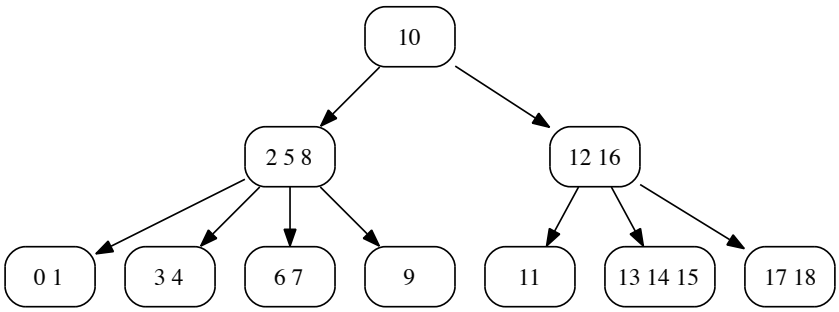

FIGURE 1: Un arbre de $\mathcal{A}$ comportant 19 éléments.

Q. 1 – Écrire un algorithme RECHERCHE(V,T) permettant de rechercher une clé V dans un arbre T de $\mathcal{A}$.
Donner une borne asymptotique sur le nombre de comparaisons effectuées par cet algorithme pour rechercher une clé dans un arbre comportant n clés (cas le pire).

Q. 2 – Expliquer, sans écrire de pseudo-code, comment lors d'une descente de la racine à une feuille donnée, il est possible de modifier cet arbre de façon à enlever tous les nœuds d'arité 4 le long de ce chemin. L'ensemble des clés présentes dans l'arbre ne devra pas changer et l'arbre résultant devra encore appartenir à $\mathcal{A}$.

Q. 3 – Expliquer, sans écrire de pseudo-code, comment insérer une clé dans un arbre en maintenant les propriétés de la famille $\mathcal{A}$. Donner une borne asymptotique sur le nombre d'opérations de comparaisons et création de nœud effectuées par cette méthode pour insérer une clé dans un arbre comportant n clés (dans le pire cas).

Q. 4 – Expliquer, sans écrire de pseudo-code, comment lors d'une descente de la racine à une feuille donnée, il est possible de modifier cet arbre de façon à enlever tous les nœuds d'arité 2 le long de ce chemin, sauf pour le nœud racine. L'ensemble des clés présentes dans l'arbre ne devra pas changer et l'arbre résultant devra encore appartenir à $\mathcal{A}$. Il n'est pas nécessaire de détailler tous les cas, s'ils sont symétriques avec des cas déjà présentés.

Q. 5 – Expliquer, sans écrire de pseudo-code, comment supprimer une clé dans un arbre en maintenant les propriétés de la famille $\mathcal{A}$. Donner une borne asymptotique sur le nombre d'opérations de comparaisons et création de nœud effectuées par cette méthode pour supprimer une clé dans un arbre comportant n clés (dans le pire cas).

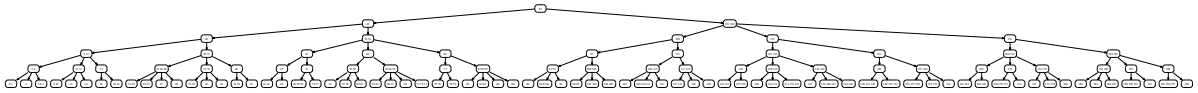

FIGURE 2: Un arbre de $\mathcal{A}$ comportant 200 éléments.

---

2 Problème 2 : Codage binaire préfixe

On s'intéresse au problème d'obtenir un encodage compact pour un texte. Dans cet exercice, on qualifiera donc d'alphabet un ensemble (fini) de symboles $\mathcal{A} = (s_i)_i$ auxquels sont associés leurs fréquences $(f(s_i))_i$ d'apparition dans le texte.

Exemple. $\mathcal{A}$ peut être l'alphabet `a,b,...,z,A,B,...,Z`, et les symboles de ponctuations en anglais (`,;:!?.()-\$@`), et les fréquences peuvent venir d'un comptage d'un texte du monde réel, comme un roman par exemple.

Codage. On appelle *codage binaire* (simplement nommé codage dans la suite) d'un alphabet $\mathcal{A}$ une fonction $c : \mathcal{A} \to \{0,1\}^*$. Le codage est dit *de longueur fixe* s'il existe $\ell \in \mathbb{N}$ tel que $c : \mathcal{A} \to \{0,1\}^\ell$.

Q. 6 – Évaluer la taille du codage d'un fichier de n caractères par un code de longueur fixe.

Nous cherchons naturellement à faire mieux, en utilisant un *codage de longueur variable*. Nous allons néanmoins ne considérer que des codages dit *préfixes* : un codage c est préfixe s'il n'existe pas deux symboles $s_1 \neq s_2 \in \mathcal{A}$ tels que $c(s_1)$ soit un préfixe de $c(s_2)$.

Q. 7 – Quel intérêt voyez-vous aux codages préfixes ?

Q. 8 – Le code $c_1 : A \mapsto 1, B \mapsto 0, C \mapsto 01$ sur l'alphabet $\mathcal{A} = \{A, B, C\}$ est-il préfixe ? Et le code $c_2 : A \mapsto 1, B \mapsto 01, C \mapsto 00$? Encoder le mot $BABA$ dans les deux codes. Essayer de décoder le mot 11010100 dans les deux codes.

Un codage va être représenté par un arbre dont les feuilles sont les caractères de l'alphabet : le chemin de la racine à une feuille définit son encodage selon la convention qu'un déplacement à gauche équivaut à un 0, un déplacement à droite équivaut à un 1. Nous identifierons dans la suite le codage à son arbre.

Q. 9 – Représenter le codage suivant sous forme d'arbre : $A \mapsto 0$, $B \mapsto 101$, $C \mapsto 100$, $D \mapsto 111$, $E \mapsto 1101$, $F \mapsto 1100$. Est-il préfixe ?

On peut ainsi définir le coût d'un codage comme suit, où $d_c(s)$ est la profondeur de la

feuille contenant le caractère s dans l'arbre de c (avec la convention que la racine a une profondeur égale à 1) : $B(c) = \sum_{s \in \mathcal{A}} f(s) * d_c(s) \in \mathbb{N}$.

Un codage est ainsi dit *optimal* si son coût est minimal. On rappelle qu'un arbre binaire est *complet* si tout nœud interne (c'est-à-dire qui n'est pas une feuille) a deux fils non vides.

Q. 10 – Montrer que tout codage optimal est un arbre binaire *complet*. Justifier qu'un codage optimal existe pour n'importe quel alphabet $(s_i)_i$ et fréquences $(f(s_i))_i$.

Q. 11 – Proposer un algorithme glouton permettant de construire l'arbre d'un codage optimal d'un alphabet.

On souhaite à présent prouver la correction de notre algorithme, c'est à dire que notre algorithme renvoie un codage optimal.

Q. 12 – Soient x et y deux caractères d'un alphabet $\mathcal{A}$ ayant une fréquence minimale dans $\mathcal{A}$. Montrer qu'il existe un codage préfixe optimal c pour $\mathcal{A}$ dans lequel $c(x)$ et $c(y)$ ont même longueur et ne diffèrent que par le dernier bit.

Q. 13 – Soient x et y deux caractères d'un alphabet $\mathcal{A}$ ayant une fréquence minimale dans $\mathcal{A}$. Soit $\mathcal{A}'$ l'alphabet obtenu en remplaçant x et y de $\mathcal{A}$ par un nouveau caractère z, de fréquence $f(z) = f(x) + f(y)$.

Soit c' un codage optimal pour $\mathcal{A}'$, montrer que le codage c obtenu en remplaçant le nœud feuille de z par un nœud interne ayant x et y pour enfants est optimal pour $\mathcal{A}$.

Q. 14 – Conclure. Ce codage s'appelle le codage de Huffman [1].

Q. 15 – Quel est le codage de Huffman d'un alphabet dont les fréquences décrivent les termes de la suite de Fibonacci ?

---

3 Problème 3 : Implémentation et exemples de compression par le codage de Huffman

Nous vous encourageons fortement à utiliser Python (version 3.4 ou plus récente), et nous mettons à disposition un fichier squelette à remplir :
`perso.crans.org/besson/teach/info1_algo1_2019/DM/compress_huffman.py`

Ce fichier contient des trous, des fois de plusieurs lignes, indiqués entre des balises "`# XXX <-- à remplir d'ici`" à "`# à là --> XXX`", et des fois des morceaux de codes sur une seule ligne sont à remplir (indiqués avec "`...`").

Vous pouvez aussi télécharger le fichier suivant qui implémente les opérations de files de priorités min (`heappop`, `heappush` [2]), si vous en avez besoin :
`perso.crans.org/besson/teach/info1_algo1_2019/DM/heap_operations.py`

Q. 16 – Remplir les trous dans la fonction `naive_code`, qui implémente *un* codage naïf à

1. Il a été inventé par David A. Huffman, et publié en 1952 durant sa thèse au MIT. Ça vous dirait vous, de léguer un tel résultat à la postérité pendant votre thèse ? C'est tout à fait possible ! !

2. Usage : `heappush(priority_queue, (priority, item))` pour insérer un nouvel élément, et `priority, item = heappop(priority_queue)` pour extraire l'élément de priorité minimum.

longueur fixée suivant la question 6.

Q. 17 – Remplir les trous dans la fonction `huffman_code` et `extract`, suivant l'algorithme mis au point entre les questions 10 et 13.

Q. 18 – Remplir les trous dans la fonction `compress`. Essayer de trouver une implémentation concise et simple (ça peut se faire en une ou quelques lignes).

Q. 19 – Remplir les trous dans la fonction `un_compress`. Expliquer l'algorithme choisi, et il vaut mieux donner une implémentation simple et un peu verbeuse qu'une implémentation efficace à laquelle on ne comprendra rien.

Q. 20 – La fin du fichier squelette `compress_huffman.py` fournit permet de lire un fichier texte (encodé en UTF-8 ou en ASCII) dont on donne le nom depuis la ligne de commande, et de compresser son contenu, avec les codages naïf et de Huffman. À la fin, le taux de compression[3] est affiché pour le fichier donné en argument.

Tester avec un petit fichier de votre choix, et avec le code source du fichier `compress_huffman.py` lui-même. Par exemple, on doit obtenir cela (avec des morceaux `...` non inclus ici) :

```
$ python compress_huffman.py compress_huffman.py
For the file compress_huffman.py ...
For the input text of length = 7499
The 20 most common letters are = {' ': 1646, 'e': 611, ...}

Gives naive code = {' ': '0001011', 'e': '1001001', ...}
The naive code gave a binary representation of length = 52493

Gives Huffman code = {' ': '01', 'e': '1110', ...}
The huffman code gave a binary representation of length = 35705

For the file compress_huffman.py
==> That's a compression ratio of 68.0%
```

Q. 21 – Trouver un texte intégral de deux œuvres classiques, les plus longues possibles, en français et en anglais[4], et reporter dans votre devoir les longueurs des fichiers (en terme de nombres de symboles), les longueurs des représentations textuelles binaires (ie, une `str` contenant des 0 et des 1) obtenues avec les deux code naïf et de Huffman, ainsi que le taux de compression obtenu.

Note : pensez à inclure ces fichiers textes dans l'archive zip de votre devoir, nous voudrons reproduire vos résultats !

Bonus : +0.5/20 au binôme avec le texte le plus long, +0.5/20 au binôme avec le meilleur taux de compression.

Q. 22 – Bonus : observe-t-on une différence entre un texte en français et en anglais ? Est-elle significative ? Si oui, proposer une interprétation.

---

3. Le taux de compression peut correspondre soit au rapport de la longueur finale divisée par la longueur initiale, ou à un moins ce rapport. Dans le code squelette fourni pour le DM, la première formule est utilisée : si un fichier en entrée utilise 100 symboles binaires lorsqu'il est représenté avec le codage naïf, et 68 avec le codage de Huffman, on dira que ce codage permet d'obtenir un *taux de compression* de $689/1000 \simeq 68\%$.

4. On peut notamment en trouver sur le site `www.Gutenberg.org`.