ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

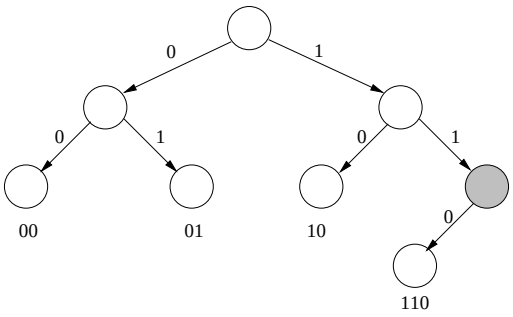
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 8 14 Nov 2011

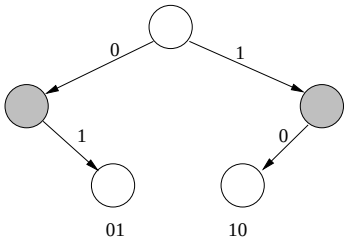
1. Le codage de Huffman

Le codage de Huffman est une méthode de construction d'un code optimal pour un alphabet avec des fréquences données. Autrement dit, le codage de Huffman nous fournit un code tel que la longueur moyenne des mots est minimale pour l'alphabet et les fréquences données.

- Le code $(0, 10, 11)$ est un code de Huffman pour l'alphabet (a, b, c) et les fréquences $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$.
- Le code $(00, 01, 10, 110)$ n'est pas un code de Huffman: en dessinant l'arbre correspondant, on voit qu'il y a un sommet interne qui n'a pas deux fils, ce qui n'arrive pas, par construction. Voici l'arbre correspondant à ce code avec le sommet en question dessiné en gris:



- De même, le code $(01, 10)$ n'est pas de Huffman. L'arbre correspondant, avec les sommet internes n'ayant pas deux fils marqués en gris:



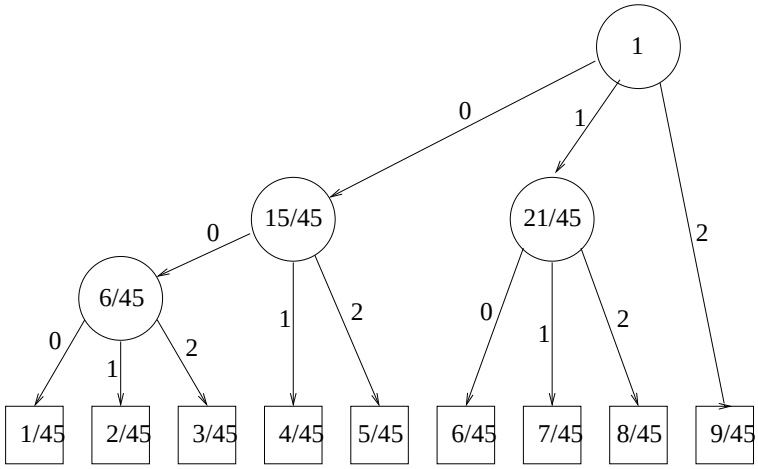
2. Codes Huffman ternaires

a) La construction de codes de Huffman ternaires est similaire à celle de codes de Huffman binaires: on utilise la même procédure, sauf qu'on combine à chaque pas trois sous-arbres au lieu de deux.

Cependant, il y a une subtilité à laquelle il faut faire attention: Il faut s'assurer qu'on puisse toujours combiner trois sous-arbres jusqu'à la fin (i.e., jusqu'à ce qu'il ne reste

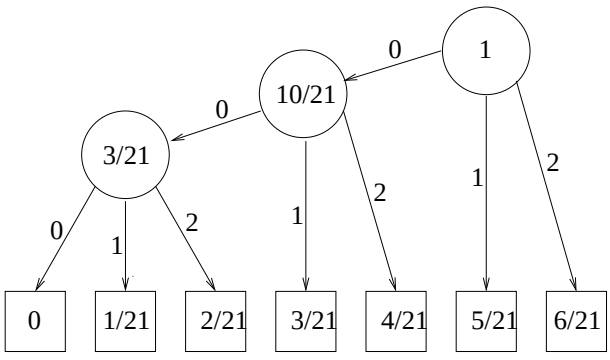
qu'un seul arbre). Il faut donc qu'il reste dans la dernière itération trois arbres et non deux. Pour s'en assurer de, il suffit d'ajouter une lettre de fréquence 0 à l'alphabet au début si le nombre de sommets est pair et d'ainsi s'assurer que l'alphabet est toujours de taille impaire: en combinant trois sous-arbres en un, on réduit à chaque pas le nombre d'objets à traiter par deux, et le fait que le nombre d'objets qu'on traite est impair est alors présérvé pendant que l'algorithme itère.

b) Le arbre suivant est l'arbre d'un code de Huffman C_1 ternaire avec les fréquences données.

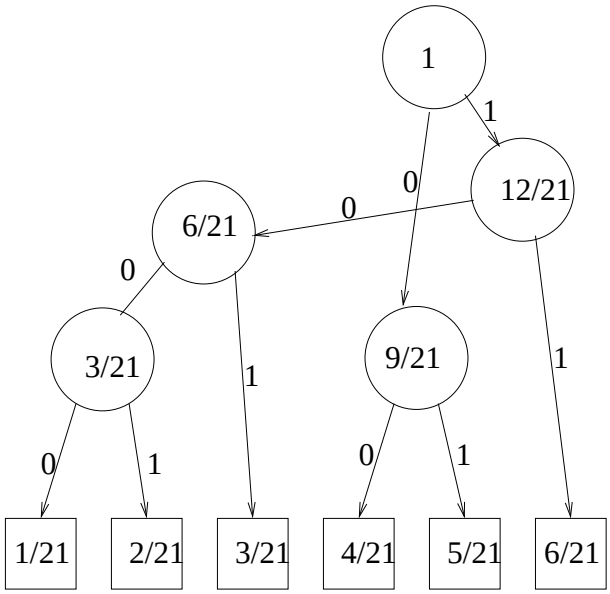


Les longueurs des mots du code sont $(3, 3, 3, 2, 2, 2, 2, 2, 1)$.

c) C'est un des cas pour lesquels il faut ajouter une lettre avec la fréquence 0 dans l'alphabet, comme mentionné au point a) ci-dessus. Appelons ce code C_2. Voici l'arbre:



Appelons C_3 le code de Huffman binaire pour les fréquences données. L'arbre correspondant:



d) On calcule la longueur moyenne de C_1, C_2, et C_3.

$$\begin{aligned}
\bar{L}_1 &= 1 \cdot \tfrac{9}{45} + 2 \cdot (\tfrac{8}{45} + \tfrac{7}{45} + \tfrac{6}{45} + \tfrac{5}{45} + \tfrac{4}{45}) + 3 \cdot (\tfrac{3}{45} + \tfrac{2}{45} + \tfrac{1}{45}) = \tfrac{29}{15} \\
\bar{L}_2 &= 1 \cdot (\tfrac{6}{21} + \tfrac{5}{21}) + 2 \cdot (\tfrac{4}{21} + \tfrac{3}{21}) + 3 \cdot (\tfrac{2}{21} + \tfrac{1}{21}) = \tfrac{34}{21} \\
\bar{L}_3 &= 2 \cdot (\tfrac{6}{21} + \tfrac{5}{21} + \tfrac{4}{21}) + 3 \cdot \tfrac{3}{21} + 4 \cdot (\tfrac{2}{21} + \tfrac{1}{21}) = \tfrac{51}{21}
\end{aligned}$$

3. Shell Sort

a) Nous utilisons Shell Sort avec les incréments $2, 1$. Nous allons donc procéder en deux étapes: Nous allons d'abord réordonner la suite pour qu'elle soit 2-triée, puis nous allons réordonner cette suite 2 triée pour qu'elle soit 1-triée (c'est-à-dire triée au sens habituel du terme).

Informellement la première étape consiste à trier les deux sous suites $(a[0], a[2], a[4])$ et $(a[1], a[3], a[5])$ (pour que la suite totale soit bien 2-triée). Chacune de ces sous-suite est triée en utilisant Insertion Sort.

Etape de l'incrément 2: Notre suite est dans son état initial:

$$(4, 3, 2, 6, 1, 5)$$

Nous commençons comme décrit dans l'algorithme avec $i = h$, donc ici $i = 2$. Nous comparons $a[2]$ (2) avec $a[0]$ (4), et puisque $a[2]$ est plus petit nous échangeons ces deux éléments:

$$(2, 3, 4, 6, 1, 5)$$

(nous avons en fait appliqué la première étape de Insertion Sort à la sous-suite $(a[0], a[2], a[4])$).

Ensuite nous passons à $a[3]$ (6), que nous comparons avec $a[1]$ (3) (c'est la première étape de Insertion Sort sur $(a[1], a[3], a[5])$. Puisque $a[1] < a[3]$ $(3 < 6)$ nous n'avons rien à faire.

Nous continuons avec l'élément $a[4]$ (1). Nous voulons faire la deuxième étape de Insertion Sort sur $(a[0], a[2], a[4])$ $((2, 4, 1))$, c'est à dire insérer $a[4]$ (1) à la bonne position dans $(a[0], a[2])$ $((2, 4))$. Nous le comparons à $a[2]$ (4), il est plus petit, donc nous continuons, nous le comparons à $a[0]$ (2), il est encore plus petit et nous savons donc qu'il doit être le premier élément dans cette sous-suite. Pour le placer en premier nous devons décaler $a[0]$ et $a[2]$ vers la droite, on obtient donc:

$$(1, 3, 2, 6, 4, 5)$$

finalement nous regardons l'élément $a[5]$ (5). Nous voulons le placer à la bonne position dans $(a[1], a[3])$ $((3, 6))$. Après deux comparaisons nous voyons qu'il doit aller au milieu (puisque $a[1] < a[5] < a[3]$, i.e. $3 < 5 < 6$), on a donc:

$$(1, 3, 2, 5, 4, 6)$$

Et nous avons fini cette étape, la suite est bien 2-triée.

Etape de l'incrément 1: Nous faisons maintenant un Insertion Sort normal. Nous commençons avec $a[1]$ (3) que nous comparons à $a[0]$ (1) et voyons qu'aucun mouvement n'est nécessaire. Ensuite nous regardons $a[2]$ (2) , il faut le placer à la bonne position dans $(a[0], a[1])$, c'est à dire au milieu:

$$(1, 2, 3, 5, 4, 6)$$

Nous continuons avec $a[3]$ (5) qu'il faut mettre à la bonne position dans $(a[0], a[1], a[2])$, il n'y a donc rien à faire. Nous regardons ensuite $a[4]$ (4), et voyons qu'il faut le mettre entre $a[2]$ (3) et $a[3]$ (5):

$$(1, 2, 3, 4, 5, 6)$$

finalement nous regardons $a[5]$ (6) et voyons qu'il est à la bonne place. Nous avons terminé et notre suite est bien triée.

Pour le nombre de mouvements, nous les comptons en supposant qu'un échange nécessite 3 mouvements ($v[1]$ vers $temp$, $v[2]$ vers $v[1]$, $temp$ vers $v[2]$).

Pour passer de $(2, 3, 4, 6, 1, 5)$ à $(1, 3, 2, 6, 4, 5)$, l'algorithme place d'abord $a[4]$ dans une variable temporaire (mouvement 1), puis il déplace $a[2]$ dans $a[4]$ (mouvement 2), puis il déplace $a[0]$ dans $a[2]$ (mouvement 3), et finalement il met la valeur temporaire dans $a[0]$ (mouvement 4). Pour ce passage il a donc fallu 4 mouvements.

Tous les autre changements sont de simples échanges d'éléments dans la liste. Comme il y a en a 4, on obtient un total de $4 \times 3 + 4 = 16$ mouvements.

b) Soit a une suite 2-triée et 3-triée. Elle peut être triée en utilisant l'algorithme suivant:

Call: SORT(a)
Input: Suite a 2-triée et 2-triée de N éléments avec des clés entières
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N - 1$

```
1: for i = 1, ..., N − 1 do
2:     if a[i] < a[i − 1] then
3:         Echanger a[i] et a[i − 1]
```

Il s'agit d'un passage de Bubble Sort. Comme souvent, on voit intuitivement que cet algorithme devrait marcher, mais les détails d'une preuve formelle sont plus compliqués:

Preuve que cet algorithme marche: Remarquons d'abord que comme la suite est 2 triée, pout tout k on a

$$a[k] \leq a[k+2] \leq a[k+4] \leq \dots \tag{1}$$

De même, puisque la suite est 3-triée on a $a[k] \leq a[k+3]$, et comme elle est 2-triée on obtient:

$$a[k] \leq a[k+3] \leq a[k+5] \leq a[k+7] \leq \dots \tag{2}$$

En combinant (1) et (2) on obtient

$$a[k] \leq a[k+2], a[k+3] \dots, a[N-1] \tag{3}$$

Nous voyons dans l'algorithme que la variable i va de 1 a $N-1$. Nous allons montrer par induction qu'après l'itération i les éléments $a[0], \dots, a[i-1]$ sont tous a la bonne place.

base: $i = 1$. Si on pose $k = 0$ dans (3) on voit que $a[0]$ est plus petit que tous les éléments, sauf $a[1]$ (on ne sait pas lequel de $a[0]$ et $a[1]$ est le plus grand).

• Si $a[0] \leq a[1]$ alors $a[0]$ est bien le plus petit élément de la suite, et on voit aussi que la première iteration de l'algorithme ne va rien changer, le plus petit élément sera donc bien à la première place.

• Si $a[1] < a[0]$ alors on a $a[1] < a[0] \leq a[2], \dots, a[N-1]$, et donc $a[1]$ est le plus petit élément de la suite. On voit aussi que la première itération de l'algorithme va échanger $a[0]$ et $a[1]$, on aura donc bien le plus petit élément de la suite à la première place.

Pas: On suppose qu'après l'itération i, $a[0], \dots, a[i-1]$ sont tous à la bonne place, et on veut montrer qu'après l'itération $i+1$ $a[0], \dots, a[i]$ seront tous à la bonne place.

Il faut donc montrer que le plus petit élément de $\{a[i-1], \dots, a[N-1]\}$ va être mis à la position $a[i-1]$.

On avait avant de commencer l'algorithme $a[i-1] \leq a[i+1], \dots, a[N-1]$. Cependant, il se peut qu'après l'itération i un nouvel élément se trouve à la position $a[i-1]$ (par contre $a[i], \dots, a[N-1]$ sont les mêmes). Mais le nouvel élément qui se trouve à la position $a[i-1]$ est un des éléments qui se trouvaient avant le début de l'algorithme dans $a[0], \dots a[i-1]$, donc la proposition suivante reste vraie:

$$a[i-1] \leq a[i+1], a[i+2] \dots, a[N-1] \tag{4}$$

• Si $a[i-1] \leq a[i]$, on voit en utilisant (4) que $a[i-1]$ est le plus petit élément de $a[i-1], \dots, a[N-1]$ il est donc à la bonne place (puisque $a[0], \dots, a[i-2]$ sont tous à la

bonne place il doit venir juste après). Et on voit on effet que l'algorithme ne modifie pas la liste.

• Si $a[i] < a[i-1]$, en utilisant (4) on voit que $a[i] < a[i-1] \leq a[i+1], a[i+2] \ldots, a[N-1]$. Donc $a[i]$ est le plus petit élément de $a[i-1], \ldots, a[N-1]$, il faut le mettre à la place de $a[i-1]$ (juste après $a[i-2]$), et c'est bien ce que fait l'algorithme.

Finalement, puisque l'algorithme fait $N-1$ itérations, et qu'il y a au plus un échange dans chaque itération, le nombre d'échanges sera au plus $N-1$. En fait il n'y a aucun input pour lequel $N-1$ échanges seront nécessaires, c'est une borne supérieure.

4. Nombre moyen d'changes avec Selection Sort

a) Voici le tableau d'changes s'il n'y a que trois lments:

permuation	nombre d'changes
012	0
021	1
102	1
120	2
201	2
210	1

En effet, 012 est dj ordonn donc aucun change n'est ncessaire. Pour 021 il suffit d'changer 1 et 2, pour 102 on change 0 et 1. Pour 120 on change d'abord 0 et 1, puis 1 et 2, donc deux changes sont ncessaires. Et ainsi de suite.

Si toutes ces permutations ont la mme probabilit, le nombre moyen d'changes sera

$$\frac{0+1+1+2+2+1}{6} = \frac{7}{6}.$$

b) (i) Il y a $k!$ permutations de $0, \ldots, k-1$ au total (il y a k lments). Nous voulons compter dans combien d'entre elles 0 est la bonne place. Pour construire une permutation dans laquelle 0 est la bonne position, on le place d'abord en premier, puis on peut placer les $k-1$ lments qui restent de n'importe quelle façon (donc selon n'importe laquelle des $(k-1)!$ permutations possibles). Ainsi la probabilit que 0 est la bonne place est

$$\frac{(k-1)!}{k!} = \frac{(k-1)\cdots 2\cdot 1}{k\cdot(k-1)\cdots 2\cdot 1} = \frac{1}{k}.$$

(ii) Comme expliqu dans la question, SelectionSort fait $N-1$ itrations pour i allant de 0 $N-2$. Pendant l'itration i on fait un change si et seulement si le i^{me} lment n'est pas la bonne place.

Pour la premire itration ce sera la cas avec probabilit $1-\frac{1}{N}$ (d'aprs la question (i), o ici on a N lments donc $k=N$).

Le nombre moyen d'changes durant la premire itration est donc

$$1 \cdot \left(1 - \frac{1}{N}\right) + 0 \cdot \frac{1}{N} = 1 - \frac{1}{N} = \frac{N-1}{N}.$$

De mme, au dbut de la i^{me} itration on sait que les lments $0, \ldots, i-1$ sont tous la bonne position. Il reste donc $N-i$ lments disposs de façon alatoire dans la liste. Il faudra faire un change si et seulement si l'lment i n'est pas la bonne position, ce qui arrive avec probabilit $1 - \frac{1}{N-i}$ (question (i) avec $N-i$ lments). Comme ci-dessus, le nombre moyen d'changes pendant la i^{me} itration est donc

$$\frac{N-i-1}{N-i}.$$

Finalement, le nombre moyen d'changes au total est gal la somme des nombres moyens d'changes chaque itration, donc

$$\frac{N-1}{N} + \frac{N-2}{N-1} + \ldots + \frac{2}{3} + \frac{1}{2}.$$

5. Tri par échanges adjacents

a) On fait par itération autant d'échanges que l'autre algorithme fait des mouvements. Comme un échange est plus cher, cet algorithme est moins vite que celui du cours, mais la différence n'est que d'un facteur constant; en l'ordre les algorithmes sont pareils.

b) On a l'galit suivante d'venements:

$$\{|\sigma(i) - i| = j\} = \{\sigma(i) \in \{i+j, i-j\}\} = \{\sigma(i) = i+j\} \cup \{\sigma(i) = i-j\}.$$

Au moins l'une des quantités $i+j$ et $i-j$ appartient à $\{1, \ldots, N\}$. Si par exemple $i+j \in \{1, \ldots, N\}$ on a alors

$$P(|\sigma(i) - i| = j) \geq P(\sigma(i) = i+j).$$

Comme σ est uniformément choisi, on a $P(\sigma(i) = k) = 1/N$ pour tout k tel que $1 \leq k \leq N$, d'où le résultat.

c) En utilisant le résultat du point précédent, on calcule

$$\begin{aligned} E[|\sigma_i - i|] &\geq \sum_{j=0}^{\lfloor N/2 \rfloor} P(|\sigma_i - i| = j) \cdot j \\ &\geq \sum_{j=0}^{\lfloor N/2 \rfloor} \frac{1}{N} \cdot j \\ &= \frac{1}{N} \cdot \frac{(\lfloor N/2 \rfloor - 1)\lfloor N/2 \rfloor}{2} \\ &\geq \frac{1}{N} \cdot \frac{(N/2-2)(N/2-1)}{2} \\ &= \frac{1}{N} \cdot \frac{(N-4)(N-2)}{8} \\ &= \frac{N^2 - 6N + 8}{8N} \\ &\geq \frac{N}{9}, \end{aligned}$$

où la dérnière inégalité n'est vraie que si N est assez grand.

d) Comme chaque transposition (échange adjacent) ne peut approcher un élément de sa position finale de 1, il faut pour chaque élément $\Omega(N)$ transpositions touchant l'élément en question. Une transposition touche au plus (en fait, exactement) deux éléments. Donc il faut effectuer

$$N\frac{\Omega(N)}{2} = \Omega(N^2)$$

transpositions en moyenne.