Correction

Examen d'algorithmique

EPITA ING2 2009 Rattrapage S1; A. DURET-LUTZ

Durée : 1 heure 30

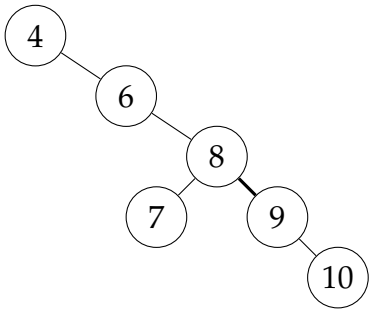
Juillet 2008

Consignes

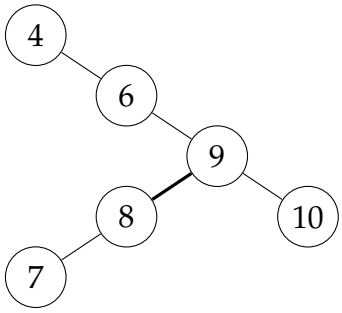
- Tous les documents sur papier sont autorisés (livres, notes de cours, polycopiés...).
  Les calculatrices, téléphones, PSP et autres engins électroniques ne le sont pas.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a 5 pages. Rappelez votre nom en haut de chaque feuille au cas où elles se mélangeraient.
- Ne donnez pas trop de détails. Lorsqu'on vous demande des algorithmes, on se moque des points-virgules de fin de commande etc. Écrivez simplement et lisiblement. Des spécifications claires et implémentables sont préférées à du code C ou C++ verbeux.
- Le barème est indicatif et correspond à une note sur 30.

Rotation d'arbre binaire de recherche (2 point)

Redessinez l'arbre binaire de recherche suivant après avoir effectué une rotation gauche de l'arc (8)—(9).



Réponse :



Arbres ternaires (6 points)

On considère ici des arbres ternaires, c'est-à-dire des arbres dont chaque nœud possède soit 3 fils (on parle alors de nœud interne) soit aucun fils (on parle alors de feuille).

La *profondeur* d'un nœud dans un arbre est la longueur du chemin qui le relie à la racine. La racine est à la profondeur 0, ses fils à la profondeur 1, ses petits-fils à la profondeur 2, etc. La *hauteur* d'un arbre est la profondeur de sa feuille la plus profonde.

Soyez précis dans vos réponses aux questions qui suivent. En particulier, si vous utilisez une partie entière choisissez bien entre partie entière $\lceil$supérieure$\rceil$ ou $\lfloor$inférieure$\rfloor$.

1. **(1 point)** Quel est le nombre maximal de feuilles que peut posséder un arbre ternaire de hauteur $h \geq 0$?

Réponse :

$$3^h$$

2. **(2 point)** Quel est le nombre maximal de nœuds que peut posséder un arbre ternaire de hauteur $h \geq 0$?

Réponse :

$$\sum_{i=0}^{h} 3^i = \frac{3^{h+1}-1}{2}$$

3. **(2 point)** Quelle est la hauteur minimale que peut atteindre un arbre ternaire de $n > 0$ nœuds ?

Réponse :

C'est une conséquence de la réponse précédente. La hauteur minimale est obtenue en trouvant le h minimal qui permet de stocker au plus n nœuds. On doit donc tirer h de :

$$n = \frac{3^{h+1}-1}{2}$$

Après passage au $\log_3$ et comme h est entier, on obtient

$$h = \lceil \log_3(2n+1) - 1 \rceil$$

4. **(1 point)** Quelle est la hauteur maximale que peut atteindre d'un arbre ternaire de $n > 0$ nœuds ?

Réponse :

$$\lceil \frac{n-1}{3} \rceil$$

Complexité (8 points)

1. **(1 point)** Ai-je le droit d'écrire $\Omega(f(n)) \cap \mathrm{O}(f(n)) = \Theta(f(n))$?

Réponse :

Évidemment !

2. **(2 points)** Est-il vrai que toutes les fonctions de la forme $f(n) = Kn^P$ avec K et P constants, sont dans la classe $\mathrm{O}(2^n)$? Justifiez votre réponse brièvement.

Réponse :

Oui, 2^n croît beaucoup plus vite que n^P.

3. **(3 points)** La complexité $T(n)$ d'un algorithme est constante pour une entrée de taille $n \leq 2$ et vérifie la relation $T(n) = 3T(n/2) + n \log n$ pour $n > 2$. À quelle classe de complexité cet algorithme appartient-il ?
(Pour information : $\log_2(3) \approx 1.585$, $\log_3(2) \approx 0.631$, $\sqrt{2} \approx 1.414$, $\sqrt{3} \approx 1.732$, $\sqrt[3]{2} \approx 1.26$, $\sqrt[3]{3} \approx 1.442$.)

Réponse :

On applique le théorème général.
$a = 3$, $b = 2$. On doit comparer la fonction $f(n) = n \log n$ avec $n^{\log_b a} = n^{\log_2(3)}$.
On a $f(n) = \mathrm{O}(n^{1.5}) = \mathrm{O}(n^{\log_2(3)-\varepsilon})$ avec $\varepsilon = \log_2(3) - 1.5 > 0$.
Le théorème général s'applique donc et on en déduit que $T(n) = \Theta(n^{\log_2(3)})$.

4. **(2 points)** Je possède deux algorithmes A et B pour résoudre un même problème (peu importe de quel problème il s'agit).

   Selon les données passées en entrée, ces algorithmes vont avoir des complexités différentes. Pour certains jeux de données l'algorithme A tournera en $\Theta(n)$ et l'algorithme B en $\Theta(n^2)$. Pour tous les autres jeux de données, ce sera le contraire : l'algorithme A tournera en $\Theta(n^2)$ et l'algorithme B en $\Theta(n)$.

   Pour un jeu de données particulier il n'est pas possible de prévoir comment vont se comporter les algorithmes, et donc de choisir à l'avance l'algorithme le plus efficace à utiliser.

   Comment feriez-vous pour résoudre le problème avec une complexité $\Theta(n)$ systématiquement ?

Réponse :

On exécute les deux algorithmes A et B en parallèle. Dès que le plus rapide des deux algorithmes a trouvé la réponse, on interrompt l'autre.

Mise en forme de paragraphes (14 points)

On s'intéresse au problème du formatage d'un paragraphe sur une largeur de taille fixe.

Dans notre cas un paragraphe est une suite de mot séparés par des espaces. Les retours à la ligne ne peuvent se faire qu'à la place d'une espace. La largeur, c'est-à-dire le nombre de caractère maximal que peut recevoir une ligne, sera indiquée par un entier supposé plus grand que tous les mots du paragraphe.

Les exemples qui suivent donnent deux formatages sur $L = 20$ colonnes de la chaîne « `The usual way in which we plan today for tomorrow is in yesterday's vocabulary.` » Il en existe d'autres.

```
12345678901234567890      12345678901234567890
====================      ====================
The usual way in          The usual way
which we plan today       in which we plan
for tomorrow is in        today for tomorrow
yesterday's               is in yesterday's
vocabulary.               vocabulary.
====================      ====================
```

Comme on considère une police de caractères à chasse fixe, la valeur de chaque caractère n'est pas importante, seul leur nombre l'est. On peut donc représenter le paragraphe à formater par un tableau contenant les longueurs de ses mots. Ainsi la chaîne de notre exemple peut être représentée par $A = [\ 3 \mid 5 \mid 3 \mid 2 \mid 5 \mid 2 \mid 4 \mid 5 \mid 3 \mid 8 \mid 2 \mid 2 \mid 11 \mid 11\ ]$. Notez que tous les caractères qui ne sont pas des espaces sont considérés comme faisant partie d'un mot. Nous ne découperons jamais un mot.

Un formatage du paragraphe peut alors être représenté par un tableau indiquant le numéro du dernier mot de chaque ligne. Ici $[\ 4 \mid 8 \mid 12 \mid 13 \mid 14\ ]$ pour l'exemple de gauche, ou $[\ 3 \mid 7 \mid 10 \mid 13 \mid 14\ ]$ pour celui de droite.

Un algorithme de formatage prend en entrée un paragraphe (c'est-à-dire un tableau de tailles de mots) ainsi qu'une largeur entière, et produit en sortie un formatage (c'est-à-dire un tableau d'indices de derniers mots).

1. **(3 points)** Écrivez le pseudo-code d'un algorithme `GreedyFormat(A,L)` qui formate le paragraphe A sur une largeur L en remplissant les lignes de haut en bas en y insérant le maximum possible de mots. (L'exemple de gauche a été produit par cet algorithme.)

Réponse :

En supposant les tableaux commençant à l'indice 1.

```
GreedyFormat(A,L):
  ligne = 1            # numéro de la ligne en cours
  reste = L            # blancs restants sur la ligne
  foreach mot in 1..len(A):
    if reste < A[mot]: # le mot déborde, on passe à la ligne suivante
       res[ligne] = mot - 1
       ligne = ligne + 1
       reste = L
    reste = reste - A[mot] - 1
  res[ligne] = len(A)  # On n'oublie pas la dernière ligne
  return res
```

2. **(3 points)** On souhaite minimiser les longs blancs en bout de ligne. On appellera « coût d'une ligne » le carré du nombre d'espaces à droite de la ligne et « coût du formatage » la somme des coûts de chaque ligne. Dans notre exemple c'est le formatage de droite qui a le moindre coût, car il possède moins de blancs importants.

12345678901234567890		12345678901234567890	
The usual way in	$c(1,4) = 4^2 = 16$	The usual way	$c(1,3) = 7^2 = 49$
which we plan today	$c(5,8) = 1^2 = 1$	in which we plan	$c(4,7) = 4^2 = 16$
for tomorrow is in	$c(9,12) = 2^2 = 4$	today for tomorrow	$c(8,10) = 2^2 = 4$
yesterday's	$c(13,13) = 9^2 = 81$	is in yesterday's	$c(11,13) = 3^2 = 9$
vocabulary.	$c(14,14) = 9^2 = 81$	vocabulary.	$c(14,14) = 9^2 = 81$
	total : 183		total : 159

En fonction du tableau A et de la largeur L, donnez une définition mathématique de $c(i,j)$: le coût d'une ligne qui contiendrait les mots de A entre les indices i et j inclus. On conviendra que $c(i,j) = \infty$ lorsque les mots de i à j ne tiennent pas sur la ligne. N'oubliez pas de compter les espaces entre les mots.

Réponse :

$$c(i,j) = \begin{cases} \infty & \text{si } L < (j-i) - \sum_{k=i}^{j} A[k] \\ \left(L - (j-i) - \sum_{k=i}^{j} A[k]\right)^2 & \text{sinon} \end{cases}$$

3. **(3 points)** Donnez une définition mathématique (et récursive) de $f(n)$: le coût minimal du formatage des n premiers mots du paragraphe. Sur notre paragraphe d'exemple, $f(14) = 159$. Par convention on supposera que $f(0) = 0$.

Réponse :

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ \min_{i \in [\![0, n-1]\!]} \big(f(i) + c(i+1, n)\big) & \text{sinon} \end{cases}$$

4. **(5 points)** Selon les principes de la programmation dynamique, écrivez le pseudo-code d'un algorithme `Format(A,L)` produisant un formatage de coût $f(|A|)$ en $O(|A|^2)$ opérations. L'exemple de droite a été formaté par cet algorithme. Vous pourrez supposer que vous disposez d'un tableau `C[i][j]` contenant les valeurs de $c(i,j)$ précalculées en fonction de A et L.

Réponse :

```
Format(A,L):
  # On calcule tous les c(i,j).  Vous pouviez supposer que
  # cela était déjà fait.  Je le mets tout de même dans la
  # correction pour montrer que cela est faisable en
  # O(|A|*|A|) opérations.
  foreach i in 1..len(A):
    taille = A[i]
    C[i][i] = (L-taille)*(L-taille)
    foreach j in i+1..len(A):
      taille = taille + 1 + A[j]
      if (L >= taille):
        C[i][j] = (L-taille)*(L-taille)
      else:
        C[i][j] = MAX_INT
  # On remplit itérativement le tableau F[n] avec
  # la formule f(n) ci-dessus.  Au passage on sauvegarde
  # le tableau best[n] les indices i correspondant aux
  # découpages de coûts minimaux.
  F[0] = 0
  foreach n in 1..len(A):
    minidx = 0   # indice du minimum
    minval = 0   # valeur du minimum
    # On saute 0 dans la boucle car minidx et
    # minval sont à jour pour le premier indice.
    foreach i in 1..(n-1):
      val = F[i] + c(i+1,n)
      if (val < minval):
        minidx = i
        minval = val
    best[n] = minidx
  # Maintenant on récupère les meilleurs découpages pour
  # produire le résultat.
  n = len(A)
  while (n > 0):
    res.push_left(n)
    n = best[n]
  return res
```