Partiel L3 - Algorithmique

Anne Benoit

Mercredi 26 Octobre 2005, 10h15-12h15

1 Tri de fusion aux 2/3

On considère un algorithme de tri qui fonctionne exactement comme le tri fusion, sauf que le tableau est coupé aux 2/3 au lieu d'être coupé à la moitié comme dans le tri usuel.

1. Ecrire l'algorithme `tri-2/3`. Expliquer succinctement la fonction de fusion utilisée.
2. Quelle est la complexité de ce tri ? Comparer avec les algorithmes de tri usuels.

Solution

1. L'algorithme de tri

tri-2/3(T, i, j)
si $(i < j)$ alors {
  tri-2/3$(T, i, i + \lfloor \frac{2(j-i)}{3} \rfloor)$
  tri-2/3$(T, i + \lfloor \frac{2(j-i)}{3} \rfloor + 1, j)$
  fusion(T, i, j)
}

La fonction de fusion fonctionne comme la fusion usuelle, sauf que l'on place le milieu du tableau à $\lfloor \frac{2(j-i)}{3} \rfloor$. On compare ensuite le premier élément de chaque partie du tableau... C'est de coût linéaire.

2. Complexité

Prouvons que $T(n) = 0(n \log n)$, i.e. il existe $c, n_0 > 0$ tels que pour tout $n > n_0$ $T(n) \leq c \times n \log n$.

La récurrence pour notre fonction est :

$$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n)$$

Le $O(n)$ représente le coût de la fusion.

Résolution de la récurrence :

- $T(1) = 1$ (évident)
- Supposons que pour tout $k < n$, $T(k) \leq c \times k \log k$.

$$\begin{aligned} T(n) &= T(\tfrac{2n}{3}) + T(\tfrac{n}{3}) + O(n) \\ &\leq c * (\tfrac{2n}{3}) \times \log(\tfrac{2n}{3}) + c \times (\tfrac{n}{3}) \times \log(\tfrac{n}{3}) + c_1 \times n \\ &\leq \tfrac{2}{3}cn \log n + \tfrac{2}{3}cn \log(\tfrac{2}{3}) + \tfrac{1}{3}cn \log n + \tfrac{1}{3}cnlog(\tfrac{1}{3}) + c_1 \times n \\ &= cn \log n + \tfrac{2}{3}cn \log 2 - \tfrac{2}{3}cn \log 3 + \tfrac{1}{3}cn \log 1 - \tfrac{1}{3}cn \log 3 + c_1 \times n \\ &= cn \log(n) + \tfrac{2}{3}cn - cn \log(3) + c_1 \times n \end{aligned}$$

On choisit c tel que $-c_1 \times n = \frac{2}{3}cn - cn \log(3)$, soit $c_1 = (log(3) - \frac{2}{3})c$.

Ainsi, $T(n) \leq c \times n \log n$ et on a prouvé la récurrence.

2 Algorithme de la collection complète

Pour promouvoir un article, une société offre des albums d'images à compléter. La collection complète est formée de N images différentes. Une image est fournie en cadeau avec chaque article acheté. On suppose que les images sont aléatoirement uniformément distribuées sur les articles, et les articles sont en stock illimité.

1. Proposer un algorithme pour obtenir une collection complète.
   On utilisera la procédure $x \leftarrow$ `achat()` qui retourne l'image x.
2. Combien d'achats faudra-t-il faire en moyenne pour avoir une collection complète ?
   Indication : On utilisera la variable $A_{i,N}$ correspondant au nombre moyen d'achats nécessaires à l'étape i pour obtenir une image x qui n'est pas déjà dans la collection. $A_{i,N}$ peut s'exprimer comme $\sum_j p_{i,j}$, où $p_{i,j}$ est la probabilité pour que le coût soit au moins j à l'étape i. Les $p_{i,j}$ pourront être calculés par récurrence sur j.

Solution

1. Algorithme

Tant qu'on n'a pas les N images, on achète un article jusqu'à ce qu'on trouve une image que l'on n'a pas. On la colle dans l'album, et on s'arrête si l'album est complet. Cela correspond à l'algorithme :

```
perm: Tab(1..N);  -- numéros des images déjà trouvées
pour i de 1 à N faire {
  x <- achat() tant que x appartient à perm(1..i-1);
  perm(i) := x;
}
```

2. Complexité en moyenne

- Calcul des $p_{i,j}$ par récurrence :
  $p_{i,1} = 1$ (à l'étape 1, 1 seul tirage)
  $p_{i,j+1} = p_{i,j} . \frac{i-1}{N}$
  ($i-1$ chances à chaque tirage pour tomber sur l'une des images que l'on a déjà)
  Donc $p_{i,j} = (\frac{i-1}{N})^{j-1}$.
- D'après la relation donnée dans l'énoncé,
  $A_{i,N} = \sum_j p_{i,j} = \sum_j (\frac{i-1}{N})^{j-1} = \frac{N}{N-i+1}$
- Le nombre d'achats moyens total est donc :
  $A_n = \sum_{i=1}^{N} A_{i,N} = N \sum_{i=1}^{N} \frac{1}{i} \sim N \log N$

On doit donc en moyenne acheter $N \log N$ articles pour compléter sa collection.

3 Problème des cageots de fraises

Nous nous intéressons à la distribution de n cageots de fraises dans p magasins. Les bénéfices que l'on peut retirer de chaque magasin est fonction du nombre de cageots fourni. Ainsi, $b_j(i)$ représente le bénéfice que tire le magasin j de la vente de i cageots ($1 \leq i \leq n$ et $1 \leq j \leq p$).

Nous appelons *gains marginaux* les gains supplémentaires obtenus par cageots :
$g_j(i) = b_j(i) - b_j(i-1)$, avec $b_j(0) = 0$ (aucun bénéfice pour la vente de 0 cageots).

1. Si, pour chaque magasin j, la fonction de gains marginaux g_j est décroissante, proposer un algorithme optimal. Donner un contre-exemple si les gains marginaux ne sont pas décroissants.
2. Combien de solutions au problème existent dans le cas général ?
3. Proposer un algorithme qui calcule le coût optimal pour répartir les cageots de fraises dans les magasins pour le cas général. Quelle est sa complexité ?

Solution

1. Gains décroissants

L'algorithme glouton est optimal : pour chaque cageot on regarde quel magasin possède le meilleur gain marginal (compte tenu du nombre de cageots qu'il possède déjà) et on donne le cageot à ce magasin là.

Par contre ça ne marche pas dans le cas général : si on a deux magasins et deux cageots, avec les coûts marginaux $g_1(1) = 4, g_1(2) = 7, g_2(1) = 5, g_2(2) = 5$, l'algorithme glouton donne les deux cageots au magasin 2 et produit un bénéfice de 10, alors que donner les deux cageots au magasin 1 aurait produit un bénéfice de 11.

2. Nombre de solutions

Il faut séparer les n cageots en p paquets, soit placer $p-1$ barres entre eux. Il y a donc C_{n+p-1}^{p-1} solutions, ce qui fait beaucoup !

3. Programmation dynamique

Pour résoudre le problème dans le cas général, on utilise donc la programmation dynamique, en faisant une récurrence sur p.

On note $A_p(n)$ le coût optimal pour répartir n cageots dans p magasins. Il est difficile de passer de n à $n+1$, d'où la récurrence sur p.

$$A_k(n) = Max_{i=0..n}(b_k(i) + A_{k-1}(n-i))$$

et le cas limite $A_1(n) = b_1(n)$.
On calcule dans l'ordre des p croissants, d'où l'algorithme :

$A_1[0..n]$ initialisé avec $b_1[0..n]$
Pour $k = 2$ à p
    Pour $i = 0..n$
        $A_k(i) = max_{j=0..i}\{A_{k-1}(i-j) + b_k(j)\}$;
Renvoyer $A_p(n)$;

Le coût est en $p \times n^2$ (une boucle en n pour le calcul du max).

4 Division du périmètre d'un polygone

On considère un polygone à n sommets, numérotés dans le sens des aiguilles d'une montre de 0 à $n-1$. La suite des longueurs des côtés est $\{a_0, a_2, \ldots, a_{n-1}\}$, comme représenté dans la figure ci-dessous.

1. On cherche tout d'abord à déterminer les deux indices i, j qui minimisent la valeur absolue de la différence entre les deux portions de périmètre qu'ils déterminent, i.e., qui minimisent (sommations modulo n) :

$$|(\sum_{l=i}^{j-1} a_l) - (\sum_{l=j}^{i-1} a_l)|$$

   (a) Donner un algorithme naïf et calculer sa complexité.

   (b) Proposer une solution en temps linéaire.

2. *Bonus :* Trouver en temps linéaire trois indices i, j, k qui minimisent la différence entre le plus grand "tiers" et le plus petit "tiers" qu'ils déterminent. Pouvez vous généraliser pour la découpe en k portions ?

Solution

Je vous renvoie ici au poly qui traite de l'exercice, c'est pas très utile que je retape le tout...

Pour la généralisation à la découpe en k portions, on n'a pas la solution mais on verra ce qu'ils nous sortent ;-).