IFT2015 automne 2011 — Devoir 4

Miklós Csűrös

21 novembre 2011

À remettre avant 20 :15 le 10 novembre par email (à csuros@iro. . .). Remettez un seul fichier PDF pour les exercices 4.1 et 4.2. Travaillez seul.

Vous pouvez avoir 10 points de boni pour une étude empirique de l'implantation de votre algorithme : remettez votre code source + fichiers `.cls` en un archive JAR, et un rapport en format PDF. N'oubliez pas de spécifier l'environnement de simulations (système d'exploitation, CPU, mémoire, Java VM) dans votre rapport. Vous avez jusqu'au 20 novembre pour soumettre une solution à 4.3.

4.1 Minimax (15 points)

On a un tableau $A[0..n-1]$ et un paramètre $0 < \ell \leq n$. On veut calculer

$$Q(A,\ell) = \max_{i=0,1,\ldots,n-\ell} \Big\{ \min\big\{ A[i], A[i+1], A[i+2], \ldots, A[i+\ell-1] \big\} \Big\}.$$

▶ Donnez un algorithme qui calcule $Q(A,\ell)$ en temps $O(n \log \ell)$, avec $O(\ell)$ espace de travail au plus.

Indice. L'algorithme parcourt A en «glissant» un fenêtre de taille ℓ au long du tableau : $m \leftarrow -\infty$; **for** $i \leftarrow 0, \ldots, n-\ell$: $m \leftarrow \max\{m, M(i,\ell)\}$ où $M(i,\ell) = \min\big\{A[i], A[i+1], A[i+2], \ldots, A[i+\ell-1]\big\}$. Une solution naïve calcule $M(i,\ell)$ dans une boucle interne qui prend $\Theta(\ell)$ temps. Une meilleure solution utilise une structure de données qui permet de calculer $M(i,\ell)$ rapidement. En particulier, la structure devrait permettre la mise à jour efficace $M(i,\ell) \to M(i+1,\ell)$, en temps $O(\log \ell)$.

4.2 Tri local (15 points)

On veut trier un tableau $A[0..n-1]$. Le tri correspond à une permutation π des indices $0, \ldots, n-1$ telle que $A\big[\pi(0)\big] \leq A\big[\pi(1)\big] \leq A\big[\pi(2)\big] \leq \cdots \leq A\big[\pi(n-1)\big]$.

Supposons qu'on sait que le tableau est «presque trié» dans le sens qu'il existe une constante Δ avec $\left|i - \pi(i)\right| \leq \Delta$ pour tout i. Dans d'autres mots, le tri déplace chaque élément par tout au plus Δ position. Il n'est pas difficile de voir que le tri par insertion prend $O(n\Delta)$ temps sur un tableau presque trié.

a. (5 points) ► Modifiez le tri par sélection pour qu'il prenne $O(n\Delta)$ temps (Δ est fourni comme argument).

b. (10 points) ► Développez un algorithme de tri qui prend le «désordre» Δ comme argument pour trier le tableau en $O(n \log \Delta)$ temps. L'algorithme ne doit utiliser que $O(\Delta)$ espace de travail. **Indice.** Exercice 4.1 devrait donner une bonne idée...

4.3 Étude empirique (10 points boni)

On utilise souvent une approche hybride pour trier un tableau : tri rapide pour de grands sous-tableaux et un autre tri (tri par insertion) pour de petits sous-tableaux. ► Implantez un tri hybride en utilisant votre solution de §4.2, et évaluez sa performance en pratique.

Plus spécifiquement, implantez le tri rapide en Java, avec la modification du cas terminal de récursion que les sous-tableaux de taille $\leq \Delta$ ne sont pas triés. Le résultat sera alors presque trié, et on a besoin d'un dernier parcours du tableau pour le trier complètement. En une implantation standarde, on utilise un tri par insertion dans ce dernier parcours. Mais vous pouvez également utiliser votre algorithme de §4.2.

Ici, vous devez comparer (1) le tri rapide classique ($\Delta = 1$), (2) une implantation hybride standarde (écrivez le code pour tri par insertion), et (3) votre algorithme de §4.2. Mesurez le temps des tris (`System.currentTimeMillis()`) en simulations : générez des tableaux initiels avec permutations aléatoires de $0, \ldots, n-1$, pour $n = 10000, 50000, 100000, 500000, 1000000$ (mesurez le temps 10 fois pour chaque n, et prenez les moyennes). Déterminez vous-même, à l'aide des simulations, comment Δ devrait être choisi en votre implantation et celle avec le tri par insertion.

L'algorithme suivant produit une permutation aléatoire.

```
    PERMUTATION(A[0..n − 1])          // performe une permutation aléatoire sur A
P1  for i ← 0, 1, ..., n − 2 do
P2      j ← i + RND(n − i)
P3      échanger A[i] ↔ A[j]          // noter que i = j est possible
```

En ligne P2 la fonction $\mathsf{RND}(k)$ donne un entier (pseudo-)aléatoire $0, 1, \ldots, k-1$, comme `java.util.Random.nextInt(int)`.