IN 101 - TD 05
15 octobre 2010

Énoncé

ENSTA ParisTech

Exercice 1 : Tri à bulles et variantes

Le tri à bulles consiste à parcourir un tableau de n entiers en comparant chaque élément avec son voisin. Si l'élément i est plus grand que l'élément $i+1$, on inverse ces deux éléments. En faisant un tel parcours pour i allant de 0 à $n-2$ on est certain de faire remonter le plus grand élément du tableau en dernière position. À chaque parcours, on fait remonter un peu les grands éléments vers le haut, un peu comme des bulles dans un verre (d'où le nom de l'algorithme).

Si un parcours fait remonter le plus grand élément à la fin, $n-1$ parcours sont suffisants pour trier entièrement le tableau en ordre croissant : on fait remonter le plus grand élément, puis le deuxième plus grand et ainsi de suite.

1.a] Programmez une fonction `void bubble(int n,int* tab)` qui implémente la version la plus simple du tri à bulles en faisant systématiquement $n-1$ parcours du tableau, en partant du début et en s'arrêtant à la $(n-i)$-ème case pour le i-ème parcours (après i parcours les i derniers éléments sont à leur place donc il n'est pas nécessaire de les comparer).

1.b] Vérifiez que votre algorithme précédent fonctionne en programmant une fonction `void print_tab(int n, int* tab)` qui affiche un tableau de taille n et un `main` qui lit une taille de tableau en argument avec `argc` et `argv` (ou utilise un tableau de taille fixée à l'avance si vous préférez) et l'initialise avec des valeurs aléatoires (en utilisant `srand(time())` et `rand()`). Vous pouvez aussi programmer une fonction `void tab_is_sorted(int n, int* tab)` qui affiche si le tableau passé en argument est bien trié.

1.c] Il se peut que $n-1$ parcours ne soient pas nécessaires pour trier entièrement le tableau. Dès qu'un parcours se déroule entièrement sans faire d'échanges on peut arrêter le tri. Écrivez une nouvelle fonction `void bubble_stop(int n,int* tab)` qui fait un tri à bulles (vous avez le droit de faire du copier-coller d'une partie de la fonction précédente !) et qui possède une variable `int stop` servant à arrêter le tri dès que possible. Vérifiez avec votre `main` que la fonction tri bien.

1.d] Le pire cas pour le tri à bulles est quand le plus petit élément du tableau est dans la dernière case : c'est le seul cas ou $n-1$ parcours sont nécessaires. Pour améliorer cela, le tri "cocktail" a été inventé : il consiste à parcourir le tableau alternativement en ordre croissant pour déplacer les plus grands éléments à la fin, et en ordre décroissant pour déplacer les plus petits éléments au début. Comme pour le tri à bulles standard, dès qu'un parcours (dans un sens ou dans l'autre) ne fait aucun échange, on peut interrompre le tri. Programmez une fonction `void cocktail(int n,int* tab)` implémentant le tri cocktail et vérifiez qu'elle fonctionne.

1.e] Dans le tri cocktail si un parcours croissant n'a échangé aucun élément après la position i, c'est que les éléments d'après sont déjà à leur place dans le tableau. On peut donc commencer le parcours décroissant suivant directement à la position i, et le parcours croissant suivant n'aura

pas non plus à aller au-delà de la position i. Ajoutez deux variables variable `int min` et `int max` à votre tri cocktail afin qu'il prenne en compte ce changement (faites cela dans une nouvelle fonction `void cocktail_minmax(int n,int* tab)` et vérifiez que le tri fonctionne encore).

1.f] (Pour ceux qui vont vite ou veulent perdre du temps) On veut comparer le nombre de comparaisons que font les 4 variantes du tri à bulles que l'on vient de programmer. Pour cela, ajoutez un compteur à chaque fonction qui compte les comparaisons et faites envoyer la valeur finale du compteur aux fonctions (il faudra changer leur signature pour qu'elles renvoient des `int`). Pour un même tableau faites afficher à votre programme le nombre de comparaison des 4 fonctions précédentes (⚠ il faut trier le *même* tableau, donc il va falloir en faire des copies avant le tri). Faites maintenant des statistiques pour voir combien de comparaisons sont nécessaires en moyenne pour chacun des algorithmes lors du tri d'un tableau de 100 éléments.

Exercice 2 : Tri fusion

Comme vu dans le cours, le tri fusion se base sur le principe "diviser pour régner". Il se compose donc de 3 opération : division, résolution et combinaison des résultats. Les opérations de division et résolution se font simplement avec deux appels récursifs, et l'opération délicate est celle de fusion. La fonction `merge_sort` est donc la suivante :

```
1 void merge_sort(int* tab, int p, int r) {
2   int q;
3   if ((r-q) > 1) {
4     q = (p+r)/2;
5     merge_sort(tab, p, q);
6     merge_sort(tab, q, r);
7     merge(tab, p, q, r);
8   }
9 }
```

2.a] Programmez la fonction `void merge(int* tab, int p, int q, int r)` qui recopie "dos à dos" les parties de `tab` de `p` à `q-1` et de `q` à `r-1` dans un nouveau tableau et et le parcourt en partant des deux bouts pour replacer les éléments dans l'ordre dans `tab`.

2.b] Ceux qui ont fait les statistiques de la question **1.f** peuvent maintenant regarder le nombre de comparaisons qu'effectue le tri fusion d'un tableau de taille 100.
⚠ Il faut ajouter un compteur à la fonction `merge`, mais aussi à la fonction `merge_sort`.

2.c] Le tri fusion fonctionne en coupant le tableau en 2 à chaque étape, mais on peut facilement le généraliser en coupant le tableau en k à chaque fois. Comment la fonction `merge` devrait-elle alors fonctionner ? Quelle serait la complexité asymptotique d'un tel tri (en fonction de n et k) ? Pour quelle valeur de k le nombre de comparaisons que l'on doit effectuer est-il le plus petit ?

2.d] (Pour ceux qui ont vraiment du temps à perdre) Programmez la fonction du tri fusion d'ordre k. La fonction `merge_sort_k` devra prendre un argument `int k` en plus et la fonction `merge_k` devra prendre `k` et un tableau de $k+1$ indices (qui remplace `p`, `q` et `r`). Elle devra utiliser un tableau de k indices (qui remplace `i` et `j`) pour savoir où l'on en est dans chacun des k sous-tableaux. Comptez pour différentes valeurs de k le nombre de comparaisons que fait le tri fusion d'ordre k pour trier un tableau de 100 éléments.