Quicksort

27 janvier 2020

L'objectif de ce TP est d'implanter l'algorithme `quicksort` en langage C.

Pour simplifier le sujet, on travaillera sur des données de type `double`, bien que l'on puisse facilement adapter le code à n'importe quel type pour lequel on dispose d'une fonction de comparaison.

Le fichier `qsort.c`, que vous pouvez télécharger sur l'ENT, contient des déclarations de fonctions et une fonction `main` permettant de faire des tests.

1 Échauffement

Implanter les fonctions

```
void echanger(double *e, double *f);
double rand_elt();
double *rand_tab(size_t n);
void aff_tab(const double *tab, size_t n);
double *copier_tab(const double *tab, size_t n);
void renverser_tab(const double *tab, size_t n);
```

en respectant les spécifications données dans le fichier `qsort.c`.

Tester chaque fonction en mettant à 1 (au lieu de 0) la constante symbolique associée : par exemple, pour tester la fonction échanger, il suffit, avant de compiler, de changer

```
#define TEST_ECHANGER 0
```

en

```
#define TEST_ECHANGER 1
```

2 Partitionnement

On rappelle le principe du partitionnement lors de `quicksort` :

1. L'une des valeurs du tableau est choisie comme *pivot*. Pour l'instant, nous choisirons toujours la dernière valeur du tableau.
2. Ensuite, il s'agit de mettre toutes les autres valeurs du tableau qui sont inférieures ou égales au pivot au début du tableau et celles qui sont supérieures ou égales vers la fin.
3. Pour finir, on met le pivot juste après les valeurs qui lui sont inférieures : il est à sa place définitive.

Implanter la fonction

```
double *partitionner(double *tab, double *fin);
```

qui prend un pointeur vers le début du tableau, et un pointeur vers la fin du tableau, puis partitionne le tableau suivant la procédure décrite ci-dessus. La valeur de retour est un pointeur vers la nouvelle position du pivot.

Tester cette fonction en modifiant la valeur de `TEST_PARTIT`.

3 Tri rapide

Une fois que l'on a partitionné le tableau, il suffit de trier le sous-tableau situé avant le pivot et le sous-tableau situé après.

Écrire la fonction (récursive)

```
void tri_rapide(double *tab, double *fin);
```

qui trie, en utilisant l'algorithme `quicksort`, le tableau d'adresse de début `tab` et d'adresse de fin `fin`.

Tester votre fonction en mettant à 1 la constante symbolique `TEST_QSORT1`

4 Tri par insertion

Nous voulons comparer empiriquement notre fonction à un tri par insertion. Pour se rafraîchir la mémoire, on pourra consulter

`https://fr.wikipedia.org/wiki/Tri_par_insertion`

1. Implanter le tri par insertion dans la fonction

   ```
   void tri_insertion(double *tab, double *fin);
   ```

   qui trie par insertion le tableau d'adresse de début `tab` et d'adresse de fin `fin`.
2. Tester votre fonction en mettant à 1 la constante symbolique `TEST_INSERT`.
3. Voir la différence de performance sur un tableau aléatoire de 100000 éléménts en mettant à 1 la constante symbolique `TEST_CHRONO`.
4. Ajouter dans la partie `TEST_CHRONO` un nouveau test pour voir le résultat sur un tableau déjà trié.

5. Ajouter dans la partie `TEST_CHRONO` un nouveau test pour voir le résultat sur un tableau trié à l'envers.

5 Sélection

On cherche à programmer la fonction

```
double quick_select(double *tab, double *fin, size_t k);
```

qui retourne le k-ième plus petit élément dans le tableau d'adresse de début `tab` et d'adresse de fin `fin`. Ceci doit bien sûr se faire sans tri, et de façon très efficace.

1. Trouver un algorithme du type « diviser pour régner » utilisant le partitonnement.
2. Écrire la fonction `quick_select` en utilisant la fonction `partitionner`.
3. Tester votre fonction en mettant à 1 la constante symbolique `TEST_SELECT`.

6 Modifications de `tri_rapide`

Les « améliorations » suivantes de `quicksort` sont mentionnées dans le livre de Robert Sedgewick « Algorithmes en langage C ». Suivant le temps qu'il vous reste, tentez de les implanter.

1. On peut choisir le pivot suivant la « médiane des trois » : le pivot est alors choisi comme étant la valeur médiane entre la première valeur du tableau, la dernière, et celle (ou l'une de celles) du milieu. Cette modification rend le tri bien meilleur sur des tableaux déjà triés ou triés en partie.
2. La fonction `tri_rapide` se termine par deux appels récursifs. On peut faire en sorte que le dernier de ces deux appels concerne le sous-tableau le plus grand. La récursivité terminale fait que la pile d'appel reste alors de taille très petite.
3. On peut dérécursiver la fonction `tri_rapide` en utilisant une pile explicite. Ceci permet de n'empiler que les positions des sous-tableaux les plus grands, en traitant directement les plus petits.
4. Enfin, on peut tirer partie du fait que le tri par insertion est très efficace sur les données « presque triées ». Ainsi, on peut ignorer les sous-tableaux de taille petite (disons inférieure à 15) et finir par appeler un tri par insertion pour finir le tri.

Pour finir, aucune implantation de `quicksort` n'est à l'abri d'un attaquant qui chercherait à deviner ses pires cas et à exploiter cette faille. Une méthode de tri utilisée en pratique pour se prémunir de cette attaque s'appelle `introsort`.

Il s'agit de compter le nombre d'itérations principales du tri (bref, le nombre d'appels à `partitionner`) et lorsqu'il dépasse un certain seuil ($K \log n$, avec K à choisir) de changer de méthode de tri et d'utiliser, par exemple, un tri fusion ou un tri par tas ou tout autre tri qui a une complexité au pire des cas $O(n \log n)$.

S'il vous reste du temps, vous pouvez implanter votre version de `introsort`.