TP d'Algo n° 1

EPITA ING1 2013; E. RENAULT

Consignes pour tous les TP de l'atelier d'algo : Tout doit être écrit en C. Le respect de la norme n'est pas indispensable ; la seule règle à respecter pour compter sur notre aide est d'*indenter* votre code. Vous pouvez utiliser toutes les fonctions de la bibliothèque standard du C, sauf évidement celles qui contiennent `sort`, `search`, ou `find` dans leur nom. Dans ces TP, vous pourriez avoir besoin entre autre des fonctions `fopen()`, `feof()`, `fclose()`, `fscanf()`, `getline()`, `printf()`, `malloc()`, `realloc()`, `free()`, `memcpy()`, `memmove()`, `log()`.

Objectif du TP1 : Mettre en place un environnement permettant la comparaison des différents algorithmes de tris vus en cours. On commence avec le tri par insertion et le tri par selection par ce qu'ils sont rapide à coder ; ce sont les fonctions à écrire autour qu'il est important d'avoir pour la suite.

1 Mise en jambe...

Nous voulons pouvoir travailler sur des tableaux d'entiers lus depuis un fichier texte. Ces fichiers possèdent un entier par ligne. Par exemple :

```
1349123
12
-42
3
```

Question 1 : Programmez la fonction `read_int_array()` qui lit un tel fichier, puis retourne un tableau alloué, rempli avec le contenu du fichier. Le paramètre `count` est lui aussi rempli par la fonction `read_int_array()` et indique le nombre d'éléments stockés dans le tableau.

```
int* read_int_array(const char* filename, size_t* count);
```

Remarque : La commande suivante engendre un fichier de 200 entiers aléatoires. Vous saurez l'adapter pour en produire d'avantage.

```
od -dAn -N400 /dev/urandom | sed 's/^ *//;s/ \+/\n/g' > tosort.data
```

Question 2 : Programmez la fonction `print_int_array()` qui permet d'afficher l'ensemble des éléments d'un tel tableau.

```
void print_int_array(int* tab, size_t count);
```

2 Un peu de tri...

Question 3 : Proposez une implémentation de la fonction de tri `insert_sort()`, effectuant un tri par insertion du tableau `tab`. (Dans l'ordre croissant.)

```
void insert_sort(int* tab, size_t count);
```

Question 4 : Implémentez de même le tri par selection :

```
void select_sort(int* tab, size_t count);
```

Question 5 : Pour tester tout cela écrivez-vous un programme `mysort` qui prend un fichier à trier en paramètre, et qui a une option pour indiquer l'algorithme de tri à utiliser.

3 Pour mettre un peu d'ordre...

Nous souhaitons pouvoir choisir l'ordre dans lequel les tableaux sont triés. Pour les entiers, le choix est restreint à croissant ou décroisant. Mais nous voulons aussi trier des chaînes de caractère.

Pour généraliser cela, nous allons écrire des fonctions de comparaison qui seront ensuite passées en argument aux différents tris. Une fonction de comparaison retourne 1, 0, ou -1, selon que le premier paramètre est *supérieur*, *égal*, ou *inférieur* au second dans l'ordre choisi. Le prototype d'une telle fonction est :

```
int compare(void* a, void* b);
```

L'argument est `void*` afin que toutes les fonctions de comparaison aient le même prototype, quelque soit le type sur lequel elles s'appliquent.

Question 6 : Écrivez une fonction de comparaison `greater_int()` respectant le prototype qui retourne 1 si $a > b$, 0 si $a = b$, et -1 si $a < b$.

```
int greater_int(void* a, void* b);
```

Question 7 : Écrivez une fonction de comparaison lesser_int respectant le prototype qui retourne 1 si $a < b$, 0 si $a = b$, et -1 si $a > b$.

```
int lesser_int(void* a, void* b);
```

Question 8 : Ecrivez les fonctions `generic_insert_sort()` et `generic_select_sort()`, qui prennent en paramètre, le tableau à trier (`tab`), son nombre d'éléments (`count`), la taille en octets d'un élément (`size`), et la fonction de comparaison entre éléments à utiliser.

```
void generic_insert_sort(void* tab,
                         size_t count,
                         size_t size,
                         int (*compare)(void* a, void* b));
void generic_select_sort(void* tab,
                         size_t count,
                         size_t size,
                         int (*compare)(void* a, void* b));
```

Ces versions des tris doivent fonctionner sur des tableaux de tout type : entiers, chaînes de caractère, flottants, etc. L'argument `tab` est donc déclaré comme `void*`. À l'intérieur

de la fonction, il faudra prendre en compte `size` pour calculer la position des éléments du tableau, utiliser `memcpy` (ou `memmove`) copier les éléments du tableau, et la fonction `compare` pour les comparer.

Question 9 : Modifiez `mysort` pour utiliser ces versions des tris. Ajoutez une option pour choisir l'ordre croissant ou décroissant.

4 Trier un dictionnaire...

Ajoutons les fonctions dont nous avons besoin pour tester ces tris avec des chaînes de caractères.

Question 10 : Écrivez une fonction `read_string_array()` qui alloue et rempli un tableau de chaîne de caractères à partir des lignes d'un fichier.

```
char** read_string_array(const char* filename, size_t* count);
```

Question 11 : Écrivez une fonction `print_string_array()` qui permette d'afficher un tel tableau.

```
void print_string_array(char** tab, size_t* count);
```

Question 12 : Écrivez deux fonctions de comparaison `greater_string()` et `lesser_string()` indiquant si une chaîne se trouve avant ou après l'autre dans l'ordre lexicographique (celui du dictionnaire).

```
int greater_string(void* a, void* b);
int lesser_string(void* a, void* b);
```

Question 13 : Modifiez `mysort` pour ajouter une option permettant de choisir si le tris doit être alphabétique ou numérique.

5 Un peu de complexité...

On cherche maintenant a instrumenter le code pour y compter le nombre d'opérations de comparaisons de chacun des algorithmes.

Question 14 : Instrumentez le code afin de compter le nombre de comparaisons effectuées lors du tri des tableaux d'entier et des tableaux de chaînes de caractère. (Une globale incrémentée dans les fonction de comparaison fera l'affaire.)

Vérifiez que vous avez les même résultats que nous sur les fichiers `tp1-num.data` et `tp1-str.data` disponible à côté de ce sujet.

```
generic_insert_sort, greater_int, tp1-num.data: 6 cmp
generic_insert_sort, lesser_int, tp1-num.data: 4 cmp
generic_select_sort, greater_int, tp1-num.data: 10 cmp
generic_select_sort, lesser_int, tp1-num.data: 10 cmp
generic_insert_sort, greater_string, tp1-str.data: 41 cmp
generic_insert_sort, lesser_string, tp1-str.data: 25 cmp
generic_select_sort, greater_string, tp1-str.data: 45 cmp
generic_select_sort, lesser_string, tp1-str.data: 41 cmp
```

6 Pour aller plus loin...

Question 15 : Écrivez un programme qui lit un fichier texte, et affiche le nombre d'occurrence de chaque ligne, comme le ferait `sort fichier | uniq -c`.