TP n°16 - Tri rapide - Quick sort.

On rappelle que, pour pouvoir exécuter les programmes compilés sur les machines du lycée, votre répertoire de travail doit être situé en dehors du répertoire `Documents`

L'algorithme de tri rapide

Exercice 1 (Codage du tri rapide).
Toutes les fonctions de cet exercice seront codées dans un fichier `quicksort.c`

1. Coder les deux fonctions classiques pour les tableaux : `void print_tab(int *tab, int n)` et `int *generate_random_array(int n, int a, int b)`. Cela doit aller très vite, nous l'avons fait de très nombreuses fois !
2. Écrire la fonction `void quicksort(int *tab, int n)` implémentant l'algorithme de tri rapide d'un tableau d'entiers.
3. Tester cette fonction dans votre fonction principale sur des tableaux aléatoires, d'abord de petite taille puis un peu plus gros.

L'algorithme de tri rapide est un tri optimal : nous le démontrerons bientôt. Enfin un tri qui n'est pas uniquement fait pour embêter les étudiants !

La bibliothèque C `sys/time.h` met à disposition des programmeurs une structure de données appelée `struct timeval` qui permet de récupérer un temps exprimé en secondes et microsecondes.

Le champ `tv_sec` de cette structure donne le nombre de secondes

Le champ `tv_usec` de cette structure donne le nombre de **micro**secondes en plus des secondes du champ `tv_sec`

La fonction `gettimeofday` permet de récupérer le temps écoulé depuis le 01/01/1970 à 0h0min0s (la création du monde pour les informaticiens, appelée Epoch [1]). Elle prend en entrée l'adresse d'un objet de type `struct timeval` et remplit les champs `tv_sec` et `tv_usec`. Le deuxième argument d'entrée prend une structure de données relative au fuseau horaire mais, par défaut, si on met ce deuxième argument à `NULL`, c'est celui de votre machine qui est choisi. Ainsi, l'appel suivant :

```
gettimeofday(&top, NULL);
```

remplit l'objet `top` qui est de type `struct timeval` avec le nombre de secondes et de microsecondes écoulées depuis l'Epoch.
Cette fonction peut être utilisée pour réaliser un chronomètre très précis permettant de relever le temps d'exécution d'une portion de code. Il suffit pour cela de récupérer le temps avant la fonction, puis après la fonction, et de faire la différence des deux.

1. https://fr.wikipedia.org/wiki/Epoch

Exercice 2 (Étude expérimentale de la complexité moyenne en temps CPU).

1. Génération du fichier d'analyse par le code C

a. Dans un nouveau fichier `time-analysis.c`, coder une fonction

```
double time_diff(struct timeval *beg, struct timeval *end
```

qui renvoie une différence de temps exprimée en secondes entre deux instants, ces deux instants étant donnés sous la forme de deux objets de type `struct timeval`. *Indication : la valeur renvoyée est non entière !*

b. Créer une interface `sort.h` pour les fonctions de votre fichier `quicksort.c` que vous souhaitez exposer pour les rendre visibles par d'autres codes.

c. Dans votre fichier `time-analysis.c`, écrire une fonction `compute_average` qui prend entrée une taille de tableau `size` et un nombre d'échantillons à créer `n_samples`, et qui va calculer la complexité moyenne en temps CPU du tri rapide sur `n_samples` tableaux de taille `size` générés aléatoirement.

d. Écrire un script `compile.sh` réalisant la compilation de vos fichiers source et créant l'exécutable. *Attention : il ne doit y avoir qu'un seul `main` pour l'ensemble du code. Commentez votre code plutôt que de l'effacer, pour garder une trace de votre travail*

e. Tester votre fonction `compute_average` sur plusieurs tailles de tableaux. On conseille un nombre d'échantillons plutôt modeste pour démarrer (de l'ordre de 5) pour éviter d'attendre trop longtemps. De même, dans cette phase de débogage, on commencera d'abord avec des tailles de tableaux raisonnables (pas plus de 1 million)

f. Écrire une fonction

```
write_analysis_file(char *path, int *sizes, int n_sizes, int n_samples)
```

qui prend en entrée un tableau listant des tailles de tableau, ainsi qu'une valeur d'échantillonnage, et qui crée un fichier de chemin `path` au format CSV, associant à chaque taille de tableau le temps CPU moyen calculé pour l'algorithme de tri rapide.

g. Tester sur plusieurs exemples, en allant regarder le contenu du fichier CSV crée.

2. Visualisation des résultats à l'aide d'un script Python

a. Écrire un script Python permettant de lire le fichier CSV crée et de tracer la courbe de complexité moyenne. On pourra utiliser le package Python `csv`.

b. Expliquer comment mettre en évidence la complexité quasi-linéaire (on dit aussi linéarithmétique) de l'algorithme de tri rapide.

c. Installer le package Python `python3-scipy` (*SCIentific PYthon*) en utilisant un gestionnaire de paquets en ligne de commande. Ce package contient de nombreuses fonctions de calcul scientifique, dont des fonctions d'analyses statistiques dans le sous-package `stats`.

d. Mettez en évidence la complexité quasi-linéaire du tri rapide en utilisant la fonction `linregress` du package `stats`.

e. **Approfondissement à la maison :** Pour le moment, le constat est qualitatif : la courbe ≪ ressemble ≫ à une droite. Pour valider de manière rigoureuse notre approche expérimentale, il faut faire une analyse d'incertitudes et tracer les intervalles de fluctuation afin de valider la pertinence de notre modèle d'un point de vue statistique. Faites-le !

Exercice 3 (Vérification expérimentale de la complexité moyenne théorique en nombre de comparaisons).
On évalue dorénavant la complexité moyenne du tri rapide en terme de **nombre de comparaisons effectuées entre 2 valeurs du tableau**. On montre par un calcul théorique que cette complexité moyenne évolue asymptotiquement de la manière suivante :

$$C_{\text{moy}}(N) \underset{N\to+\infty}{\sim} \alpha\, N \log_2(N)$$

où N est la taille du tableau à trier donné en entrée.

1. Modifier le code de tri rapide pour qu'il renvoie le nombre total de comparaisons entre deux éléments effectuées pour une exécution complète de l'algorithme. *Indication : on pourra essayer de mimer les stratégies avec accumulateurs utilisées abondamment en OCaml*
2. Dans le fichier `time-analysis.c`, adaptez la fonction `compute_average` pour qu'elle renvoie également, en plus du temps CPU moyen, le nombre moyen de comparaisons effectuées sur des tableaux générés aléatoirement, pour une taille de tableau et un nombre d'échantillons donné en entrée.
3. Adaptez la fonction `write_analysis_file` pour qu'un troisième champ représentant le nombre moyen de comparaisons effectuées soit ajouté dans le fichier CSV.
4. Enrichissez votre script Python pour tracer la courbe de complexité évaluée en terme de nombre de comparaisons.
5. A l'aide du module `stats`, estimez la valeur de la constante α telle que $C_{\text{moy}} = \alpha N \log_2(N)$. Jouez avec les paramètres de votre analyse (nombre d'échantillons, nombre de points de la courbe de complexité) pour vérifier la valeur obtenue.
6. Une analyse théorique montre que $\alpha \approx 1.39$. Redémontrer ce résultat ce résultat. Vous rédigerez proprement le raisonnement en expliquant les hypothèses de modélisation probabilistes et en détaillant les calculs.
7. Comment expliquer d'éventuelles différences avec votre résultat et la valeur théorique $\alpha \approx 1.39$?

Exercice 4 (Fonctions d'ordres supérieures en C - HORS-PROGRAMME).
Pour gagner en généricité, on peut également créer des **fonctions d'ordre supérieur** en C, comme on le fait en OCaml. On rappelle qu'une fonction d'ordre supérieur est une fonction dont au moins l'un des arguments d'entrée est aussi une fonction.
Pour cela, on utilise des pointeurs de fonctions. Allez lire la page suivante pour essayer de comprendre comment cela fonctionne :

https://fr.wikibooks.org/wiki/Programmation_C-C%2B%2B/Pointeurs_et_r%C3%A9f%C3%A9rences_de_fonctions

1. La librairie standard du C fournit plusieurs fonctions de tri, et notamment la fonction `qsort` qui implémente l'algorithme de tri rapide et qui permet de trier des tableaux contenant des valeurs de n'importe quel type pourvu qu'une relation d'ordre pour les objets de ce type soit fournie. A l'aide du manuel `man`, essayez de comprendre comment appeler la fonction `qsort` depuis votre code et faites un petit code `qsort_test.c` testant le bon fonctionnement de cette fonction.
2. Imaginons que nous souhaitions utiliser notre code d'analyse de complexité moyenne sur un autre tri, par exemple le tri insertion `insertion_sort`. La fonction `compute_average` ne changerait pas beaucoup car seul l'appel de la fonction de tri devrait être modifié : il faut remplacer `quicksort` par `insertion_sort`. De façon plus général, on peut imaginer qu'il serait intéressant de passer la fonction à étudier comme paramètre d'entrée de la fonction `compute_average`. Cela nous permettrait d'utiliser la même fonction d'analyse quelque soit l'algorithme de tri utilisé, pourvu que toutes les fonctions de tri aient le même prototype.

   Adaptez le code `time-analysis.c` pour prendre en compte cette remarque. Une fois la modification effectuée, copiez la fonction de tri par insertion précédemment implémentée et lancez son analyse. *Remarque : attention au prototype de la fonction de tri par insertion, il doit correspondre à celui de `quicksort`, modifiez-le en conséquence.*