PROG
TP#6

2021-11-15

Instructions de rendu

Ce TP fait l'objet d'un rendu *en binôme*. Vous devez envoyer votre travail sous la forme d'une archive à :
pierre.karpman@univ-grenoble-alpes.fr (pour le groupe 1) ;
hippolyte.signargout@univ-grenoble-alpes.fr (pour le groupe 2) ;
au plus tard le 2021-12-03 à 18 :00. Celle-ci doit contenir :

- Votre programme répondant aux questions ci-dessous. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
- Un Makefile et les instructions permettant de l'utiliser.
- Un compte-rendu détaillé en format PDF qui décrit vos choix de conception et vos réponses aux questions.

Première partie : quicksort

Le but de cet exercice est d'implémenter l'algorithme de tri *quicksort*, d'abord sur un tableau d'entiers puis pour des structures de donnée arbitraires.

On commence par rappeler l'algorithme de partition en ***pseudocode***, l'algorithme complet pouvant s'obtenir par une série d'appels récursifs à ce dernier.

```
/****************************************
input: T
input: d, f, x = T[f]
output: n, la nouvelle position de x
dans le tableau modifié en place t.q.
T[d..n-1] <= x; T[n+1..f] > x
****************************************/
x = T[f];
n = f;
for (i = f - 1; i >= d; i--)
{
        if (T[i] > x)
        {
                T[n] = T[i];
                T[i] = T[n-1];
                n = n - 1;
        }
}
T[n] = x;
```

Q.1 : Écrivez une fonction `int sorted(size_t Tlen, const uint32_t T[Tlen])` qui teste si un tableau `T` de `Tlen` `uint32_t` est trié.

Q.2 :

1. Écrivez une fonction `void qsortu(size_t Tlen, uint32_t T[Tlen])` qui trie en place un tableau `T` de `Tlen` `uint32_t` en utilisant quicksort.
2. Testez votre fonction. Ces tests devront au minimum utiliser `sorted`, sur des tableaux remplis aléatoirement.
3. Evaluez expérimentalement le coût de votre implémentation en mesurant le temps d'exécution sur des tableaux de taille variable (par ex. de quelques millions à quelques centaines de millions d'éléments)remplis aléatoirement. Faites attention à tirer vos éléments aléatoires (par ex. uniformément) dans un ensemble suffisamment grand par rapport à la taille du tableau (pourquoi ?), et à exclure le temps de génération des tableaux de vos mesures. Vous pourrez par exemple utiliser la fonction `gettimeofday` pour une mesure raisonnablement précise du temps d'exécution.

Q.3 :

1. Que pouvez-vous prévoir sur le temps d'exécution de votre tri sur un tableau dont les éléments sont initialement strictement décroissant ? Vérifiez le expérimentalement.
2. Modifiez votre fonction de partition afin de résoudre ce problème en moyenne. Une solution suffisante est d'échanger `T[f]` avec un élément au hasard de `T[d..f]`, puis de procéder comme auparavant.
3. Testez l'effet de votre changement sur le cas problématique précédent.

Q.4 :

1. Écrivez une nouvelle fonction `qsortg` qui modifie votre fonction `qsortu` pour qu'elle offre le même prototype et comportement que la fonction `qsort` de la bibliothèque standard. Vous pourrez par exemple utiliser `memcpy` pour effectuer les affectations nécessaires.
2. Comparez les performances de vos deux implémentations ainsi que celle de la bibliothèque standard pour des tableaux d'`uint32_t`.

Seconde partie : radixsort

On souhaite maintenant écrire une fonction `radixsortu` qui implémente le tri par base pour des entiers de 32 bits, en utilisant un tri par dénombrement comme sous-routine. L'idée du tri par base est d'écrire les entrées en base 2^b et d'utiliser un tri *stable* pour trier ses entiers suivant d'abord les b bits de poids faible, puis les b bits suivants, etc. Le ***pseudocode*** d'un tel tri est le suivant :

```
/****************************************
input: nlen, b
input: T, un tableau de nombres entiers non
signésdont les valeurs peuvent être
représentées avec nlen*b bits
output: Un nouveau tableau contenant les
éléments de T triés
****************************************/
T1 = T
T2 = T
for (int i = 0; i < nlen; i++)
{
        denomsort(T1, T2, b, i);
        swap(T1, T2);
}
return T1;
```

Ici, `denomsort` est un tri par dénombrement qui trie les entrées de `T` en considérant seulement leurs bits de position $\mathtt{i} \times \mathtt{b} \ldots (\mathtt{i}+1) \times \mathtt{b} - 1$.

Q.5 :

1. Écrivez une fonction `denomsort` qui implémente un tri par dénombrement pour des entiers stockés sur 32 bits. Faites particulièrement attention à son prototype, et réfléchissez bien (étant donné son usage) à ses arguments et à votre politique d'allocation/désallocation mémoire.
2. Testez votre fonction.
3. Testez les performances de votre implémentation pour trier un tableau de 60 000 000 nombres de 24 bits, et comparez avec votre implémentation de quicksort `qsortu`.

Q.6 :

1. Écrivez une fonction `radixsort` qui implémente le tri par base pour des entiers stockés sur 32 bits, et permettant d'aisément changer la base utilisée.
2. Testez votre fonction.

3. Testez les performances de votre implémentation pour trier un tableau de 240 000 000 nombres de 24 bits pour les bases 2^{24}, 2^{12}, 2^8, 2^6, 2^4, 2^3, 2^2.
4. Même question pour 240 000 000 nombres de 32 bits, pour des bases vous semblant appropriées.

Q.7 : Comparez expérimentalement la croissance des temps d'exécution de vos fonctions `qsortu` et `radixsortu` pour trier des tableaux de nombres de 24 bits de taille comprise entre 1000 et 500 000 000.