TP : Autour du tri rapide en C

Objectifs et conventions

On reprend l'algorithme du tri rapide vu en cours mais on l'adapte aux tableaux en C. On s'impose de travailler sans récursion et *en place*, c'est à dire sans utiliser de tableau auxiliaire. Cela nécessite d'utiliser deux indices entre lesquels il faut placer le pivot.

En application, on présente le « Quickselect » de Hoare qui permet de rechercher avec une bonne efficacité le k-ième plus petit élément d'un tableau sans passer par un tri préalable.

On se limite aux tableaux d'entiers à trier par ordre croissant.

Comme il est d'usage dans les sujet d'informatique, un même objet **x** est représenté avec deux polices de caractères : la notation droite **x** fait référence à un morceau de code dans lequel figure **x** tandis que la notation en italique x désigne **x** dans une preuve mathématique.

Pour désigner le sous-tableau d'un tableau **t** entre les positions **g** et **d**, nous écrivons **t[g :d]** (ou $t[g : d]$ dans une preuve). L'élément i du tableau est désigné par t_i dans une preuve et par **t[i]** dans un extrait de code.

1 Tri rapide

1.1 Partitionnement

Le principe de la *partition* est de trouver dans un tableau la *bonne place* d'un élément particulier appelé *pivot*. Par « bonne place », on entend la position définitive du pivot dans la version triée du tableau.

La fonction **int partition(int * t, int g, int d)** prend en paramètres un tableau d'entiers t et deux indices licites g, d du tableau avec $g \leq d$. Le pivot est au départ en position g (on améliorera ce point par la suite) et on cherche à le placer entre les indices g et d pour le mettre à sa position définitive (la position qu'il occupera quand le tableau sera complètement trié). cette position est la valeur retournée par la fonction.

Le pivot est l'élément t_g. Le tableau est parcouru durant une simple boucle entre $g + 1$ et d. On convient que l'indice i désigne la position courante explorée dans l'intervalle $[\![g + 1, d]\!]$. On gère également un indice j tel que, de g à $j - 1$, tous les éléments sont plus petits que le pivot.

Le pivot reste durant toute la boucle en position g. À la fin de la boucle, on échange l'élément position g avec sa position finale.

Q1 Écrire la fonction **partition** en utilisant les indices i, j décrits plus hauts et leurs spécifications. On impose que le parcours du tableau se fasse en une seule boucle et que la complexité spatiale soit en $O(1)$.

Q2 Les tours de boucles sont numérotés à partir de 1. Quel est le numéro du dernier tour ? Dans la suite, le tableau au tour k est noté t^k, j_k et i_k désignent les valeurs de **i** et **j**, $t^k_{[\![a,b]\!]}$ le sous-tableau entre les positions a, b (si $a > b$, le sous-tableau est vide).

(a) Établir un invariant en deux points faisant intervenir t^k, i_k, j_k.

(b) Montrer que l'invariant est vérifié avant l'entrée dans la boucle.

(c) On suppose avoir la preuve que la position définitive du pivot après le tri du tableau est entre g et d (ce qui signifie qu'on sait que les éléments avant la position g dans la version triée du tableau sont plus petits que le pivot et ceux à droite de la position d, plus grands).

Montrer que si notre invariant sur j, i est vérifié à la fin du dernier passage, alors la fonction est correcte :

- La valeur retournée est bien la place définitive du pivot (la place qu'il occuperait si tous le tableau avait été trié).
- Le pivot est à sa place définitive.

(d) On suppose qu'à la fin du tour k, on a $j_k = i_k$. Qu'est ce que cela signifie ?

(e) Montrer que si l'invariant est vérifié à la fin du tour $k < d - g$, alors il est réalisé à l'étape $k + 1$.

1.2 Tri rapide impératif

On veut maintenant implanter le tri rapide. Au lieu de s'appuyer sur une récursion comme en cours, on choisit la version impérative. Celle-ci est appelée *LampSort* et utilise une pile d'intervalles.

1.2.1 Piles

On demande d'écrire deux fichiers **stack.h** et **stack.c** qui implémentent la notion de pile.

Q3 Les intervalles sont représentés par la structure **intervalle** à deux champs entiers **g** (borne gauche et **d** (borne droite).

Définir la structure d'alias **intervalle** dans **stack.h**

Q4 Les piles sont constituées de *maillons* enchaînés entre eux et d'une *poignée* qui est un simple pointeur sur le premier maillon.

La structure d'alias **maillon** contient un champ **val** qui est un intervalle et un champ **prev** qui est un pointeur sur le maillon précédent dans la pile.

La structure d'alias **stack** est contient un unique champ qui est un pointeur sur le premier maillon de la pile. On pourrait d'ailleurs ajouter un champ **size** qui indiquerait la taille de la pile mais ce n'est pas demandé ici.

Écrire les structures **maillon** et **stack**.

Q5 On donne cet extrait de stack.h :

```
1 bool estVide(stack * p);// indique si la stack est vide
2 stack * init (); // crée une stack vide (renvoie un pointeur sur stack)
3 void affiche(stack * p);// affiche le contenu d'une stack
4 void push(stack * p, intervalle * v); // ajoute un nouvel élément
5  intervalle pop(stack * p);// déstackment
6  afficheIntervalle ( intervalle * pi);
```

Compléter le fichier **stack.h** ; écrire le fichier **stack.c** et écrire un fichier de test **tests_stack.c**.

1.2.2 Lampsort

On crée une pile vide dans laquelle on met l'intervalle de toutes les positions possibles.
A chaque tour, on dépile un intervalle :

- S'il est vide ou si c'est un singleton, on le retire simplement de la pile.
- Sinon, on le partitionne et on empile les deux sous-intervalles définis par cette partition.

L'algorithme s'arrête bien sûr quand la pile est vide.

Q6 Écrire la procédure **lampsort(int * p, int n)** qui prend en paramètres un tableau d'entiers et sa longueur n et trie le tableau selon le LampSort.

Q7 Un tri est dit *stable* si l'ordre des éléments de même valeur n'est pas modifié par le tri.
Le LampSort est-il stable ?

2 Un peu de hasard

Nous avons déjà étudié la complexité du tri rapide : dans le pire des cas, le tableau est trié et l'un des sous-tableaux issus du partitionnement est vide à chaque étape. Dans ce cas la complexité en nombre de comparaisons est en $O(n^2)$.

Pour éviter ce problème, à chaque appel de la fonction **partition t g d**, une position aléatoire k est tirée entre **g** et **d**. Puis un échange s'effectue entre les cases g et k ce qui a pour effet d'introduire du hasard dans le choix du pivot. Le reste de l'algorithme est identique à ce qui a déjà été vu.

On donne ici un petit tutoriel, très inspiré de Développez.com. N'y passons pas trop de temps : l'objectif est juste de modifier la fonction **partition**.

En C, les fonctions usuelles de production de nombres aléatoires s'appelent **rand** et **srand**, l'appel de la seconde devant précéder les appels de la première.

```
1  int rand(void);
2
```

Cette fonction retourne un nombre aléatoire à chaque appel compris entre 0 et **RAND_MAX**.

```
1  void srand(unsigned int seed);
2
```

Cette fonction initialise le générateur de nombres pseudoaléatoires avec une *graine* différente (1 par défaut). Elle ne doit être appelée qu'une seule fois avant tout appel à **rand**.

Dans un fichier **hasard.c** écrivons :

```
1  // tiré de Developpez.com
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5
6  int my_rand (void);
7
8  int main (void)
9  {
10   int i;
11
12   for (i = 0; i<10; i++)
```

```
13    {
14      printf  ("%d\n", my_rand());
15    }
16    return (EXIT_SUCCESS);
17  }
18
19  int my_rand (void)
20  {
21    return (rand ());
22  }
```

Q8 Compilez puis exécutez plusieurs fois. Que constate-t-on ?

Pour l'inconvénient soulevé à la question précédente, il faut appeler **srand** à chaque démarrage du programme avec une graine différente. Le plus simple est de prendre comme graine le nombres de secondes écoulées depuis le 1er janvier 1970 à 0h00 : il est donné par la fonction **time** qui est présente dans l'entête **time.h**.

Q9 Cette méthode n'est pourtant pas la panacée. Quel problème peut-on immédiatement identifier ? Peut-on facilement le résoudre ?

Nous introduisons dans la fonction **my_rand()** une variable entière statique **first** initialisée à zéro et modifiée lors du premier appel.

Q10 Dans quel segment de la mémoire virtuelle du programme se trouve la variable locale **first** ?

Lors du premier appel, la fonction initialise la graine en passant **time(NULL)** en paramètre de **srand**. On passe en paramètre de **time** le pointeur **NULL** car il n'est pas nécessaire de stocker le nombre de secondes effectivement produit.

Pour les appels suivants, **first** est différent de zéro et donc la graine n'est pas modifiée.

```
1  int my_rand (void)
2  {
3    static int  first  = 0;
4    if ( first  == 0)
5    {
6       srand (time (NULL));
7       first  = 1;
8    }
9    return (rand ());
10 }
```

Q11 Compilez et exécutez plusieurs fois.

Le problème de la question Q4 semble résolu.

Nous voulons maintenant générer un nombre aléatoire entre 0 et une borne supérieure $N - 1$ qui est bien souvent inférieure à **RAND_MAX**.

Q12 Une façon simple est de prendre le reste de la division par N du nombre pseudoaléatoire produit par **rand**. Mais cela ne fonctionne pas bien si N ne divise pas **RAND_MAX**. En prenant $N = 10$ et 25 pour **RAND_MAX** établir que certains nombres apparaissent plus souvent que d'autres et donc que la répartition des nombres produits n'est pas uniforme.

La solution a ce problème est appelée « mise à l'échelle » et consiste à prendre pour valeur **randomValue** retournée l'expression suivante :

```
1    int randomValue = (int)(rand() / (double)(RAND_MAX + 1L) * N);
```

Si $N = 5$ par exemple, cela revient à diviser $[0, 1[$ en 4 intervalles semi-ouverts à droite de tailles $\frac{1}{5}$ ($[\frac{0}{5}\frac{1}{5}[,[\frac{1}{5}\frac{2}{5}[,[\frac{2}{5}\frac{3}{5}[,[\frac{3}{5}\frac{4}{5}[$) et un dernier intervalle fermé $[\frac{4}{5}, 1 - \frac{1}{\text{RAND\_MAX} + 1}]$ soit $[\frac{4}{5}, 1 - \frac{1}{2147483648}]$ sur ma machine. Une fois ce résultat obtenu, on le multiplie par 4 et on obtient une valeur dans $\{0, 1, 2, 3\}$. Le dernier intervalle est certes un peu plus court que les autres mais on n'est quand même pas loin d'une distribution uniforme.

Q13 Écrire la fonction **int choix(int n)** qui prend en paramètre un nombre n et renvoie un entier pseudoaléatoire pris entre 0 et $n - 1$.

Q14 Écrire la fonction **int partition_rand(t,g,d)** qui réalise un échange aléatoire entre l'élément en position **g** et une postion aléatoire prise entre g et d avant de partitionner de la façon usuelle.

3 QuickSelect

Nous nous intéressons maintenant à la recherche du k-ième plus petit élément dans un tableau de longueur n. Par exemple 3 est le quatrième plus petit élément de **{10 ;2 ;8 ;0 ;1 ;3 ;0 ;18}** (si on commence à compter les positions à partir de zéro). Une façon simple de résoudre ce problème est de trier la liste (en $O(n \log n)$) et de renvoyer l'élément d'indice k du tableau trié.

Mais nous voulons éviter de trier le tableau : c'est ce que permet le *QuickSelect* de Hoare.

Comme pour le tri rapide, on partitionne encore le tableau à chaque étape. La position obtenue pour le pivot permet de choisir entre s'arrêter, poursuivre en explorant la partie à droite du pivot ou explorer la partie à sa gauche. La différence avec le tri rapide est qu'on explore un seul des sous-tableaux séparés par le pivot.

Q15 Écrire la fonction **int quickselect(int * t, int k, int n)** qui cherche le k-ième plus petit élément du tableau t de longueur n. On suppose que $k < n$.

Cette fonction s'appuie sur une fonction récursive qui cherche le k-ième plus petit élément entre deux indices $g < d$.

Q16 Etablir la complexité au mieux au pire et en moyenne du **quickselect** pour le nombre de compraisons en fonction de la longueur n du tableau d'entrée.

Pour un tableau trié de longueur N, la *médiane* est l'élément situé au milieu du tableau si celui-ci existe :

- Si N est impair, la médiane est exactement l'élément milieu.
- Si N est pair et le tableau est non vide, le milieu du tableau n'est pas un indice car il n'est pas entier. L'usage est de retourner la moyenne des deux éléments dont les positions sont les plus proches du milieu.

Q17 Écrire la fonction **double mediane(int * t, int n)** qui renvoie la médiane du tableau t de longueur n selon la définition ci-dessus.