TD XXXII

AUTOUR DU TRI RAPIDE

1 Tri rapide en C

Pour passer un sous-tableau à une fonction en C, on peut passer un pointeur vers la première case de la partie qui nous intéresse et la longueur de la partie. Pour obtenir un pointeur vers la partie qui commence à l'indice `i`, il suffit de prendre `&t[i]` (ce n'est pas idiomatique du tout, mais c'est la manière de procéder en restant dans les bornes du programme).

Exercice XXXII.1 p. 3

1. Écrire une fonction `partition` ayant le prototype suivant :

```c
int partition(int *arr, int len);
```

Cette fonction partitionnera le tableau passé en argument en utilisant le premier élément comme pivot.

2. Écrire une fonction `quicksort` ayant le prototype suivant :

```c
void quicksort(int *arr, int len);
```

Remarque
Contrairement à ce que nous avons fait en OCaml, il n'y aura pas de fonction auxiliaire ici.

Exercice XXXII.2 p. 3

1. À l'aide des fonctions présentes dans le squelette, écrire une fonction `test_quicksort` vérifiant la correction de `quicksort` sur divers tableaux aléatoires de petite taille (entre 1 et 100 éléments, disons).
2. Observer le comportement de `quicksort` sur des tableaux plus grands (dizaines ou centaines de milliers d'éléments), dans le cas d'un tableau aléatoire d'entiers entre 0 et 1 000 000 et dans le cas d'un tableau trié,

2 Quickselect

Le problème de la *sélection* est le suivant :

- on nous donne un tableau t de n éléments (d'un type totalement ordonné, des entiers par exemple), et un entier k vérifiant $0 \leqslant k < n$;
- on doit renvoyer l'élément de t qui serait à l'indice k si t était trié.

Exercice XXXII.3 p. 4

1. Que doit renvoyer $\texttt{select}(t, 0)$? $\texttt{select}(t, n-1)$?
2. Comment peut-on calculer une médiane d'un tableau si l'on dispose d'une fonction `select` ?
Remarque
Une médiane d'un tableau de taille n est un élément x de ce tableau tel que au moins $n/2$ éléments de t soient inférieurs ou égaux à x et au moins $n/2$ soient supérieurs ou égaux.

Exercice XXXII.4

p. 5

1. Proposer une méthode permettant de réaliser la sélection en temps $O(n \log n)$.

Pour faire mieux, on peut adapter l'algorithme du tri rapide : c'est l'algorithme *quickselect*. L'idée est de réutiliser la fonction `partition` écrite précédemment (sans la modifier), mais de ne pas trier entièrement le tableau : un seul appel récursif est nécessaire à chaque étape.

2. Écrire une fonction `quickselect_aux` de prototype

```
int quickselect_aux(int *arr, int k, int len);
```

Cette fonction renverra le $k-1$-ème plus petit élément du tableau (c'est-à-dire $\texttt{select}(\texttt{arr}, k)$) et pourra avoir un effet secondaire quelconque sur le contenu de ce tableau.

3. Écrire une fonction `quickselect` ayant le même prototype que `quickselect_aux` et la même valeur de retour, mais ne modifiant pas le tableau qu'on lui passe en paramètre.
4. Analyser la complexité temporelle de `quickselect` :
   - dans le pire cas ;
   - dans le cas où le pivot choisi partage équitablement le tableau à chaque étape.

3 Introsort

Exercice XXXII.5

p. 6

Écrire une fonction `heapsort` ayant le prototype suivant :

```
void heapsort(int *arr, int len);
```

Cette fonction triera le tableau `arr` par ordre croissant en utilisant l'algorithme du tri par tas. On réfléchira aux fonctions auxiliaires utiles (on n'a pas besoin de toutes les fonctions sur les tas).

Exercice XXXII.6

1. Écrire une fonction `ilog` calculant la partie entière du logarithme en base 2 de son argument (que l'on pourra supposer strictement positif).

```
int ilog(int n);
```

2. Dans le cas où le pivot partage équitablement le tableau à chaque étape, quelle est la profondeur de récursion maximale de `quicksort` ?
3. Écrire une fonction `introsort` de prototype

```
void introsort(int *arr, int len);
```

Cette fonction triera le tableau passé en argument, en place, en commençant par un tri rapide mais en basculant vers un tri fusion dès que la profondeur de récursion est deux fois supérieure à ce que l'on pourrait attendre dans un « bon » cas.

Remarque

Une fonction auxiliaire sera nécessaire.

Solutions

Correction de l'exercice XXXII.1 page I

1. On procède comme dans le cours (schéma de Lomuto), sauf qu'il est inutile ici de déplacer le pivot (puisqu'on utilise le premier élément). La toute fin diffère également un peu : le pivot doit être échangé avec le dernier éléments de la zone « petits » et pas avec le premier de la zone « grands ».

```
int partition(int *t, int len){
    int vpiv = t[0];
    int i = 1;
    for (int j = 1; j < len; j++){
        if (t[j] <= vpiv){
            swap(t, i, j);
            i++;
        }
    }
    swap(t, 0, i - 1);
    return i - 1;
}
```

2. C'est une traduction directe du cours, en utilisant la remarque sur la possibilité de passer un pointeur vers autre chose que le début du tableau :

```
void quicksort(int *t, int n){
    if (n <= 1) return;
    int k = partition(t, n);
    quicksort(partition, &t[k + 1], n - k - 1);
    quicksort(partition, t, k);
}
```

Correction de l'exercice XXXII.2 page I

1. On peut par exemple écrire cela :

```
void test_quicksort(int maxlen, int iterations, int bound){
    for (int len = 1; len <= maxlen; len++){
        for (int i = 0; i < iterations; i++) {
            int *arr = random_array(len, bound);
            int *arr_copy = copy(arr, len);
            insertion_sort(arr, len);
            quicksort(arr_copy, len);
            assert(is_equal(arr, arr_copy, len));
            free(arr);
            free(arr_copy);
        }
    }
}
```

Choisir une valeur de `bound` pas trop grande permet de s'assurer que l'on sera bien confronté à des tableaux avec valeurs répétées.

2. On écrit un petit programme qui prend une taille en ligne de commande et mesure le temps pour trier un tableau d'entiers aléatoires et un tableau déjà trié de cette taille :

```c
int main(int argc, char* argv[]){
    assert(argc == 2);
    int len = atoi(argv[1]);
    int bound = 1000 * 1000;
    int *random = random_array(len, bound);
    int *sorted = malloc(len * sizeof(int));
    for (int i = 0; i < len; i++){
        sorted[i] = i;
    }
    clock_t t0 = clock();
    quicksort(random, len);
    clock_t t1 = clock();
    quicksort(sorted, len);
    clock_t t2 = clock();
    double time_random = 1.0 * (t1 - t0) / CLOCKS_PER_SEC;
    double time_sorted = 1.0 * (t2 - t1) / CLOCKS_PER_SEC;
    printf("random array of size %d : %.3f s\n", len, time_random);
    printf("sorted array of size %d : %.3f s\n", len, time_sorted);
    free(random);
    free(sorted);
    return 0;
}
```

On obtient alors (en compilant en `-O2` et sans `fsanitize` pour obtenir des résultats un minimum réalistes) :

```
$ gcc -o qsort -Wall -Wextra -O2 qsort.c
$ ./qsort 10000
random array of size 10000 : 0.003 s
sorted array of size 10000 : 0.047 s
$ ./qsort 20000
random array of size 20000 : 0.001 s
sorted array of size 20000 : 0.183 s
$ ./qsort 30000
random array of size 30000 : 0.002 s
sorted array of size 30000 : 0.408 s
$ ./qsort 40000
random array of size 40000 : 0.003 s
sorted array of size 40000 : 0.712 s
# ./qsort 50000
random array of size 50000 : 0.004 s
sorted array of size 50000 : 1.127 s
$ ./qsort 100000
random array of size 100000 : 0.008 s
sorted array of size 100000 : 4.354 s
```

Correction de l'exercice XXXII.3 page I

1. $\texttt{select}(t, 0)$ doit renvoyer le minimum du tableau, $\texttt{select}(t, n-1)$ le maximum.
2. $\texttt{select}(t, \lfloor n/2 \rfloor)$ renvoie une médiane.

Correction de l'exercice XXXII.4 page 2

1. Pour faire la sélection en temps $O(n \log n)$, il suffit de trier le tableau à l'aide d'un algorithme ayant cette complexité dans le pire cas (tri fusion ou tri par tas, par exemple), puis de renvoyer la case k du tableau trié.
2. Pour calculer $\texttt{select}(t, k)$, on partitionne t et l'on regarde où se retrouve le pivot :
   - s'il est en position k, alors c'est la valeur recherché ;
   - s'il est en positon $i_{piv} < k$, alors l'élément recherché est à droite du pivot, et l'on fait donc un appel récursif $\texttt{select}(t[i_{piv} + 1 : n], k - i_{piv} - 1)$ (il faut changer la valeur de k puisqu'il y a k éléments dans la zone de gauche, plus le pivot) ;
   - s'il est en position $i_{piv} > k$, alors l'élément cherché est dans la partie de gauche, et l'on effectue simplement l'appel $\texttt{select}(t[0 : i_{piv}], k)$.

   Cette fonction a pour effet secondaire de « trier partiellement » le tableau.

```c
int quickselect_aux(int *t, int k, int len){
    assert (i < len);
    int ipiv = partition(t, len);
    if (ipiv == k) {
        return t[ipiv];
    }
    if (ipiv < k) {
        return quickselect_aux(&t[ipiv + 1], k - ipiv - 1, len - ipiv - 1);
    }
    return quickselect_aux(t, k, ipiv);
}
```

3. On effectue une copie du tableau puis l'on appelle `quickselect_aux` sur cette copie. On n'oublie pas de libérer la copie avant de renvoyer notre résultat.

```c
int quickselect(int *t, int k, int len){
    int *t_copy = malloc(len * sizeof(int));
    memcpy(t_copy, t, len * sizeof(int));
    int res = quickselect_aux(t_copy, k, len);
    free(t_copy);
    return res;
}
```

4. À chaque étape, on effectue une partition (temps $O(n)$), puis :
   - soit le pivot tombe à la bonne place et l'on a terminé ;
   - soit il faut faire un appel récursif sur l'une des parties, de taille au plus $n - 1$.

   Si le tableau initial est trié par ordre croissant et que l'on calcule $\texttt{select}(t, n-1)$ (en prenant le premier élément de la zone comme pivot à chaque fois), on partagera systématiquement en une zone vide et une de taille $n - 1$ et il faudra continuer les appels jusqu'à arriver à une zone de taille 1. La complexité est alors en $O(\sum_{k=1}^{n} k) = O(n^2)$.
   Si le pivot tombe à chaque fois au milieu, on a $T(n) \leqslant T(n/2) + An$, avec A une constante.

$$T(2^i) \leqslant T(2^{i-1}) + A2^i$$
$$T(2^i) - T(2^{i-1}) \leqslant A2^i$$
$$T(2^n) - T(1) \leqslant A \sum_{i=2}^{n} 2^i \leqslant B2^n$$

   On en déduit $T(2^n) = O(2^n)$. Par croissance de T, on obtient $T(n) \leqslant T(2^{\lceil \log_2 n \rceil}) \leqslant B2^{\lceil \log_2 n \rceil} \leqslant B2^{\log_2 n + 1} = O(n)$. La complexité est donc linéaire si le pivot tombe au milieu du tableau à chaque fois. On peut prouver que la complexité en moyenne (sur un tableau mélangé de manière uniforme) est elle aussi linéaire.

Correction de l'exercice XXXII.5 page 2

On utilise un tas max (pour obtenir un tableau trié en ordre croissant à la fin). On n'utilise pas la fonction `siftdown` (donc on ne l'écrit pas), et la fonction d'extraction du maximum n'a pas à renvoyer de résultat (on l'utilise uniquement pour son effet de bord).

```c
void siftdown(int *heap, int i, int len){
    int imin = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < len && heap[left] > heap[i]) imin = left;
    if (right < len && heap[right] > heap[imin]) imin = right;
    if (imin != i){
        swap(heap, i, imin);
        siftdown(heap, imin, len);
    }
}

void extract_max(int *heap, int len){
    swap(heap, 0, len - 1);
    siftdown(heap, 0, len - 1);
}

void heapify(int *heap, int len){
    for (int i = (len - 1) / 2; i >= 0; i--){
        siftdown(heap, i, len);
    }
}

void heapsort(int *heap, int len){
    heapify(heap, len);
    for (int i = len; i > 0; i--){
        extract_max(heap, i);
    }
}
```

Correction de l'exercice XXXII.6 page 2

1. On va au plus simple :

```c
int ilog(int n){
    int i = 0;
    while (n > 1){
        i++;
        n = n >> 1;
    }
    return i;
}
```

2. Si le pivot tombe au milieu à chaque fois, la taille du tableau est divisée par 2 à chaque étape (à une unité près) et il faut donc $\log_2 n$ étapes (à une unité près) pour arriver à un tableau de taille inférieure ou égale à 1. La profondeur de récursion est donc de $\log_2 n$ dans ce cas.
3. On se fixe une profondeur maximale de $2\log_2 n$, et si l'on dépasse cette profondeur on finit par un tri par tas. On peut donc avoir certaines zones du tableau qui finiront par être traitées par `heapsort` alors que d'autres seront traitées par `quicksort` jusqu'au bout.

```c
void introsort_aux(int *t, int len, int max_depth){
    if (len <= 1) return;
    if (max_depth < 0) {
        heapsort(t, len);
    } else {
        int k = partition(t, len);
        introsort_aux(&t[k + 1], len - k - 1, max_depth - 1);
        introsort_aux(t, k, max_depth - 1);
    }
}


void introsort(int *t, int len){
    int max_depth = 2 * ilog(len);
    introsort_aux(t, len, max_depth);
}
```