TP4 : tableaux, pointeurs

Algorithmique et projet de programmation — M1 mathématiques

27 septembre 2016

Exercice 1 (Warm up)**.** Créer les fonctions suivantes traitant des tableaux d'entiers :

```c
/* print the elements of an array of given size ; separate
   elements by spaces and print a newline at the end */
void pp_array (int size, int *array);

/* swap elements of index i and index j in an array */
void swap (int i, int j, int *array);

/* copy the array src into the array dest ; both should have the
   same size ; notice the 'const' keyword */
void copy_array (int size, int *dest, const int *src);

/* reverse the order of src to put into dest ; both should have
   the same size */
void reverse_array (int size, int *dest, const int *src);

/* return the index of the maximum or minimum of an array of
   given size */
int max (int size, int *array);
int min (int size, int *array);
```

1 Tris

Commençons par les tris naïfs, qui s'éxécute en $O(n^2)$.

Exercice 2 (Par sélection)**.** Le tri par sélection opère de la façon suivante : il parcourt le tableau à la recherche du plus grand élément, puis l'échange avec le dernier élement du tableau ; il cherche ensuite le plus grand élément parmi les valeurs restantes et l'échange avec l'avant dernier élément du tableau ; etc.

Écrire une fonction qui implémente le tri par sélection :

```c
void select_sort (int size, int *array);
```

Indice : en C, un tableau de taille n est tout autant un tableau de taille $i < n$.

Exercice 3 (À bulles)**.** Le tri à bulles repose sur le même principe que le tri par sélection : placer l'élément maximal en dernière position et recommencer sur le sous-tableau des éléments restants. En revanche, on ne cherche pas le maximum explicitement : on va le « faire remonter » (d'où le nom de *tri à bulles*) en inversant

localement les valeurs des cases i et $i+1$ quand la première est supérieure à la seconde.

Écrire une fonction qui implémente le tri à bulles :

```
void bubble_sort (int size, int *array);
```

Exercice 4. Le tri par insertion change le point de vue : considérant que la fin d'un tableau, des indices i à $n-1$, est déjà trié, il suffit d'insérer la valeur de la case i au bon endroit pour avoir une fin triée d'une case de plus. En itérant le processus depuis une fin vide jusqu'à une fin de la taille du tableau, on aura trié le tableau en entier.

Écrire une fonction qui implémente le tri par insertion :

```
void insert_sort (int size, int i, int *array);
```

qui suppose que le tableau `array` de taille `size` est trié entre les index `i+1` et `n-1`, et qui trie entièrement le tableau `array`.

On passe maintenant au tri dont la complexité est $O(n \log n)$.

Exercice 5 (Rapide)**.** Le tri rapide trie les valeurs entre les cases d'index `min` et `max` d'un tableau de la façon suivante : on choisit une valeur p (appelée pivot) du tableau à trier (souvent on prend juste la dernière valeur du tableau), puis on met dans n'importe quel ordre tous les éléments inférieurs au pivot à sa gauche et tous les élements supérieurs à sa droite ; on réitère sur chacun des deux sous-tableaux gauche et droit ainsi créés.

Écrire une fonction qui implémente le tri rapide :

```
void quick_sort (int min, int max, int *array);
```

Exercice 6 (Fusion)**.** Le tri fusion se base sur le principe « diviser pour régner » : trier un tableau revient à trier la première moitié et la deuxième moitié indépendemment, puis à les fusionner. La fusion suppose voir deux tableaux triés de même taille,qu'elle parcoure simultanément, insérant dans un nouveau tableau le plus petit des deux éléments lus à chaque fois.

Écrire une fonction :

```
void merge (int start, int mid, int end, int *array);
```

qui suppose que les sous-tableaux {`array[start],...,array[mid-1]`} et {`array[mid],...,array[end]`} sont triés, et écrit la fusion des deux dans le sous-tableaux {`array[start],...,array[end]`}. On n'hésitera pas à utiliser un tableau intermédiaire temporaire (la modification *en place* étant difficile à réaliser).

Écrire une fonction récursive (en s'aidant bien entendu de la fonction précédente) qui implémente le tri fusion pour le sous-tableau compris entre `start` et `end` d'un tableau `array` :

```
void merge_sort (int start, int end, int* array);
```

2 Autres exemples

Exercice 7 (Crible d'Ératosthène)**.** Au III^{e} scièle avant J-C., Ératosthène invente une manière astucieuse de générer tous les nombres premiers jusqu'à une certaine

borne n. Il suffit pour cela d'écrire les entiers de 2 à n sur une feuille puis de répéter l'action suivante en partant de 2 : si le nombre n'est pas barré, on barre tous ces multiples sauf lui-même ; si le nombre est barré, on passe au suivant. Lorsqu'on arrive à n, les nombres non barrés sont exactement ceux qui ne sont multiples de personne à part eux même, c'est-à-dire les nombre premiers.

Implémenter le crible d'Ératosthène en une fonction :

```
void sieve(int n, char *era);
```

où le tableau `era` contiendra `0` dans les cases indexées par des non-premiers et `1` dans celles indexées par des premiers.

Exercice 8 (Euclide étendu, le retour)**.** Maintenant que l'on connaît les pointeurs, on peut implémenter une version plus simple de l'algorithme d'Euclide étendu. Rappelons que cet algorithme renvoie, pour tout couple $(a\,,b) \in \mathbb{N}$, le pgcd de a et b, ainsi que deux entiers $u\,, v \in Z$ tels que $ua + bv = \gcd(a\,,b)$. Le problème est qu'un programme `C` n'est autorisé à renvoyer qu'une seule valeur ! Heureusement, grâce aux pointeurs, on peut contourner ce problème.

Implémenter l'algorithme d'Euclide comme une fonction récursive de signature :

```
void ext_euclid (int a, int b, int *gcd, int *u, int *v);
```

3 À rendre pour la prochaine fois

Exercice 9.

(i) Écrire une fonction, la plus naïve possible, renvoyant la somme maximale que l'on peut obtenir en additionnant des éléments consécutifs d'un tableau `array` de taille `size` :

```
int max_sum (int size, int *array)
```

Par exemple, la fonction doit renvoyer `11` sur le tableau `{-1,1,8,-7,9,-2}` (réalisé par les éléments consécutifs $1\,, 8\,, -7\,, 9$).

(ii) Estimer la complexité de la fonction précédente (en fonction de la taille du tableau).

(iii) Le but de cette question est d'obtenir un algorithme plus efficace. Notons m_i la somme maximale d'éléments consécutifs pour le sous-tableau de `array` donné par les indexs de 0 à i. Par exemple, m_0 est 0 (la somme vide) si `array[0]` est négatif et vaut `array[0]` sinon.

En exprimant m_{i+1} en fonction de m_i, écrire une fonction calculant la même chose qu'en question (a) mais plus efficacement :

```
int max_sum_better (int size, int *array);
```

(iv) Quelle est la complexité de cette nouvelle fonction ?

Cet exercice est une initiation à une technique que vous allez beaucoup utiliser par la suite : la programmation dynamique.