Références et programmation impérative

Dans le TP précédent, nous avons utilisé des fonctions récursives et des listes. Nous allons maintenant aborder les bases de la programmation impérative, en utilisant les vecteurs, les références ainsi que de nouvelles structures de données (en utilisant les types somme et les types enregistrement).

Notions : références, boucles **for** et **while**, tri par insertion, tri fusion, tri rapide et sélection en temps linéaire

1 Introduction aux références

Un processeur classique (Pentium, Athlon, ...) n'a que quelques variables internes (on les appelle des « registres »). Il y a deux types de registres : les registres qui peuvent contenir des nombres entiers (compris entre -2^{31} et $2^{31}-1$) et les registres qui peuvent contenir des nombres flottants. Il n'y a donc pas de registres pour stocker des chaînes de caractères, des listes ou des vecteurs.

Comment peut-on faire pour manipuler ces objets ?

Toutes les données sont stockées dans la mémoire de l'ordinateur (« RAM »), qui est exactement comme un grand (typiquement de taille un milliard) vecteur. Chaque caractère d'une chaîne va occuper une case de ce vecteur. Par exemple, si dans votre programme vous utilisez la chaîne machaine="azerty", il existe n tel que RAM[n]='a', RAM[n+1]='z', RAM[n+2]='e', RAM[n+3]='r', RAM[n+4]='t', RAM[n+5]='y'. Ce n (qui est un entier tout comme ceux que vous manipulez en Caml) porte un nom en informatique, c'est l'adresse de machaine.

Ainsi, quand vous appelez string_length machaine, l'ordinateur lui donne en fait l'adresse de machaine. Par contre, quand vous faites une soustraction d'entiers, l'ordinateur donne directement la valeur des deux entiers à la fonction de soustraction.

Si vous voulez changer la valeur de machaine (par exemple, pour avoir machaine = "qwerty" – désolé pour les exemples peu inspirés), Caml va écrire "qwerty" à une nouvelle position dans la RAM (à une nouvelle adresse, donc), et donner cette adresse à machaine.

Par choix, Caml interdit de changer la valeur des variables (qui ne sont plus très ... variables). Pour avoir des variables vraiment variables, il faut utiliser des références.

Qu'est-ce donc qu'une référence ?

Si vous faites une référence vers une variable, l'ordinateur écrit la valeur de cette variable en mémoire et ne manipulera plus que son adresse. Cela permet de changer la valeur de cette variable en écrivant juste dans la RAM.

```
let a = 4;;
let b = ref 5;;
b := 6;;
```

Dans le premier cas, un registre du processeur contiendra réellement 4, alors que dans le second il ne contiendra que l'adresse d'une case de la RAM contenant 5. La troisième ligne va écrire 6 dans la mémoire (à la place de 5), mais *sans* changer l'adresse de b. Ainsi, nous avons des variables qui peuvent avoir des valeurs différentes sans réellement changer.

Autre conséquence : si deux variables a et b sont des références vers la même case mémoire, toute modification de a entraînera une modification de b.

```
let a = ref 4;;
let b = a;;
b := 5;;
a;;
!a;;
!b;;

a : int ref = ref 4
b : int ref = ref 4
- : unit = ()
- : int ref = ref 5
- : int = 5
- : int = 5
```

Comme on peut le voir dans cet exemple, on crée une référence à un objet avec ref, on modifie la valeur pointée par cette référence avec := et un récupère la valeur pointée par cette référence avec !.

▶ **Question 1** *Écrivez en Caml une fonction qui crée*

un vecteur de vecteurs de taille $n \times m$. Quel est le problème si on écrit une fonction trop naïve ? Quel est le lien avec ce qui précède ?

```
make_matrix : int -> int -> 'a -> 'a vect vect
```

2 Le tri à bulle

Il s'agit sûrement du tri le plus simple : tant qu'on trouve deux éléments qui ne sont pas rangés dans le bon ordre, on les permute et on recommence à parcourir le tableau.

▶ **Question 2** *Quelle est la complexité en temps (dans le pire cas) de cet algorithme ?*

▶ **Question 3** *Ecrivez une fonction qui échange les éléments d'indice* i *et* j *dans le tableau* v.

```
echange : 'a vect -> int -> int -> unit
```

▶ **Question 4** *Ecrivez une fonction* tri_bulle *ayant le prototype suivant :*

```
tri_bulle : 'a vect -> 'a vect
```

3 Un tri déjà vu : le tri par insertion

Le tri par insertion a déjà été vu lors du TP précédent, mais voici un rappel de son principe : on considère un vecteur non trié de n éléments $0, \ldots, n-1$.
On parcourt le vecteur dans l'ordre $0, \ldots, n-1$. À chaque i-ème élément, on recherche sa place correcte parmi les $i-1$ éléments précédents, et on l'y insère. Ensuite, on passe à l'élément précédent. Les nombres donnés en entrée sont *triés sur place* : ils sont réorganisés à l'intérieur du vecteur, avec tout au plus un nombre constant d'entre eux stockés à l'extérieur du vecteur à tout instant.

▶ **Question 5** *Quelle est la complexité dans le pire des cas de cet algorithme ?*

▶ **Question 6** *Écrivez en CamlLight une fonction* tri_insert v *qui trie un tableau* v *en suivant l'algorithme de tri par insertion.*

```
tri_insert : 'a vect -> 'a vect
```

4 Le tri rapide (« quicksort »)

Le tri rapide (ou tri de *Hoare*, du nom de son inventeur) est le plus souvent utilisé pour sa complexité moyenne en $n\log(n)$. Dans cette partie, nous considérons un vecteur v de taille v. Le tri rapide consiste d'abord à séparer v en trois morceaux :
– un pivot x, par exemple le premier élément de v,
– un sous-tableau v1 constitué des éléments de v inférieurs à x,
– un second sous-tableau v2, constitué des éléments de v supérieurs à x.

On trie alors récursivement v1 et v2 et on concatène v1, x et v2 pour obtenir le tableau v, trié.
Maintenant, nous allons couper v en deux, en prenant le premier élément de v comme pivot.

▶ **Question 7** *Écrivez une fonction* decoupe *qui réorganise le sous-vecteur* v(a .. b) *de telle sorte que tous les éléments d'indice inférieur à* c *soient inférieurs à* x = v.(a)*, tous les éléments d'indice supérieur à* c *soient supérieurs à* x *et enfin que* v.(c) = x. decoupe *devra retourner* c *en résultat.*

```
decoupe : 'a vect -> int -> int -> int
```

Indication : on pourra parcourir v *à la fois de gauche à droite et de droite à gauche, en échangeant un élément de gauche avec un élément de droite. On placera* x *après avoir fait cela.*

▶ **Question 8** *En s'aidant de la fonction précédente, écrivez une fonction qui trie le sous-vecteur* v(a .. b) *par ordre croissant.*

```
tri_partiel : 'a vect -> int -> int -> unit
```

▶ **Question 9** *Finissez d'écrire la fonction de tri rapide. S'il vous reste du temps, modifiez les fonctions précédentes pour pouvoir utiliser une fonction de comparaison générique.*

```
tri_rapide : 'a vect -> unit
```

5 Sélection en temps linéaire

Formellement, le problème de la sélection peut être défini de la façon suivante :
Considérons un ensemble A de n nombres distincts, et un entier i tel que $1 \leq i \leq n$. On cherche l'élément x qui soit plus grand qu'exactement $i-1$ éléments de A.
On remarque que si le tableau A est trié, il suffit de prendre le i-ème élément de A. On peut donc facilement résoudre le problème en temps $O(n \log n)$, car on sait trier A en ce temps. Cependant, nous allons utiliser un algorithme SELECTION plus intelligent, pour obtenir ce même résultat en temps $O(n)$.
SELECTION va partitionner récursivement le vecteur donné en entrée, mais en garantissant le bon découpage du tableau. S'il y a un seul élément dans le tableau, on peut immédiatement retourner cet unique élément.

1. On découpe A en $\lfloor \frac{n}{5} \rfloor$ groupes de 5 éléments chacun, plus éventuellement le groupe constitué de n mod 5 éléments restants,
2. on trouve le médian de chacun des $\lceil \frac{n}{5} \rceil$ groupes en commençant par trier par insertion les éléments du groupe (il y en a 5 au plus), puis en prenant le médian de la liste triée des éléments du groupe,
3. on utilise SELECTION de façon récursive pour trouver le médian x de $\lceil \frac{n}{5} \rceil$ médians trouvés à l'étape 2 (s'il y a un nombre pair de médian, x sera le médian inférieur, par convention),
4. on partitionne le vecteur originel autour du médian des médians x à l'aide de la version modifiée de PARTITION. Soit k le nombre d'éléments de la région inférieure de la partition, augmenté de 1 ; ainsi x est le k-ème plus petit élément et il y a $n-k$ éléments dans la région haute de la partition,

5. Si $i = k$, alors on retourne x ; sinon, on utilise SELECTION récursivement pour trouver le i-ème plus petit élément de la région inférieure si $i < k$, ou le $(i - k)$-ème plus petit élément de la région supérieure si $i > k$.

▶ **Question 10** *Reprenez la fonction de tri par insertion* tri_insert_partiel *pour qu'elle puisse trier sur place les éléments d'indice entre* m *et* n−1 *et qui renvoie l'élément médian. Vous pouvez commencer par écrire la fonction sans prendre en compte la contrainte* m <= i < n.

```
tri_insert_partiel : 'a vect −> int −> int −> 'a
```

▶ **Question 11** *Écrivez une fonction* partition *telle que* partition v e m n *place tous les éléments de* v *plus grands que* e *à la fin, et les éléments plus petits au début. L'ensemble du tableau ne sera pas considéré, seuls les éléments d'indices compris entre* m *et* n−1 *doivent être pris en compte. Le résultat renvoyé doit être l'indice de l'élément séparateur* e*, le tableau doit être modifié en place et il n'y a pas besoin de le trier.*

```
partition : 'a vect −> 'a −> int
```

▶ **Question 12** *En utilisant les fonctions précédentes, écrivez une fonction* selection v index *qui renvoie l'élément qui aurait l'indice* index *si le tableau était trié. La complexité totale de l'algorithme doit être en* $O(n)$.

```
selection : 'a vect −> unit
```

▶ **MP & MP* – Option Informatique**
Second TP Caml
Mardi 25 septembre et 2 octobre 2007

Références et programmation impérative

Un corrigé

▶ Question 1

```
let make_matrix = fun n m e ->
    let x = make_vect n (make_vect m e) in
    for i = 0 to (n−1) do
        x.(i) <− make_vect m e
    done;
    x;;
```

▶ Question 3

```
let echange = fun v i j ->
    let x = v.(i) in
    v.(i) <− v.(j);
    v.(j) <− x;;
```

▶ Question 4

```
let tri_bulle = fun v ->
    let condition = ref true in
    let n = vect_length v in
    while !condition do
        condition := false;
        for i = 0 to (n − 2) do
            if(v.(i) > v.(i+1)) then (
                echange v i (i+1);
                condition := true;
            ) done;
    done;
    v;;
tri_bulle [| 4 ; 2 ; 8 ; 10 ; 2 |];;
```

▶ **Question 5** La complexité dans le pire des cas est en $O(n^2)$ et est atteinte quand le tableau est trié par ordre décroissant.

▶ Question 6

```
let tri_insert = fun v ->
    let n = vect_length v in
    let clef = ref v.(0) in
    let j = ref 0 in
    for i = 1 to n−1 do (
        clef := v.(i);
        j := i − 1;
        while (!j >= 0 && v.(!j) > !clef) do (
            v.((!j)+1) <− v.(!j);
            decr j;
        ) done;
        v.((!j)+1) <− !clef;
    ) done;
    v;;
tri_insert [| 2 ; 3 ; 7 ; 2 ; 1 ; 9 ; 8 ; 4 |];;
```

▶ Question 7

```
let decoupe = fun v a b ->
    let i = ref(a+1) and j = ref b and x = v.(a) in
    while !i <= !j do
        while(!i <= !j) & (x >= v.(!i)) do i := !i+1 done;
        while(!i <= !j) & (x <= v.(!j)) do j := !j−1 done;
        if(!i < !j) then (
            echange v !i !j;
            i := !i+1;
            j := !j−1;
        )
        else if !i = !j then j := !j−1;
    done;
    if a <> !j then echange v a !j;
    !j
;;
```

▶ Question 8

```
let rec tri_partiel = fun v a b ->
    let c = decoupe v a b in
    if a < c−1 then tri_partiel v a (c−1);
    if c+1 < b then tri_partiel v (c+1) b;
```

▶ Question 9

```
let tri_rapide = fun v ->
    let n = vect_length v in
    if (n >= 2) then tri_partiel v 0 (n−1);;
tri_rapide [| 4 ; 5 ; 1 ; 3 ; 10 ; 3 |];
```

▶ Question 10

```
let tri_insert_partiel = fun v m n ->
    let clef = ref v.(0) in
    let j = ref 0 in
    for i = m + 1 to (n−1) do (
        clef := v.(i);
        j := i − 1;
        while (!j >= m && v.(!j) > !clef) do (
            v.((!j)+1) <− v.(!j);
            decr j;
        ) done;
        v.((!j)+1) <− !clef;
    ) done;
    v.(((n−1) − m) / 2 + m)
;;
```

▶ Question 11

```
let partition = fun v e m n ->
    let low = ref m in
    let high = ref (n - 1) in
    while !low < !high do (
        if(v.(!low) < e) then incr low
        else if(v.(!high) > e) then decr high
        else let tmp = v.(!low) in (
            v.(!low) <- v.(!high);
            v.(!high) <- tmp;
        );
    ) done;
    !low
;;
```

▶ Question 12

```
let ceil = fun n d ->
    if d * (n / d) = n then n / d
    else (n / d) + 1
;;
let rec selection_aux = fun v index m n ->
    if(n - m = 1) then v.(m)
    else (
        let k = ref m in
        let i = ref 0 in
        let c = ceil (n - m) 5 in
        let sub_vect = make_vect c v.(0) in
        while (!k) < n do (
            sub_vect.(!i) <- tri_insert_partiel v (!k) (min n (5 + !k));
            i := 1 + !i;
            k := 5 + !k;
        ) done;
        let x = selection_aux sub_vect (c / 2) 0 c in
        let k = partition v x m n in
        if(k = index) then v.(k)
        else if(k < index) then selection_aux v index (k+1) n
        else selection_aux v index m (k)
    )
;;
let selection = fun v index ->
    selection_aux v index 0 (vect_length v)
;;
selection [|1 ; 3 ; 6 ; 4 ; 2 ; 5 ; 0|] 3;;
```