TRAVAUX PRATIQUES XIII

Trier une liste

Ce TP vise à recenser divers algorithmes de tris, et à les mettre en oeuvre sur des listes OCaml. C'est donc aussi et surtout un TP de révision de manipulations de listes OCaml.

Vous devez tous faire la partie sur le tri fusion. Je vous conseille de prévoir 45min.

Ce TP est peu guidé. Il faut que vous expérimentiez, et que vous m'appeliez en cas de problème. J'ai de sjeux de cartes à disposition si vous voulez les manipuler.

A Tri par sélection

Revoyons tout d'abord le tri par sélection. Son principe est le suivant : sélectionner le maximum du tableau, et l'échanger avec la dernière case du tableau. Puis sélectionner le maximum du tableau privé de sa dernière case, et l'échanger avec l'avant-dernière case. Etc. Ainsi, quand on recherche la valeur à mettre dans la case d'indice `i`, on cherche le maximum du tableau entre les indices 0 et `i` inclus. [1]

Pour épicer un peu le tout, on va travailler non pas sur des entiers mais sur des points du plan. Un point est défini par ses coordonnées (x, y). On dit que $(x, y) \leq (x', y')$ si $y < y'$ ou si $y = y'$ et $x < x'$. On se donne le type structure suivant en C :

C
```c
struct point_s {
  int x;
  int y;
};
typedef struct point_s point;
```

Remarque :À partir de maintenant, quand une fonction attend un tableau `T` en argument, on ne précisera plus sa taille : ce n'est pas obligatoire. On peut ainsi déclarer une fonction via `int f(int T[], int nb_de_cases)` plutôt que via `int f(int T[1000], int nb_de_cases)`. C'est plus pratique.

1. Écrire une fonction `int indice_pt_max(point T[], int i)` qui prend en argument un tableau de points `T` et son nombre de cases `i`, et renvoie l'indice du point maximal de ce tableau.
2. En déduire une fonction `void tri_selection(point T[], int n)` qui trie un tableau de points en utilisant un tri par sélection.

Le tri sélection effectue un nombre quadratique de comparaisons.

B Tri rapide

B.1 Rappels OCaml

Une liste est un objet défini récursivement (qui correspond à une listre simplement chainée). Une liste est soit :

- `[]`, la liste vide.
- `h::t` avec `h` un élément (nommé « tête », le début de la liste) et `t` une autre liste (nommée « queue », la suite de la liste).

On peut faire cette disjonction de code à l'aide d'un `match ... with` :

OCaml
```ocaml
(** Fonction qui renvoie le maximum d'une liste. *)
let rec max_list l = match l with
  | [] -> failwith "Liste Vide"
```

1. On peut symétriser et travailler avec les minimums.

```
4    | h::t when t = [] -> h
5    | h::t -> max h (max_list t)
```

On rappelle aussi que les n-uplets existent en OCaml. Par exemple, `(0,1)` est la paire contenant 0 et 1. On rappelle enfin qu'il est utile de faire un let déstructurant pour accéder au contenu d'un n-uplets. Par exemple, si `c` est une paire, `let (a,b) = c` permet de donner un nom aux coordonnées de `c` (marche également avec un `let in` ou un filtrage).

Enfin, utiliser `utop` pour tester ses codes est très important.

B.2 Le tri

On veut maintenant implémenter un tri sur les listes en OCaml. L'idée du tri rapide est le suivant :

— Regarder la valeur de la tête de la liste. Créer deux sous-listes : la première contient les éléments qui sont plus petits que la tête, la seconde ceux qui sont plus grands.

— Trier récursivement chacune des deux moitiés.

— Les mettre bout à bout. The end.

3. Écrire une fonction `split : 'a -> 'a list -> 'a list * 'a list` qui prend en entrée une valeur `p` et une liste `l`, et renvoie une paire `inf,sup` de listes. `inf` contient les éléments de `l` strictement inférieurs à `p`, `sup` les autres.
4. En déduire une fonction qui implémente un tri rapide. On pourra utiliser la fonction `append` du module `List` pour « mettre bout à bout » des listes.
5. (Bonus) Comparer l'efficacité avec la fonction `fast_sort` du module `List`. On pourra utiliser la commande terminal `time` pour mesurer le temps d'exécution d'un programme.
   Il faudrait aussi générer des listes aléatoires. Vous pourrez faire cela en fin de TP si vous avez du temps. Il vous faudra aller chercher en ligne comment faire.

Le tri rapide effectue un nombre quadratique de comparaisons.

Cependant, ce pire cas est très rare et en pratique ce tri est un des plus efficaces.

Nous reviendrons sur ce tri au S2.

C Tri par insertion

L'idée du tri par insertion est de construire au fur et à mesure la liste triée en parcourant la liste d'origine. Chaque élément de la liste d'origine est insérée au bon endroit dans la liste triée (c'est à dire avant les éléments qui lui sont plus petits et après ceux qui lui sont plus grands).

6. Écrire une fonction `insere : 'a list -> 'a` qui prend en argument une liste triée et un élément et renvoie la liste obtenue en insérant cet élément à sa place dans la liste.
7. En déduire une fonction qui réalise un tri par insertion.

Le tri rapide effectue un nombre quadratique de comparaisons.

Il est cependant très efficace sur les petits tableaux, c'est pourquoi il est utilisé comme « cas de base » de nombreux tris récursifs (quand il n'y a plus beaucoup d'éléments, on trie par insertion).

D Tri Fusion

L'idée du tri fusion est le suivant :

— Couper la liste en deux moitié égales, arbitrairement. Si il y a un nombre impair d'éléments, une des deux listes peut avoir un élement de plus que l'autre.

— Trier récursivement chacune des deux moitiés.

— Les fusionner en une seule liste triée. The end.

Par exemple :

— On veut trier `[1;-2;5;3;0]`. On la coupe arbitrairement en deux : `1; 5; 0` et `-2; 3` (ici j'ai coupé selons indices pairs et impairs : c'est arbitraire).

— On trie récursivement chacune des deux moitiés. On obtient `[0; 1; 5]` et `[-2; 3]`.

— On va fusionner ces deux moitiés :

- On compare les têtes des deux moitiés. La plus petite est -2 : la liste finale commence donc par `(-2)::la_suite`.
  La suite est la fusion de `[0;1; 5]` et `[3]`.
- On fusionne ces deux listes-ci (récursivement, mais je vais ici détailler cette récursion). `0` est la plus petit, donc la fusion de ceux deux listes-ci est `0::...`.
  Ce `...` est la fusion de `[1; 5]` et `[3]`.
- On fusionne ces deux nouvelles listes. On obtient `1::...`.
  Ce `...` est la fusion de `[5]` et `[3]`.
- On maintenant `3::...`.
  Ce `...` est la fusion de `[5]` et de `[]`.
- Cas de base facile à gérer : si une liste est vide, la fusion est simplement l'autre : `[5]`.
- De sortie d'appel en sortie d'appel, on déduit que la fusion des deux listes initiales est `(-2)::0::1::3::[5]`, soit `[-2;0;1;3;5]`.

La première étape (la séparation) est un peu technique à inventer en OCaml. Je vous en donne ici une réalisation :

OCaml
```
let rec separe l = match l with
  | [] -> ( [] , [] )
  | [h] -> ( [h] , [] )
  | h0 :: h1 :: t -> let (g,d) = separe t in
                     (h0::g, h1::d)
```

La dernière étape (la fusion) est également technique, c'est pourquoi je l'ai détaillée précédemment.

8. Vérifiez que vous comprenez comment `separe` fonctionne. Appelez-moi si vous avez un doute.

9. Écrire une fonction `fusionne` qui prend en argument deux listes triées et les fusionne en une seule.
On devra faire attention à bien couvrir tous les cas de base possibles.

10. En déduire un tri fusion.

Le tri fusion trie n éléments en faisait $n \log n$ comparaisons.

E Pour occuper les plus rapides

11. Débloquez le niveau 3 de France-IOI. Si c'est déjà fait, débloquez le 4. Et ainsi de suite. [2]

2. Plus facile à dire qu'à faire.