Colle 4 : Listes Caml et tris

1 Tris simples

On s'intéresse au problème de trier une liste. Nous supposerons ici qu'il s'agit d'une liste d'entiers, et on veut l'ordonner selon l'ordre naturel sur les entiers, de manière croissante.

▷ **Question 1.** Ecrire une fonction `minimum` qui prend en entrée une liste et renvoie la valeur du plus grand élément ainsi que la liste privée de cet élément.

Ecrire une fonction `triselec` qui prend en entrée une liste et qui la trie en sélectionnant successivement le plus petit élément de la liste en entrée et en l'insérant dans la liste résultat triée.

Y aurait-il une différence si on travaillait avec une fonction `maximum` plutôt que `minimum`? ◁

▷ **Question 2.** Écrire une fonction `insertion_tri` qui, étant donnés une liste triée `liste` et un élément `elt`, renvoie la liste triée contenant `elt` ainsi que les éléments de `liste` (on parle d'insérer un élément dans une liste triée).

En déduire (et écrire) une fonction qui effectue le tri par insertion pour trier une liste. ◁

▷ **Question 3.** Ecrire une fonction qui prend en entrée deux listes triées et qui renvoie une liste dont les éléments sont la réunion des éléments des deux suites prises en entrée et qui est elle même triée.

En déduire un algorithme de tri basé sur le principe diviser pour régner.

Comparer la complexité de cet algorithme avec celle des deux précédents. ◁

2 Le tri casier

Nous verrons en cours une preuve du fait que tout tri dit "par comparaison" c'est-à-dire qui ne peut utiliser que des tests de comparaison pour ordonner les éléments nécessite une compexité temporelle d'au moins $O(n\log(n))$. Cependant, si on dispose d'informations supplémentaires sur nos entrées, on peut parfois trier en temps linéaire. Nous vous présentons ici le tri casier qui en est un exemple. Ce tri fonctionne pour trier un tableau d'entiers naturels.

Voici son principe :

1. On détermine le plus grand élément du tableau noté M.
2. On crée un tableau `occ` de taille $M+1$ rempli initialement de zéros.
3. On parcourt le tableau à trier afin de compléter `occ` de telle sorte que `occ.(i)` contienne le nombre de fois où la valeur i apparait dans le tableau à trier.
4. On modifie le tableau à trier à l'aide de ces informations.

▷ **Question 4.** Ecrire une fonction `tri_casier : int array -> unit =<fun>` qui trie le tableau t qui lui est passé en entrée. ◁

3 QUICKSORT

Continuons avec le tri rapide.

Voici le principe : étant donné une liste sous la forme `a::liste`, on se sert de `a` comme pivot. On construit deux listes, `linf` et `lsup`, la première contenant les éléments de `liste` strictement inférieurs à `a`, et la seconde tous les autres. On trie récursivement `linf` et `lsup`, puis on concatène les résultats.

▷ **Question 5.** Écrire une fonction `pivot` qui, étant donné un élément `a` et une liste `liste`, renvoie le couple de listes (`linf`,`lsup`) comme décrit ci-dessus.

En déduire (et écrire) une fonction qui effectue un tri rapide pour trier une liste. ◁

Un inconvénient du tri rapide précédent est la gestion de l'espace mémoire : on crée de nouvelles listes à chaque appel récursif ce qui se révèle assez gourmand. Nous allons réécrire un tri rapide en utilisant un tableau plutôt qu'une liste. On cherchera à partitionner le tableau passé en entrée en place : on ne crée pas de nouveau tableau dans l'étape de partition mais on réorganise le tableau passé en entrée. Nous allons présenter deux algorithmes différents qui permettent de faire cette partition.

Voici le code principal du tri rapide sur un tableau :

```
let tri_rapide tab=
   let rec quick deb fin =
      if deb < fin then
         begin let k = separation tab deb fin in
                   quick deb (k-1)
                   quick (k+1) fin end in
quick 0 (Array.length tab -1);;
```

3.1 LOMUTO

La première technique de séparation fonctionne de la manière suivante : pour séparer le tableau entre les cases `deb` et `fin` on suit les étapes suivantes :

1. On choisit `tab.(fin)` comme pivot.
2. On initialise une variable k à deb et pour i variant de `deb` à `fin-1` on veut maintenir l'invariant suivant :
   - les éléments d'indices `deb` à $k-1$ sont inférieurs au pivot.
   - les éléments d'indice k à i sont supérieurs au pivot.
3. Ainsi, si `tab.(i)` est supérieur au pivot, on ne fait rien et on poursuit la boucle et sinon on échange les termes en positions k et i et on incrémente k.
4. Quand la boucle sur i est terminée, on place le pivot à sa place en échangeant les positions k et `fin`.
5. On renvoie la position du pivot : k.

▷ **Question 6.** Coder la fonction de partition qui suit la méthode de Lomuto. On obtiendra une fonction `separation : int array-> int ->int -> int.` ◁

3.2 HOARE

Voici la méthode originale de séparation. On crée deux indices de parcours initialisés respectivement au début et à la fin de la zone à partitionner. Ces indices vont se rejoindre en échangeant au fur et à mesure les éléments de gauche de valeurs supérieures au pivot et les éléments de droite de valeurs inférieures au pivot. Ici on crée deux indices i et j et on fait respecter l'invariant suivant : "les termes d'indices `deb` à $i-1$ sont inférieurs à la valeur de pivot et ceux entre $j+1$ et `fin` lui sont supérieurs". On ne progressera que tant que $i \leq j$.

▷ **Question 7.** Ecrire une fonction `seprationH : int array -> int -> int ->int` qui sépare suivant cette nouvelle méthode. ◁

3.3 Efficacité du tri rapide

▷ **Question 8.** Déterminer la complexité du tri rapide dans le pire cas. Commenter. ◁

Une méthode pour contourner ce problème consiste à prendre le pivot au hasard dans le tableau. Une seconde méthode consisterait à prendre la médiane mais il n'existe pas d'algorithme efficace qui accomplit cette tâche, cependant on peut calculer une pseudo-médiane en temps linéaire c'est-à-dire une valeur dont on sait qu'au moins une proportion significative de valeurs lui sont inférieures tout comme une proportion significative lui sont supérieures.

4 Tris et permutations

1. Les cours d'algèbre nous enseignent que toute permutation de l'ensemble $\{1, \ldots, n\}$ peut se décomposer en un produit de transpositions. Pouvez-vous établir ce résultat en analysant l'algorithme de tri par sélection?
2. Les mêmes cours d'algèbre nous enseignent qu'il suffit de composer au plus $n-1$ transpositions pour représenter une permutation sur $\{1, \ldots, n\}$. Pouvez-vous établir ce résultat en analysant de nouveau l' algorithme de tri par sélection?
3. Tris et permutations circulaires.

   Une permutation circulaire de $0, \ldots, n-1$ est une permutation σ telle que pour tout $i \in \{0, \ldots, n-1\}$, $\sigma(i) = (i+k) \mod n$ où $k \in \{0, \ldots, n\}$ est le paramètre de la permutation.

   (a) Toute permutation est-elle une permutation circulaire ?

   (b) Proposer un algorithme qui prend en argument un tableau t *obtenu en appliquant une permutation circulaire à un tableau trié* et qui retourne la position (pas la valeur) du maximum dans ce tableau t. Ecrire la fonction `max_circ : int array -> int`

   (c) Proposer un algorithme qui prend en argument un tableau t *obtenu en appliquant une permutation circulaire à un tableau trié* et qui en utlisant la position du maximum dans ce tableau renvoie un tableau t' contenant les éléments du tableau t triés. Ecrire la fonction `tri_circ : int array -> int array`

   (d) Proposer un algorithme qui trie un tableau de taille n *obtenu en appliquant une permutation circulaire à un tableau trié* et qui effectue au plus $O(\log n)$ comparaisons d'éléments (pour cette dernière question il n'est pas demandé d'écrire une fonction Caml mais simplement d'expliquer la démarche algorithmique et de justifier la complexité).