TP1 - Ordre supérieur et itérateurs

Le tri fusion

Le but de cette partie est d'implémenter une grande partie de ce qui a été fait lors des deux premiers TD, en y ajoutant quelques nouveautés.

On pensera à tester, pour chaque fonction implémentée, qu'elle est bien typée et qu'elle effectue bien ce qu'on veut. L'interpréteur OCaml est là pour ça.

Couper

On veut implémenter la fonction `couper: 'a list -> 'a list * 'a list` telle que `couper l` coupe une liste `l` en deux sous-listes plus petites que `l`, dont les longueurs diffèrent au plus d'un. Il existe deux façons différentes de le faire :

- En distinguant les cas où la liste est vide, composée d'un seul élément ou composée d'au moins deux éléments ;
- En alternant l'ajout des éléments entre les deux listes qui seront retournées en résultat (système de bascule).

Implémenter les deux versions puis utiliser l'itérateur `List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` pour implémenter une troisième version.

Fusion

Implémenter en forme récursive terminale la fonction `fusion: ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list` telle que `fusion comp l1 l2` fusionne les deux listes `l1` et `l2` supposées triées et renvoie une liste triée selon la fonction de comparaison `comp`.

Tri fusion

En utilisant les fonctions précédentes, implémenter la fonction `trier: ('a -> 'a -> int) -> 'a list -> 'a list` telle que `trier comp l` trie la liste `l` selon l'ordre défini par la fonction `comp`

Création d'une liste quelconque

Implémenter une fonction `make_list: int -> int list` telle que `make_list n` retourne une liste de longueur `n` remplie aléatoirement par des entiers compris entre `0` (inclus) et `n-1` (exclus) *(on remarquera que cette construction permet de s'assurer la présence d'au moins un doublon de valeurs)*

Test de stabilité

On veut savoir si le tri fusion est stable.

La stabilité d'un algorithme de tri est déterminée par le fait que deux éléments considérés comme égaux doivent garder le même ordre qu'ils avaient dans la liste d'origine dans la liste triée.

Considérons, par exemple, la liste de caractères `['c'; 'b'; 'a'; 'c']`. La liste triée (par ordre croissant) renvoyée par un algorithme de tri est `['a'; 'b'; 'c'; 'c']`. Cependant, rien ne garantit que la première occurrence du caractère `'c'` dans cette liste corresponde à la première

occurrence de ce même caractère dans la liste de départ.

Pour le savoir, considérons, la liste de paires `[('c',1); ('b',2); ('a',3); ('c',4)]`, où chaque caractère est associé à un numéro unique. Pour retrouver le même tri que sur les listes de caractères, on compare ces paires par rapport à la première composante. *(Cette liste est une liste d'associativité, c'est à dire une liste qui associe à chaque clé (premier élément), une valeur (second élément).)*

L'algorithme de tri est :

- **stable** si la liste triée est `[('a',3); ('b',2); ('c',1); ('c',4)]`
- **instable** si la liste triée est `[('a',3); ('b',2); ('c',4); ('c',1)]`

Numérotation

Afin de vérifier la stabilité, il va falloir associer à chaque élément de la liste d'origine son ordre d'apparition dans la liste.

Implémenter à l'aide de l'itérateur `List.fold_left` la fonction `numerotation: int list -> (int * int) list` *(les parenthèses autour du `int * int` sont importantes sinon on lirait "qui renvoie un couple composé d'un entier et d'une liste d'entiers")* telle que `numerotation l` associe, dans l'ordre, à chaque élément de `l` son indice.

Fonction de comparaison

Implémenter la fonction `comp_assoc: int * 'a -> int * 'a -> int` qui compare deux paires selon leurs clés par ordre croissant (`comp_assoc (1, 4) (2, 1) < 0`).

Affichage

Afin de se débarasser de l'interpréteur, implémenter à l'aide de l'itérateur `List.iter: ('a -> unit) -> 'a list -> unit` la fonction `affiche_liste: (int * int) list -> unit` pour afficher chaque paire d'une liste d'associativité.

Stabilité du tri fusion

Utiliser les fonctions implémentées pour déterminer la stabilité du tri fusion.

D'après les tests effectués, le tri par fusion est :

○ Stable ? ○ Instable ?

Vérifier la réponse ☐

Empaquetage

Comme on l'a vu lors du TD 2, le fait de donner aux différentes fonctions manipulant une liste triée la fonction de comparaison ayant permis ce tri peut conduire à des erreurs à l'exécution si la fonction de comparaison donnée en argument n'est pas la bonne.

On travaillera donc avec un type enregistrement `type 'a maliste = {l : 'a list; comp : 'a -> 'a -> int}` permettant de mettre dans un même paquet une liste et la fonction servant à la trier.

Tri

En utilisant la fonction de tri déjà implémentée, implémenter la fonction `trier_ml: 'a maliste -> 'a maliste` telle que `trier_ml ml` trie la liste `ml.l` selon l'ordre défini par la fonction `ml.comp`.

Recherche dans une liste triée

Implémenter la fonction `recherche_triee_ml: 'a maliste -> 'a -> bool` telle que `recherche_triee_ml e ml` retourne `true` si `e` est dans `ml.l` selon l'égalité définie par `ml.comp` et `false` sinon.

Liste triée et recherche associée

Il n'est plus utile de garder la fonction de comparaison une fois que la liste a été triée. Il est bien plus utile de créer un nouveau type enregistrement `liste_triee` contenant à la fois une liste triée et une fonction de recherche.

- Définir le type `liste_triee`.
- Implémenter une fonction `trier_et_constr: 'a maliste -> 'a liste_triee` telle que `trier_et_constr ml` renvoie une valeur de type `'a liste_triee`.

Pour continuer

D'une certaine façon, on vient de créer une classe contenant une liste triée en attribut et une méthode de recherche d'élément.

On peut vouloir ajouter de nouvelles fonctions à notre type `'a liste_triee` *(et modifier en conséquence `trier_et_constr`)*.

Voici une liste non exhaustive de fonctions qu'il peut être intéressant d'ajouter :

- `min: 'a list -> 'a` telle que `min lt` retourne l'élément minimal de `lt` *(on prendra soin de gérer les cas absurdes)*;
- `count: 'a -> 'a list -> int` telle que `count e lt` compte le nombre d'occurences de `e` dans `lt`;
- `delete_double: 'a list -> 'a list` telle que `delete_double lt` supprime tous les doublons de `lt` ;
- `add: 'a -> 'a list -> 'a list` telle que `add e lt` ajoute l'élément `e` dans `lt`, la liste résultante doit rester triée ;
- `append: 'a list -> 'a list -> 'a list` telle que `append lt1 lt2` concatène les deux listes `lt1lt2`>, la liste résultante doit rester triée ;
- ...