Devoir surveillé 3

Autour du tri-fusion

Option informatique MPSI1 & MPSI2

Dans ce problème, on s'intéresse au problème du tri de listes.
Les éléments des liste sont comparés par une fonction `plusGrand` telle que `plusGrand a b` renvoie `true` si et seulement si a est strictement plus grand que b. Dans les exemples, on utilisera des entiers :

Comparaison d'entiers

```
let plusGrand a b =
  a > b;;
```

mais on discutera aussi des cas où des éléments distincts peuvent être indiscernables par la relation d'ordre, par exemple dans le cas de couples formés d'un entier et d'un élément d'un autre type

Comparaison de couples

```
let plusGrand a b =
  fst a > fst b;;
```

Définition 1 : Stabilité

Un tri est stable si les éléments égaux pour la relation d'ordre gardent l'ordre qu'ils avaient avant le tri dans le résultat trié.

Par exemple, la liste `[(4, "tour"); (6, "maison"); (4, "banc"); (2, "un")]` doit être trié sous la forme `[(2, "un");(4, "tour"); (4, "banc"); (6, "maison")]` par un tri stable.
Un tri non stable pourrait donner `[(2, "un"); (4, "banc"); (4, "tour"); (6, "maison")]`

I Tri fusion de listes

I.1 Tri classique stable

Tri-fusion

```
let rec triFusion liste =
  match liste with
  |[] -> []
  |[x] -> [x]
  |_ -> let l1, l2 = separation liste in
        let t1 = triFusion l1 in
        let t2 = triFusion l2 in
        fusion t1 t2;;
```

Le tri utilise deux fonctions auxiliaires que l'on va écrire pour assurer la stabilité.
La séparation vue en cours distribue les éléments de la liste alternativement dans les deux listes ; la stabilité est impossible. On va utiliser un découpage de liste selon une nombre de termes.

Question 1 - Longueur
Écrire en OCaml une fonction `long : 'a list -> int` qui calcule la longueur d'une liste.

Question 2 - Découpage
Écrire en OCaml une fonction `decoupage : int -> 'a list -> 'a list * 'a list` telle que `decoupage k liste` renvoie deux listes dont la première contient les k premiers éléments de la liste initiale et la seconde contient les éléments restant, dans le même ordre.

`separation 4 [4; 8; 2; 6; 3; 1]` doit renvoyer `[4; 8; 2; 6], [3; 1]`.
`separation 7 [4; 8; 2; 6; 3; 1]` peut renvoyer `[4; 8; 2; 6; 3; 1], []` ou bien renvoyer une erreur, le choix est libre.

Question 3 - Séparation
Écrire en OCaml une fonction `separation : 'a list -> 'a list * 'a list` qui renvoie deux listes `l1` et `l2` de tailles égales ou ne différant que de 1 et telles que `l1 @ l2` redonne la liste initiale.

Question 4 - Fusion
Écrire en OCaml une fonction `fusion : int list * int list -> int list` qui reçoit deux listes qu'on suppose triées et qui renvoie une liste triée contenant tous les éléments des deux listes initiales. Pour assurer la stabilité du tri la fonction doit placer le terme de la première liste en cas d'égalité.

On admet que le nombre maximal de comparaisons effectuées lors de la fusion de deux listes de tailles n_1 et n_2 est $n_1 + n_2 - 1$.
On note $C(n)$ le nombre maximal de comparaisons effectuées pour le tri d'une liste de taille n.

Question 5 - Complexité dans un cas particulier
Prouver que si n est pair, $n = 2 \cdot m$ alors $C(n) = 2 \cdot C(m) + n - 1$.
En déduire que $C(2^p) = p \cdot 2^p - 2^p + 1$.

I.2 Utilisation d'une file d'attente

Nous allons utiliser une structure de file d'attente d'objets de type `'a` dont le type sera noté `'a queue`. Les fonctions que nous utiliserons sont :

- `createQueue : 'a -> 'a queue`, pour créer une file vide,
- `isEmptyQueue : 'a queue -> bool`, pour tester si une file est vide,
- `length : 'a queue ->int`, donne la longueur, le nombre d'éléments en attente, d'une file,
- `enque : 'a queue -> 'a -> unit`, `enque f x` ajoute x à la file f,
- `first : 'a queue -> 'a` renvoie le premier élément d'une file, **sans l'enlever**
- `deque : 'a queue -> unit`, retire le premier élément d'une file, **sans le renvoyer**.

La partie III proposera une implémentation efficace d'une file.

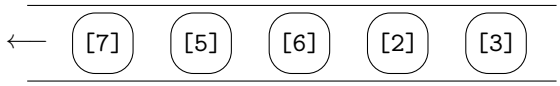
Question 6 - Simplification
Écrire une fonction OCaml `take : 'a queue - > 'a` qui retire le premier élément d'une file **et** le retire de la file.

Pour trier une liste, on va utiliser, la méthode suivante.

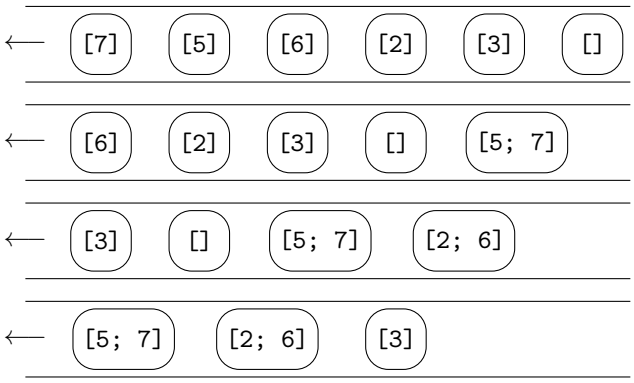
1. On crée une file vide
2. On insère tous les éléments de la liste dans la file sous la forme de liste à un éléments.
3. Tant que la vide contient au moins deux éléments,
   - si le nombre d'éléments est impair, on ajoute une liste vide
   - la longueur de file est paire, $2p$,
   - on répète p fois la boucle
     - on retire deux éléments, deux listes,
     - on en réalise la fusion,
     - on ajoute cette fusion dans la liste.
4. L'unique élément restant dans la liste est la liste triée.

Par exemple, si on part de la liste `[7; 5; 6; 2 ; 3]`

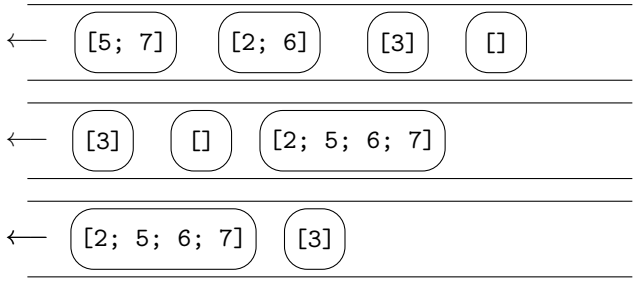
L'étape 2 donne



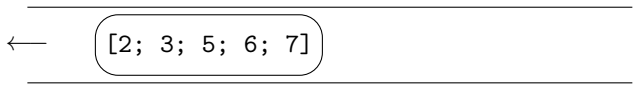
La première itération de la boucle **tant que** du 3) donne



La deuxième itération de la boucle **tant que** du 3) donne



La dernière itération de la boucle **tant que** du 3)fournit la liste triée



Question 7 - Stabilité
Quelle est l'utilité de l'ajout d'une liste vide en cas d'un nombre impair de listes ?

Question 8 - Écriture
Écrire la fonction OCaml `triFile : int list -> unit` qui implémente cette méthode. On utilisera la fonction `fusion` et les fonctions de files ci-dessus.

Question 9 - Complexité d'une boucle
Montrer que, lors d'un passage de la boucle **tant que**, le nombre total de comparaisons est, au maximum, $n - \left\lceil \frac{r}{2} \right\rceil$ où r est le nombre d'éléments dans la file au départ.

Question 10 - Complexité dans un cas particulier
Combien de comparaisons sont effectuées au maximum lors du tri d'une liste de taille 2^m ?

II Optimisation des fusions

Le tri écrit ci-dessus ne tient pas compte du fait qu'une liste peut être partiellement triée ; par exemple dans la liste `[4; 1; 3; 5; 6; 7; 2]`, la partie `[1; 3; 5; 6; 7]` va être, au moins partiellement, triée par des fusion.
On va proposer une technique qui permet d'éviter ces comparaisons inutiles.

Définition 2 : Croissance

Une croissance d'une liste d'entiers $[a_0; a_1; \ldots; a_{n-1}]$ est une portion croissante de la liste, de la forme $[a_i; a_{i+1}; \ldots; a_j]$ avec $0 \leqslant i \leqslant j < n$ et $a_i \leqslant a_{i+1} \leqslant \cdots \leqslant a_j$.

Question 11 - Croissance commençante
Écrire une fonction `croissance : int list -> int list` qui détermine la monotonie la plus longue possible formée des premiers éléments de la liste.

`croissance [4; 5; 5; 6; 2; 8]` renverra `[4; 5; 5; 6]`,
`croissance [7; 4; 5; 5; 6; 2; 8]` renverra `[7]`,
`croissance []` renverra `[]`, , `croissance [5]` renverra `[5]`

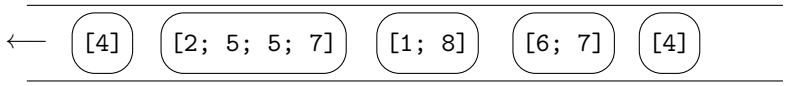
Question 12 - Décomposition initiale
Modifier la fonction précédente en une fonction `debutFin : int list -> int list * int list` qui renvoie la croissance initiale calculée ci-dessus et le reste de la liste.

`debutFin [4; 5; 5; 6; 2; 8]` renverra `[4; 5; 5; 6], [2; 8]`,
`debutFin [7; 4; 5; 5; 6; 2; 8]` renverra `[7], [4; 5; 5; 6; 2; 8]`,
`debutFin []` renverra `[], []`, , `debutFin [5]` renverra `[5], []`

Question 13 - Décomposition
En déduire une fonction `decompositionC : int list -> int list list` qui renvoie une liste de listes non vides dont chacune est une croissance de la liste initiale et dont la concaténation reconstruit la liste initiale. On cherche des croissances maximales.

`decomposition [4; 2; 5; 5; 7; 1; 8; 6; 7; 4]` doit renvoyer
`[[4]; [2; 5; 5; 7]; [1; 8]; [6; 7]; [4]]`.

On peut alors trier la liste en calculant la décomposition puis en appliquant l'algorithme de la partie I.2 en remplaçant le remplissage initial par les croissances issues de la décomposition.
Avec l'exemple ci-dessus l'étape 1 donne



Question 14 - Écriture
Écrire la fonction OCaml `triFile : int list -> unit` qui implémente cette méthode.

Question 15 - Un exemple de complexité
On suppose que la décomposition en croissances d'une liste de taille 2^m donne 2^r croissances de taille 2^{m-r}. Déterminer le nombre maximal de comparaisons effectuées lors du tri.

III Implémentation de files

Le langage OCaml implémente les files à l'aide cellules qui contiennent sont soit une cellule vide soit un enregistrement contenant une valeur et un lien vers une autre cellule. On retrouve une structure de liste mais, ici, le lien est mutable.

```
type 'a cellule = Vide | Cell of {contenu : 'a;
                                  mutable next : 'a cellule};;
```

Une file est maintenant définie par sa première cellule, comme une liste, mais on ajoute un lien vers la dernière cellule pour pouvoir facilement ajouter un élément ainsi que la longueur.

```
type 'a queue = {mutable len   : int;
                 mutable debut : 'a cellule;
                 mutable fin   : 'a cellule};;
```

Une file vide sera donc de longueur nulle avec `debut` et `fin` qui sont des cellules vides.

Question 16 - Première fonctions
Écrire les fonction `createQueue`, `isEmptyQueue`, `length` et `first` définie dans la partie I.2.

Pour ajouter un élément x à une file,

- on commence par créer une cellule avec x de suivant vide
  `let c = Cell {contenu = x; next = Vide}`
- si la file est vide, on remplace `debut` et `fin` par cette cellule, on change la longueur,
- sinon la dernière cellule de la file va devoir envoyer vers `c` et la dernière cellule de la file devient `c` et on change la longueur.

Question 17 - Adjonction
Écrire les fonction `enque` définie dans la partie I.2.

Question 18 - Retrait
Écrire les fonction `deque` définie dans la partie I.2.

La structure définie est très riche, on peut par exemple concaténer deux files en temps constant.

Question 19 - Concaténation
Écrire une fonction fonction `union : 'a queue -> 'a queue -> unit` telle que `union f1 f2` ajoute à `f1` tous les éléments de `f2` (à la fin) en temps constant.

Solutions

Solution de la question 1 - Longueur

```
let rec long liste =
  match liste with
  |[] ->0
  |t::q -> 1 + (long q);;
```

Solution de la question 2 - Découpage

```
let rec decoupage k liste =
  match liste with
  |[] -> [],[]
  |t::q -> if k = 0
           then [], liste
           else let l1, l2 = decoupage (k-1) q in
                (t::l1), l2;;
```

Solution de la question 3 - Séparation

```
let separation liste =
  let n = long liste in
  |[] -> [],[]
  separation (n/2) liste;;
```

Solution de la question 4 - Fusion

```
let rec fusion l1 l2 =
  match l1,l2 with
  |[], _ -> l2
  |_, [] -> l1
  |t1::q1, t2::q2 -> if plusGrand t2 t1
                    then t1::(fusion q1 l2)
                    else t2::(fusion l1 q2);;
```

On effectue au plus $n_1 + n_2 - 1$ comparaisons.

Solution de la question 5 - Complexité dans un cas particulier

Une liste de taille $2 \cdot m$ est découpée en deux listes de taille m dont les tris demandent au maximum $C(m)$ comparaisons puis la fusion ajoute $n - 1$ comparaisons.
Pour $p = 0$, $C(2^0) = C(1) = 0 = 0 \cdot 2^0 - 2^0 + 1$, le résultat est vrai pour $p = 0$.
S'il est valide pour $p - 1$ alors, comme $2^p = 2 \cdot 2^{p-1}$,
$C(2^p) = 2 \cdot \left((p-1) \cdot 2^{p-1} - 2^{p-1} + 1\right) + 2^p - 1 = p \cdot 2^p - 2^p - 2^p + 2 + 2^p - 1 = p \cdot 2^p - 2^p + 1$.
On a prouvé le résultat pour tout p par récurrence sur p.

Solution de la question 6 - Simplification

```
let take f =
  let x = first f in
  deque f;
  x;;
```

Solution de la question 7 - Stabilité
Sans cet ajout, la dernière liste, isolée, devrait être fusionnée à celle qui la suit ce qui ferait passer en tête des éléments de la fin de la liste initiale ce qui rendrait impossible la stabilité du tri.

Solution de la question 8 - Écriture
On commence par la création de la file initiale

```
let listeVersFile liste =
  let f = createQueue [] in
  let rec aux l =
    match l with
    |[] -> f
    |t::q -> enque [t] f;
             aux q in
  aux liste;;
```

```
let triFile liste =
  let f = listeVersFile liste in
  while length f > 1 do
    if mength f mod 2 = 1
    then enque [] f;
    for i = 0 to ((length f)/2 - 1) do
      let l1 = take f in
      let l2 = take f in
      enque f (fusion l1 l2)
    done
  done;
  first f;;
```

Solution de la question 9 - Complexité d'une boucle
Le nombre de comparaisons maximal est $\sum_{i=0}^{p-1} n_{i,1} + n_{i,2} - 1 = n - p$ car tous les termes de la liste apparaissent dans une des listes fusionnées et eux seuls. p est le nombre de fusions, c'est la moitié du nombres d'éléments qui vaut r si r est pair ou $r+1$ sinon.

Solution de la question 10 - Complexité dans un cas particulier
Pour une liste initiale de longueur 2^m la file va contenir 2^m puis 2^{m-1} puis 2^{m-2} jusqu'à 2 donc le nombre total de comparaisons est majoré par
$\sum_{k=0}^{m-1} \left(2^m - \frac{2^{m-k}}{2}\right) = m \cdot 2^m - 2^m + 1$, on retrouve la même complexité que dans le cas classique.

Solution de la question 11 - Croissance commençante

```
let rec croissance liste =
  match liste with
  |[] -> []
  |[t] -> [t]
  |t :: s :: q when plusGrand t s -> [t]
  |t :: q -> t :: (croissance q);;
```

Solution de la question 12 - Décomposition initiale

```
let rec debutFin liste =
  match liste with
  |[] -> [], []
  |[t] -> [t], []
  |t :: s :: q when plusGrand t s -> [t], s::q
  |t :: q -> let l1, l2 = debutFin q in
             t::l1, l2;;
```

Solution de la question 13 - Décomposition

```
let rec decompositionC liste =
  match liste with
  |[] -> []
  |_ -> let mono, reste = debutFin liste in
        mono :: (decompositionC reste);;
```

Solution de la question 14 - Écriture
On commence par la création de la file initiale

```
let listeVersFile liste =
  let f = createQueue [] in
  let croiss = decompositionC liste in
  let rec aux l =
    match l with
    |[] -> f
    |t::q -> enque f t;
             aux q in
  aux croiss;;
```

Le reste ne change pas.

```
let triFile liste =
  let f = listeVersFile liste in
  while length f > 1 do
    if length f mod 2 = 1
    then enque f [];
    for i = 0 to ((length f)/2 - 1) do
      let l1 = take f in
      let l2 = take f in
      enque f (fusion l1 l2)
    done
  done;
  first f;;
```

Solution de la question 15 - Un exemple de complexité
Le même raisonnement que dans l'exercice 10, donne une complexité maximale
$\sum_{k=1}^{r}\left(2^m - \frac{2^k}{2}\right) = r\cdot 2^m - 2^r + 1$, on a amélioré le nombre de comparaisons.
Il faut alors ajouter les comparaisons utilisées lors de la décomposition : on compare chaque élément avec le suivant : il y a donc $2^m - 1$ comparaisons. Au total on aboutit à $(r+1)\cdot 2^m - 2^r$ comparaisons.

Solution de la question 16 - Première fonctions

```
let createQueue x =
  {len = 0; debut = Vide; fin = Vide};;

let isEmptyQueue f =
  f.len = 0;;

let length f =
  f.len;;

 let first f =
   match f.debut with
   |Vide -> failwith "La file est vide"
   |Cell c -> c.contenu;;
```

Solution de la question 17 - Adjonction

```
let enque f x =
  let c = Cons {contenu = x; next = Vide} in
  match f.fin with
  |Vide -> f.debut <- c;
           f.fin <- c;
           f.len <- 1
  |Cell c1 -> c1.next <- c;
              f.fin <- c;
              f.len <- f.len + 1;;
```

Solution de la question 18 - Retrait

```
let deque f =
  match f.debut with
  |Vide -> failwith "La file est vide"
  |Cell c when c.next = Vide -> f.len <- 0;
                                f.debut <- Vide;
                                f.fin <- Vide
  |Cell c ->  f.debut <- c.next;
              f.len <- f.len - 1;;
```

Solution de la question 19 - Concaténation

```
let union f1 f2 =
  match f1.fin, f2.debut with
  |_, Vide -> ()  (* Il n'y a rien à faire si f2 est vide ,*)
  |Vide, _ -> f1.len <- f2.len;
              f1.debut <- f2.debut;
              f1.fin <- f2.fin
  |Cell c1, n2 -> c1.next <- n2
                  f1.fin <- f2.fin;;
```