Programmation Fonctionnelle : TP2

Université de Tours

Département informatique de Blois

Tris et étude de complexité

*
* *

Partie 1

Étude de complexité : Le tri à bulles – Partiel 2019

On s'intéresse ici à l'étude, d'un point de vue empirique, de la complexité de différents algorithmes en particulier à la génération de listes mais aussi à une nouvelle méthode de tri d'une liste, le *tri à bulles.*

1. Pour commencer, on va chercher à générer des grandes listes.

   (a) Écrire le code d'une fonction `grandeListeDecroiss n` qui, pour un entier n, crée une liste l décroissante de la forme $l = [n, n-1, ..., 3, 2, 1]$.

   (b) Sur la génération de listes croissante de la forme $l = [1, 2, 3, ..., n-1, n]$.

      i. Écrire une fonction naïve `grandeListeCroiss1` à l'aide de l'opérateur `@` qui permet de crée une liste croissante.

      ii. **Sans utiliser l'opérateur `@`.** Écrire une fonction `aux i n` qui permet de générer une liste $l' = [i, i{+}1, ..., n]$. En déduire une fonction `grandeListeCroiss2` équivalente à `grandeListeCroiss1`.

      iii. **Sans utiliser l'opérateur `@`.** Écrire une fonction `aux acc n'`, récursive terminale qui permet de générer une liste $l' = [1, 2..., n']$. En déduire une fonction `grandeListeCroiss3`, récursive terminale, équivalente à `grandeListeCroiss1`.

   (c) Chronométrer le temps d'exécution en secondes pour les appels suivants :

      - `grandeListeCroiss1 10000;;`
      - `grandeListeCroiss2 10000;;`
      - `grandeListeCroiss3 10000;;`
      - `grandeListeCroiss1 20000;;`
      - `grandeListeCroiss2 20000;;`
      - `grandeListeCroiss3 20000;;`
      - `grandeListeCroiss1 40000;;`
      - `grandeListeCroiss2 40000;;`
      - `grandeListeCroiss3 40000;;`
      - `grandeListeCroiss2 1000000;;`
      - `grandeListeCroiss3 1000000;;`

      Que remarquez-vous ? Que pouvez-vous en conclure ?

   (d) On suppose que `grandeListeCroiss1` est de complexité $O(n^2)$. Plus précisément, on peut estimer que le temps d'exécution[1] T en millisecondes pour une liste de longueur n est donné par

$$T(n) = \frac{1}{26000}n^2 - 0.5n + 2460$$

[1] Le temps est estimé pour les ordinateurs disponibles en salle TP, soit un Intel Core i5-4590S CPU @ 3.00 GHz.

Vérifier que $T(n)$ fournit des résultats concordants avec vos observations. Combien de temps prendrait l'exécution de `grandeListeCroiss1 1000000` ?

Pour `grandeListeCroiss1 100000000` ? Exécuter `grandeListeCroiss3 100000000`.

2. La permutation d'éléments est une opération essentielle pour le tri des listes[2]. Les prochaines questions visent à implémenter cette opération :
   (a) Écrire la spécification et le code d'une fonction `get i l` qui, étant donné une liste l retourne l'élément x_i.
   (b) Écrire la spécification et le code d'une fonction `replace e i l` qui, étant donné une liste l remplace l'élément x_i par l'élément e.
   (c) Écrire la spécification et le code d'une fonction `permute l i j` qui, étant donné une liste l d'éléments, permute les éléments x_i et x_j de la liste.
   Exemple : L'appel `permute [3;4;5;2;8;0] 3 5` retournera la liste `[3;4;5;0;8;2]`.

3. On propose la fonction suivante `grandeListeAlea n` qui génère une liste représentant une permutation de $\mathfrak{S}_n$. Pour se faire, on crée une liste $l = [n, ..., 2, 1]$ puis on mélange au hasard chacun des éléments de l :

```
let grandeListeAlea n =
   let rec melange l =
     let l_melange = permute l 0 (Random.int (List.length l)) in
      match l_melange with
        [] -> []
     | [x] -> [x]
     | h::t -> h::(melange t)
  in
melange (grandeListeDecroiss n);;
```

   (a) Quelle est la complexité de la fonction `grandeListeAlea n` ? Expliquez l'intuition de votre réponse.
   (b) Soit une liste l de taille n. Le *tri stupide* est une méthode de tri qui consiste à mélanger tous les éléments de l au hasard et recommencer tant que celle-ci n'est pas triée.
   Quelle est la probabilité que le tri stupide arrive à ordonner la liste du premier coup ? Quelle est sa complexité ?

4. On considère le code suivant qui réalise un tri à bulles sur une liste d'entrée l :

```
let rec tri_a_bulles l =
  let rec _tri_a_bulles = function
  | h :: h2 :: t when h > h2 -> begin
      match _tri_a_bulles (h :: t) with
    | [] -> h2 :: h :: t
    | t2 -> h2 :: t2
    end
  | h :: h2 :: t -> begin
```

[2]On supposera que les listes débutent à l'indice 1 et se terminent à l'indice n telle que $l = [x_0, ..., x_{n-1}]$.

```
      match _tri_a_bulles (h2 :: t) with
     | [] -> []
     | t2 -> h :: t2
     end
    | _ -> []
in
   match _tri_a_bulles l with
 | [] -> l
 | l -> tri_a_bulles l;;
```

(a) Générer des listes de taille 5000, 10000, 20000 et 25000 à l'aide des méthodes :

- `grandeListeDecroiss`
- `grandeListeCroiss`
- `grandeListeAlea`

On pourra les nommer `l5000C` pour la liste croissante de taille 5000, `l20000D` pour la liste decroissante de taille 20000 par exemple, etc.

(b) Exécuter la fonction `tri_a_bulles` pour chacune des listes précédemment créées et noter le temps que l'algorithme met lors de l'exécution.

À votre avis quelle est la complexité de calcul du tri à bulles lorsque la liste est décroissante (pire des cas) ? Lorsque la liste est croissante (meilleur des cas) ? Lorsque la liste est aléatoire (cas moyen) ? Expliquez votre réponse.

Partie 2

Appropriation du cours

1. On reprend le code suivant qui permet la réalisation du tri par insertion sur des listes d'entiers.

```
let rec insere x l = match l with
  [] -> [x]
| h::t -> if (h > x) then x::l
          else h::(insere x t);;

 let rec tri_insertion l = match l with
  [] -> []
| [x] -> [x]
| h::t -> insere h (tri_insertion t);;
```

Exécuter cet algorithme pour la liste `l1 = [0;(-1);1;(-2);2;(-3);3]` et `l2 = [13;11;7;5;3;2]`.

Suivez l'exécution à l'aide de la commande `#trace` puis commentez le résultat.

2. On reprend le code suivant qui permet la réalisation du tri par fusion sur des listes d'entiers.

```
let rec separe l = match l with
  [] -> ([], [])
| [x] -> ([x], [])
```

```
| a::b::c -> let (l1, l2) = separe c in
                (a::l1, b::l2);;

let rec fusionne l1 l2 = match (l1, l2) with
  ([], l2) -> l2
| (l1, []) -> l1
| (a::b, h::t) -> if (a <= h) then a::(fusionne b l2)
                  else h::(fusionne l1 t);;

let rec tri_fusion l = match l with
  [] -> []
| [x] -> [x]
| h::t -> let (l1,l2) = (separe l) in
              fusionne (tri_fusion l1) (tri_fusion l2);;
```

Exécuter cet algorithme sur les précédents exemples et avec la commande `#trace` puis commentez le résultat.

Problème 1

1. Écrire la spécification et l'algorithme d'une fonction `partitionne l p` qui prend en paramètre une liste d'entiers l et un entier p et retourne pour résultat un couple de listes (l_1, l_2) tel que $l = l_1 \cup l_2$ où :

$$l_1 = \bigcup_{e \in l | e \leq p} e \qquad \text{et} \qquad l_2 = \bigcup_{e \in l | e > p} e$$

   Plus simplement, l_1 est la liste avec tous les éléments e de l plus petits ou égaux à p, et l_2, la liste de tous les éléments e de l strictement plus grands que p.

2. Écrire la fonction `tri_rapide l` du cours qui prend une liste l et l'ordonne selon la relation $\leq$.

Problème 2

1. Écrire la spécification et l'algorithme d'une fonction `extract_min l` qui prend en paramètre d'entrée une liste d'entiers l et retourne pour résultat un couple (min, l') où :

$$min = \min_{x \in l}\{x\} \qquad \text{et} \qquad l' = l \backslash \{min\}$$

   Plus simplement, min est le plus petit élément de l et l' est la liste l sans l'élément min.

   On pourra créer plusieurs fonctions intermédiaires afin de simplifier l'algorithme principal.

2. Écrire la fonction `tri_selection l` du cours qui prend une liste l et l'ordonne selon la relation $\leq$.