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 $l = [n, n-1, ..., 3, 2, 1]$.

   ```
   let rec grandeListeDecroiss n = if (n = 0) then []
                                   else n::(grandeListeDecroiss (n - 1));;
   ```

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

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

      ```
      let rec grandeListeCroiss1 n = if (n = 0) then []
                                     else (grandeListeCroiss1 (n - 1))@[n];;
      ```

      Nous allons voir plus loin que cette solution est loin d'être optimale ! En effet, si `grandeListeCroiss1` st aussi mauvaise, c'est parce qu'elle utilise l'opérateur `@` qui lui est de complexité $O(n)$. Comme il est utilisé n fois lors de l'appel à `grandeListeCroiss1 n` il vient que la fonction est en $O(n^2)$.

      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`.

      ```
      let grandeListeCroiss2 n =
         let rec aux i n' = if (i > n') then []
                            else i::(aux (i + 1) n')
      in aux 1 n;;
      ```

      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`.

```
let grandeListeCroiss3 n =
   let rec aux acc n' = if (n' = 0) then acc
                        else aux (n'::acc) (n' - 1)
in aux [] n;;
```

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

- `grandeListeCroiss1 10000;;` $\Longrightarrow \approx 1s$
- `grandeListeCroiss2 10000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss3 10000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss1 20000;;` $\Longrightarrow \approx 8s$
- `grandeListeCroiss2 20000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss3 20000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss1 40000;;` $\Longrightarrow \approx 40s$
- `grandeListeCroiss2 40000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss3 40000;;` $\Longrightarrow < 1s$
- `grandeListeCroiss2 1000000;;` $\Longrightarrow$ `Stack overflow during evaluation`
- `grandeListeCroiss3 1000000;;` $\Longrightarrow < 1s$

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

On remarque que plus $n \to \infty$, plus l'exécution de `grandeListeCroiss1` est longue. En particulier, doubler la taille de la liste ne double pas le temps d'éxécution ce qui laisse supposer que la complexité de cet algorithme n'est pas linéaire.

Au contraire, `grandeListeCroiss2` et `grandeListeCroiss3` ont des temps d'exécution très courts et similaires. On remarquera néanmoins que l'exécution de `grandeListeCroiss2 1000000;;` produit une explosion de la pile d'exécution.

(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

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

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`.

Le temps d'exécution prévu par le modèle théorique peut varier de quelques secondes en fonction de la machine que vous utilisez, du processeur, du compilateur voire de certains autres paramètres. L'important est de constater une tendance et des ordres de grandeurs assez similaires. On suppose ici que le polynôme $T(n)$ est un bon estimateur du temps d'exécution de `grandeListeCroiss1`.

Ainsi, on peut approximer le temps d'exécution de `grandeListeCroiss1 1000000` par $T(1000000) \approx 37964000ms$ soit 37964 secondes, c'est-à-dire à peu près 10h30.

Pour `grandeListeCroiss1 100000000`, on peut faire le calcul similaire et voir qu'il faudra à peu près 12 ans (et pour une liste de taille 1 millard à peu près 12 siècles....).

Heureusement, on a `grandeListeCroiss3 100000000` qui s'exécute en moins de 1 min. `;)`

2. La permutation d'éléments est une opération essentielle pour le tri à bulles[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.

[1] Le temps est estimé pour les ordinateurs disponibles en salle TP, soit un Intel Core i5-4590S CPU @ 3.00 GHz.
[2] On supposera que les listes débutent à l'indice 0 et se terminent à l'indice $n-1$ telle que $l = [x_0, ..., x_{n-1}]$.

$$get : \begin{cases} \texttt{int}, \texttt{List}\langle'\texttt{a}\rangle & \to \texttt{'a} \\ i, l & \mapsto \begin{cases} l[i] & \text{si } n \geq 1 \text{ et } 0 \leq i < n \\ \text{non défini} & \text{sinon} \end{cases} \end{cases}$$

```
let rec get i l = match (i, l) with
  (i, []) -> failwith "Indice supérieur à la taille de l"
| (i, _) when i < 0 -> failwith "Indice négatif"
| (0, h::t) -> h
| (i, h::t) -> get (i-1) t;;
```

(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.

$$replace : \begin{cases} \texttt{'a}, \texttt{int}, \texttt{List}\langle'\texttt{a}\rangle & \to \texttt{List}\langle'\texttt{a}\rangle \\ e, i, l & \mapsto \begin{cases} l' = [x_0, ..., x_{i-1}, e, x_{i+1}, ..., x_{n-1}] & \text{si } n \geq 1 \text{ et } 0 \leq i < n \\ \text{non défini} & \text{sinon} \end{cases} \end{cases}$$

```
let rec replace e i l = match (i, l) with
  (i, []) -> failwith "Indice supérieur à la taille de l"
| (i, _) when i < 0 -> failwith "Indice négatif"
| (0, h::t) -> e::t
| (i, h::t) -> h::(replace e (i-1) t);;
```

(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]`.

$$permute : \begin{cases} \texttt{List}\langle'\texttt{a}\rangle, \texttt{int}, \texttt{int} & \to \texttt{List}\langle'\texttt{a}\rangle \\ l, i, j & \mapsto \begin{cases} l' = [x_0, ..., x_{i-1}, x_j, x_{i+1}, ..., x_{j-1}, x_i, x_{j+1}, ..., x_{n-1}] & \text{si } n \geq 1 \text{ et } 0 \leq i < n \\ \text{non défini} & \text{sinon} \end{cases} \end{cases}$$

On peut supposer ici que $0 \leq i \leq j < n$ pour se passer des `failwith`.

```
let permute l i j =
  let ei = get i l in (* element a l'indice i *)
  let ej = get j l in (* element a l'indice j *)
replace ej i (replace ei j l);;
```

3. On propose la fonction suivante `grandeListeAlea` qui génère une liste représentant une permutation de $\mathfrak{S}_n$. Pour ce 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` ? Expliquez l'intuition de votre réponse.

On peut décomposer le nombre de calculs effectué par la fonction `grandeListeAlea`.

On remarque que `grandeListeAlea` est en fait égale à la fonction `melange (grandeListeDecroiss n)`, on va donc analyser sa complexité.

- `melange`est appelée pour une liste l , puis elle crée une liste intermédiaire $l_{melange}$ dont l'élément de tête est permuté avec un élément au hasard.
- La création de $l_{melange}$ est égale à la complexité de `List.length + permute`.
  - La fonction qui donne la taille d'une liste est de complexité linéaire $O(|l|)$, où $|l|$ désigne la taille de la liste.
  - La fonction `permute` est égale elle même aux complexité des fonction `get` et `replace`. Comme ces dernières sont appelées toutes de façon séquentielle, la complexité de permute est également en $O(|l|)$.

  Dès lors, la création de $l_{melange}$ s'effectue en temps linéaire en $O(|l|)$.
- Enfin, comme `melange` est appelée sur une liste initiale de taille n, on va donc effectuer le calcul de $l_{melange}$ pour une liste au départ de taille n, puis une liste de taille $n-1$, puis liste de taille $n-2$, etc. Dès lors, on a une complexité de `melange` en $O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$.

(a) 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 ?

Le tri stupide revient à tirer une permutation au hasard parmi l'ensemble de toutes les permutations possibles de $\mathfrak{S}_n$. Ordonner la liste du premier coup revient à tirer la permutation identité dans l'ensemble $\mathfrak{S}_n$. Comme $|\mathfrak{S}_n| = n!$, on a 1 chance sur $n!$ que la liste soit ordonnée du premier coup.

Notons que, dans le pire des cas, la complexité n'est pas bornée et on peut tirer indéfiniment une liste non triée, pour la même raison qu'une pièce de monnaie peut tomber arbitrairement longtemps sur pile ou sur face. Dans ce cas, l'algorithme ne se termine.

Cependant, les probabilités nous indiquent que l'on finira tout de même pas avoir de la chance. On peut ainsi chercher le nombre de tirages moyen T à faire avant que d'obtenir la liste triée. On sait que T suit une loi géométrique de probabilité de réussite $p = \frac{1}{n!}$. Alors la probabilité de trier la liste, c'est-à-dire de tirer la permutation identité, au bout du k essais est telle que:

$$\mathbb{P}(T = k) = (1-p)^k \times p = \left(\frac{n! - 1}{n!}\right)^k \times \frac{1}{n!}$$

Si l'on calcule l'espérance $\mathbb{E}(T)$, c'est-à-dire le nombre moyen de tirages avant d'obtenir la permutation identité, on obtient $\mathbb{E}(T) = \frac{1}{p} = n!$.

Dès lors, on voit qu'en moyenne il nous faudra $n!$ tirages avant de trier notre liste.

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
      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.

```
let l5000D = grandeListeDecroiss 5000;;
let l5000C = grandeListeCroiss3 5000;;
let l5000A = grandeListeAlea 5000;;
let l10000D = grandeListeDecroiss 10000;;
let l10000C = grandeListeCroiss3 10000;;
let l10000A = grandeListeAlea 10000;;
let l20000D = grandeListeDecroiss 20000;;
let l20000C = grandeListeCroiss3 20000;;
let l20000A = grandeListeAlea 20000;;
let l25000D = grandeListeDecroiss 25000;;
let l25000C = grandeListeCroiss3 25000;;
let l25000A = grandeListeAlea 25000;;
```

(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.

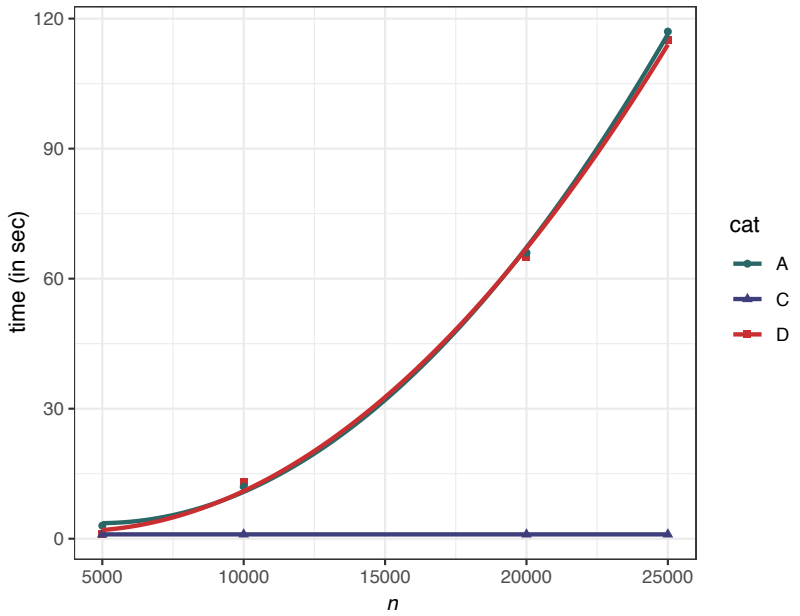
À votre avis quelle est la complexité de calcul du tri à bulles lorsque la liste est décroissante (pire des cas) ? Lorsque la liste est croissante (meilleur des cas) ? Lorsque la liste est aléatoire (cas moyen) ? Expliquez votre réponse.

On cherche donc à évaluer la complexité du nombre de comparaison effectuer lors du tri à bulles.

On peut tracer le graphe suivant qui donne le temps en secondes en fonction de la taille d'une liste. À noter que les temps peuvent différer en fonction de votre machine. L'important ici est la tendance des courbes représentatives.

Quelques éléments de preuve sur la complexité du tri à bulles qui viennent expliquer les observations faites. On considère une liste l de taille $|l| = n$. On va tâcher de dénombrer le nombre de comparaisons d'entiers à effectuer :

- La tendance est linéaire en $O(n)$ lorsque la liste est croissante (catégorie C), il suffit de parcourir la liste une seule fois pour constater qu'elle est triée.
- La tendance est quadratique en $O(n^2)$ lorsque la liste est décroissante (catégorie D). En effet, le tri à bulles consiste à parcourir la liste et à permuter chaque élément pour faire remonter progressivement l'élément maximal. On parcourt alors les n éléments au départ pour faire remonter le plus grand en fin de liste ; on sait que le dernier est trié, on fait alors remonter le deuxième élément maximal à la position $n-1$, etc.
  On a alors $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ éléments à parcourir au total. Il vient que la complexité est en $O(n^2)$ ce que l'on observe sur la courbe rouge
- La preuve lorsque la liste est aléatoire (catégorie A) est similaire. Il s'agit précisément de compter le nombre de paires en inversion de la liste, c'est-à-dire le nombre de couples (x_i, x_j) tel que $i < j$ et tel que $x_i > x_j$. On peut montrer qu'il y en a en moyenne $\frac{n(n-1)}{4}$ ce qui fournit une complexité également en $O(n^2)$.



Vous pouvez visiter la page Wikipédia : `https://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles` pour constater tous les éléments que l'on vient de démontrer ensemble. `:)`

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.

$$partitionne : \begin{cases} \texttt{List}\langle\texttt{int}\rangle, \texttt{int} & \to \texttt{List}\langle\texttt{int}\rangle * \texttt{List}\langle\texttt{int}\rangle \\ l, p & \mapsto (l_1, l_2) \end{cases}$$

avec l_1 et l_2 définies ci-dessus.

```
let rec partitionne l p = match l with
  [] -> ([], [])
| h::t -> let (l1,l2) = (partitionne t p) in
               if(h <= p) then (h::l1, l2) else (l1, h::l2);;
```

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

```
let rec tri_rapide l = match l with
  [] -> []
| h::t -> let (l1, l2) = partitionne t h in
              (tri_rapide l1)@[h]@(tri_rapide l2);;
```

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.

$$extract\_min : \begin{cases} \texttt{List}\langle\texttt{int}\rangle & \to \texttt{int} * \texttt{List}\langle\texttt{int}\rangle \\ l & \mapsto (min, l') \end{cases}$$

avec min et l' définis ci-dessus.

```
(* Calcul du minimum d'une liste min=\underset{x\in l}{\min}\{x\}
   List<int> -> int *)
let rec min_liste l = match l with
  [] -> failwith "Liste vide"
| [x] -> x
| h::t -> min h (min_liste t);;

(* Liste l sans l'élément n l'=l\backslash\{n\}
   List<int> -> List<int> *)

let rec retire l n =
  match l with
  [] -> []
```

```
| h::t -> if(h = n) then t else h::(retire t n);;


let extract_min l =
  let m = min_liste l in (m, retire l m);;
```

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

```
let rec tri_selection l = match l with
  [] -> []
| h::t -> let (m, _l) = extract_min l in
               m::(tri_selection _l);;
```