# Option Informatique

**Devoir Surveillé *n*°01** – Durée : 2h. Le 27/03/2024.

Tout matériel électronique est interdit. Tout document est interdit mais l'aide mémoire Caml est donné en annexe. Longueur : 3 pages + 1 page d'annexe.

**Important :** tout code doit être précédé d'une explication de son fonctionnement, **brève** et **claire**.

## Déjà vus en cours (normalement)...

Exercice 1 : **suites récurrentes croisées**

1. On définit deux suites par $\begin{cases} u_0 = 1 \\ v_0 = 1 \end{cases}$ et $\forall n \in \mathbb{N}$, $\begin{cases} u_{n+1} = 2u_n + v_n \\ v_{n+1} = u_n + 2v_n \end{cases}$.
   Écrire deux fonctions `u : int → int` et `v : int → int` tq pour tout entier $n$ on ait `u n`$=u_n$ et `v n`$=v_n$.

Exercice 2 : **valuation 10-adique**

2. Écrire une fonction `v10 : int → int` qui prend comme argument un entier $n$ et renvoie comme résultat la valuation 10-adique de $n$, c'est-à-dire le plus grand entier $p$ tel que $10^p \mid n$.

   *Vous pouvez procéder par itération, par récursivité enveloppée ou par récursivité terminale, ce qui vous arrange le plus (mais ne donnez qu'une seule solution !).*

## Déjà vu en TD (normalement)...

Exercice 3 : **tri par insertion d'une liste**

Pour trier une liste $\ell$ par insertion, on procède comme suit :

- Si $\ell$ est vide, alors elle est triée.
- Sinon, $\ell$ est de la forme $h :: t$. On trie récursivement $t$ et on insère $h$ à sa place dans la liste triée.

3. Écrire en OCaml une fonction `insere : 'a →'a list→'a list` telle que, si `l` est une liste triée alors `insere x l` retourne une liste **triée** ayant les mêmes éléments de `l`, plus l'élément `x` (à sa place).

4. En utilisant la fonction précédente, implémenter l'algorithme du tri par insertion sur les listes en OCaml : écrire une fonction `tri1 : 'a list→'a list`.

5. Montrer que dans le pire des cas, le nombre de comparaisons effectuées est de l'ordre de $n^2$.

Une variante du tri par insertion est la suivante : pour trier une liste $\ell$ :

- On considère deux listes : la liste $\ell_1$ des éléments qui sont encore à tirer, initialement $\ell$, et la liste $\ell_2$ des élément déjà triés, initialement [].
- On prend la tête de $\ell_1$ et on l'insère à sa place dans $\ell_2$ ;
- On recommence jusqu'à que ce $\ell_1$ soit vide.

Bien sûr, dans la description précédente, il n'y a pas de réelle modification des deux listes, simplement des appels récursifs sur des listes modifiées.

6. Implémenter cette version de l'algorithme en OCaml. On utilisera la fonction `insere` vue dans la question 3 (y compris si vous n'avez pas répondu à cette question), voire d'autres fonctions auxilliaires pour obtenir une fonction `tri2 : 'a list→'a list`.

## Problème : tri par séquences croissantes

On présente dans cet exercice un algorithme de tri adapté aux listes (inspiré du *Timsort*, le tri de Python) et que l'on appelera *tri par séquences croissantes*.

**Dans tout l'exercice, l'unité de mesure des complexité sera la comparaison $<$ (ou $>$ ou $\leqslant$ ou $\geqslant$).** Ainsi, lorsqu'on demandera d'estimer une complexité, c'est le nombre de comparaisons qu'on demandera de compter.

Le principe du *tri par séquences croissantes* sur les listes est le suivant : on découpe la liste à tirer en liste de listes triées, en découpant la liste initiale à chaque observation d'une décroissance. Par exemple, le découpage de la liste $[1;2;3;4;5;2;5;7;9;3;6;7;0;1]$ doit donner la liste de listes $[[1;2;3;4;5];[2;5;7;9];[3;6;7];[0;1]]$. Ceci doit être fait en un seul parcours de la liste afin de garantir que cette étape soit de complexité linéaire.

Dans un second temps, on fusionne les listes triées obtenues.

7. Écrire une fonction `decouper : 'a list → 'a list list` réalisant le découpage décrit dans l'introduction. Attention, la complexité de cette fonction (au sens défini dans l'introduction) devra être linéaire (mais il n'est pas demandé de le montrer).

8. Pour fusionner les listes obtenues par découpage, on aura besoin tout d'abord d'une fonction

   `fusionner : 'a list → 'a list → 'a list`

   telle que, si `l1` et `l2` sont deux listes triées, alors `fusionner l1 l2` est une liste triée dont les éléments sont exactement les éléments de `l1` et `l2`. Écrire une telle fonction. Attention, la complexité de cette fonction (au sens défini dans l'introduction) devra également être linéaire (mais il n'est pas demandé de le montrer).

Une première idée pour fusionner toutes les listes obtenues après découpage est la suivante : en notant $m$ le nombre de listes obtenues, on réalise $m-1$ appels de la fonction `fusionner` ; on fusionne tout d'abord la première et la seconde liste, puis le résultat obtenu avec la troisième liste, puis le résultat obtenu avec la quatrième, etc. Mais si $m$ est grand, une stratégie de type "diviser pour régner" sera beaucoup plus efficace : on découpe la liste de listes en deux (disons les $\lfloor \frac{m}{2} \rfloor$ premières et les $\lceil \frac{m}{2} \rceil$ suivantes, ou l'inverse), on réalise récursivement la fusion des $\lfloor \frac{m}{2} \rfloor$ premières et la fusion des $\lceil \frac{m}{2} \rceil$ suivantes, puis on réalise la fusion des deux listes obtenues à l'aide de la fonction `fusionner`.

9. Montrer que, en utilisant la stratégie de type "diviser pour régner" décrite ci-dessus pour fusionner $m$ listes, on a toujours besoin de $m-1$ appels à la fonction `fusionner`.

L'intérêt de cette stratégie est d'obtenir que les fusions se fassent en général sur des listes de tailles comparables.

10. Écrire une fonction `scinder : 'b list -> 'b list * 'b list` permettant de scinder une liste en deux listes de longueurs égales à une unité près. L'idée est d'utiliser ensuite cette fonction `scinder` dans le cas où le type `'b` est le type `'a list`. Par exemple, `scinder [[1; 2; 3; 4; 5]; [2; 5; 7; 9]; [3; 6; 7]; [0; 1]]`
    pourra retourner `([[1; 2; 3; 4; 5]; [3; 6; 7]], [[2; 5; 7; 9]; [0; 1]])`, ou encore, suivant ce qui vous arrange : `([[1; 2; 3; 4; 5]; [2; 5; 7; 9]], [[3; 6; 7]; [0; 1]])` (propositions non exhaustives).

11. Écrire une fonction `megafusionner : 'a list list → 'a list` telle que, si `l` est une liste de listes triées, alors `megafusionner l` fusionne toutes les listes de `l` en utilisant l'algorithme de type "diviser pour régner" décrit plus haut.

12. Finalement, en utilisant les questions précédentes, écrire une fonction `tri_sc : 'a list → 'a list` permettant de trier une liste à l'aide de l'algorithme de tri par séquences croissantes.

On s'intéresse maintenant à la complexité (au sens défini dans l'introduction, je ne l'écrirai plus) de `tri_sc`.

13. Donner une estimation de la complexité de `tri_sc l` lorsque `l` est une liste de longueur $n$ **déjà triée**.

14. Supposons avoir `l = [1;2;3;4;5;2;5;7;9;3;6;7;0;1]`. Expliciter les listes qui seront fusionnées deux à deux au cours de l'exécution de l'algorithme. On suggère une représentation arborescente.

15. Soit `l` une liste de longueur $n$ telle que `l` est formée de $m$ séquences croissantes, où évidemment on a $1 \leqslant m \leqslant n$ (autrement dit, `l` est une liste de longueur $n$ et `decouper l` est une liste de longueur $m$).

    Montrer que la complexité de `tri_sc l` est un $O(n + n\lg(m))$.

# Aide-mémoire OCaml

## Déclarations et instructions

| | |
|---|---|
| commentaires | (* ... *) |
| définition d'une valeur | let v = expression |
| récursive | let rec v = ... |
| locale | let v = ... in expression |
| définitions simultanées | let v = ... and w = ... |
| successives | let v = ... in let w = ... |
| référence (modifiable) | let v = ref expression |
| valeur d'une référence | !v |
| modifier une référence | v := ... |
| fonction sans argument | let f () = ... |
| à un argument | let f x = ... |
| à n arguments | let f $x_1$ ... $x_n$ = ... |
| expression conditionnelle | if condition then $expression_1$ else $expression_2$ |
| filtrage simple | match valeur with<br>\| $motif_1$ -> $expression_1$<br>...<br>\| $motif_k$ -> $expression_k$<br>\| _ -> expression-par-défaut |
| filtrage avec garde | match valeur with<br>\| $motif_1$ when $condition_1$ -> ...<br>... |
| ne rien faire | () |
| suite d'instructions | begin ... end |
| boucle croissante | for i = début to fin do ... done |
| conditionnelle | while condition do ... done |
| exception personnalisée | failwith "message d'erreur" |
| fonction anonyme | function x -> ... |
| à plusieurs arguments | fun x y -> ... |

## Expressions booléennes

| | |
|---|---|
| vrai, faux | true, false (en minuscules) |
| et, ou, non | &&, \|\|, not |
| comparaison | <, <=, =, <>, >=, > |
| booléen ⟼ chaîne | string_of_bool |
| chaîne ⟼ booléen | bool_of_string |

## Expressions entières

| | |
|---|---|
| opérations arithmétiques | +, -, *, /, mod |
| valeur absolue | abs |
| minimum, maximum | min a b, max a b |
| entier ⟼ chaîne | string_of_int |
| chaîne ⟼ entier | int_of_string |
| entier aléatoire de {0,...,n-1} | Random.int n |
| afficher un entier | print_int |

## Expressions flottantes

| | |
|---|---|
| opérations arithmétiques | +., -., *., /. |
| puissance | ** |
| minimum, maximum | min a b, max a b |
| fonctions mathématiques | abs_float, exp, log, sqrt, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, atan2 |
| flottant ⟼ entier | int_of_float |
| entier ⟼ flottant | float_of_int |
| flottant ⟼ chaîne | string_of_float |
| chaîne ⟼ flottant | float_of_string |
| flottant aléatoire entre 0 et a | Random.float a |
| afficher un flottant | print_float |

## Listes

| | |
|---|---|
| liste | [x ; y ; z ; ...] |
| liste vide | [ ] |
| tête et queue | List.hd, List.tl, h : :t |
| longueur d'une liste | List.length |
| concaténation | @ |
| image miroir | List.rev |
| appliquer une fonction | List.map fonction liste |
| appliquer un traitement | List.iter traitement liste |
| itérer une opération | List.fold_left f e liste |
| test d'appartenance | List.mem élément liste |
| test de présence | List.exists prédicat liste,<br>List.for_all prédicat liste |
| indice d'un élément | List.nth liste élément |
| tri | Sort.list ordre liste |

## Tableaux

| | |
|---|---|
| tableau | [\|x ; y ; z ; ...\|] |
| tableau vide | [\| \|] |
| i-ème élement | v.(i) |
| modification | v.(i) <- qqch |
| longueur d'un tableau | Array.length |
| création d'un tableau | Array.make longueur valeur |
| d'une matrice | Array.make_matrix n p valeur |
| extraction | Array.sub tableau début longueur |
| concaténation | Array.append $tableau_1$ $tableau_2$ |
| copie | Array.copy tableau |
| appliquer une fonction | Array.map fonction tableau |
| appliquer un traitement | Array.iter traitement tableau |
| tableau ⟼ liste | Array.to_list |
| liste ⟼ tableau | Array.of_list |

## Chaînes de caractères

| | |
|---|---|
| caractère | 'x' |
| chaîne de caractères | "xyz..." |
| $i^{ième}$ caractère | chaîne.[i] |
| modification | chaîne.[i] <- qqch |
| longueur d'une chaîne | String.length |
| création d'une chaîne | String.make longueur caractère |
| caractère ⟼ chaîne | String.make 1 caractère |
| extraction | String.sub chaîne début longueur |
| concaténation | $chaîne_1$^$chaîne_2$, String.concat |
| afficher un caractère | print_char |
| une chaîne | print_string |

## Commande de l'interpréteur

| | |
|---|---|
| tracer une fonction | #trace fonction |
| ne plus tracer | #untrace fonction |
| charger une bibliothèque | open nom, #load "nom" |
| charger un fichier source | #use "nom" |

## Entrées-sorties

| | |
|---|---|
| impression formatée | Printf.fprintf |

### Sur le terminal

| | |
|---|---|
| impression de valeurs | print_int, print_float, print_char, print_string |
| changer de ligne | print_newline () |
| lecture de valeurs | read_int, read_float, read_line |

### Dans un fichier

| | |
|---|---|
| ouverture en lecture | let canal = open_in "nom" |
| en écriture | let canal = open_out "nom" |
| lecture | input_char, input_line, input_byte, input_value |
| écriture | output_char, output_string, output_byte, output_value, flush |
| fermeture | close_in, closeout |