Algorithmes standard

TD
*

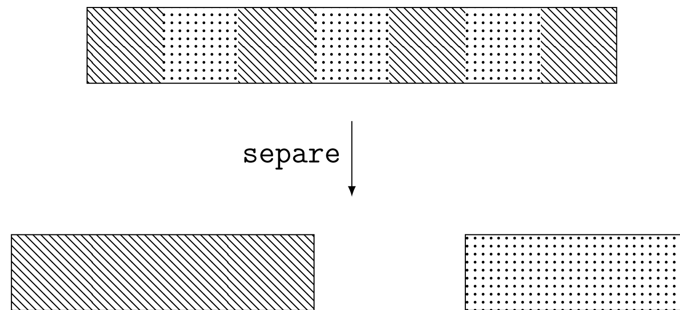
Ceci est un petit document d'exercices, de niveaux variés : sauf exercices indiqués (⋆), ils doivent être maîtrisés et compris. Un corrigé est disponible sur `pierrebeaur.fr`.

1 Tris

Exercice 1 : tri fusion sur les listes

Le tri le plus standard sur les listes est le tri fusion, qui nécessite toutefois une astuce.

1. Écrire une fonction `separe : 'a list -> 'a list * 'a list` qui partitionne une liste en deux de tailles quasi égales. L'astuce (à retenir) est de ne pas découper la liste en blocs contigus, mais de répartir un élément sur 2 dans les deux listes :



   On prendra soin de traiter le cas des listes de longueur impaire.
2. Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` qui fusionne deux listes triées.
3. Écrire une fonction `tri_fusion : 'a list -> 'a list` procédant au tri fusion d'une liste.

Exercice 2 : tri par sélection d'un tableau

Écrire une fonction `tri_selection : 'a array -> unit` qui trie **en place** un tableau.

Exercice 3

1. Écrire une fonction `est_trie : 'a list -> bool` qui teste si une liste est triée.
2. Écrire une fonction `est_trie_tab : 'a array -> bool` pour un tableau.

2 Fonctions d'agrégat

Exercice 4

1. Écrire une fonction `maxi : 'a list -> 'a` renvoyant le max d'une liste.
2. Écrire une fonction `maxi_t : 'a array -> 'a` pour un tableau.

On exige des complexités linéaires.

Exercice 5

1. Écrire une fonction `som : int list -> int` qui renvoie la somme des éléments d'une liste.
2. Même chose pour un tableau.

Exercice 6

1. Écrire une fonction `moyenne : float list -> float` renvoyant la moyenne des éléments d'une liste.
2. Même chose pour un tableau.

On fera attention, lors des tests, aux types (entre `int` et `float`). La fonction `float_of_int : int -> float` pourra vous aider.

3 Recherche

Exercice 7 : recherche linéaire

1. Écrire une fonction `mem : 'a list -> 'a -> bool` qui teste l'appartenance d'un élément à une liste.
2. Même chose pour un tableau.

Exercice 8 : recherche dichotomique

On rappelle le principe de la recherche dichotomique : soit t un tableau trié et x un élément.

- si le tableau est vide, alors l'élément n'est pas dedans ;
- sinon, on regarde au milieu.
  - si au milieu on trouve x, youpi !
  - si le milieu est trop grand, on regarde à gauche ;
  - sinon, on regarde à droite.

On rappelle que, dans cette explication, on ne manipule pas **littéralement** la moitié gauche ou la moitié droite (en faisant cela, on aurait une complexité catastrophique) : on manipule plutôt deux variables désignant les bords gauche et droit actuels.

1. Écrire **en Python** une fonction `dicho(tab,x)` qui procède à la recherche dichotomique. La recherche dichotomique est un code assez piégeux, qui a vite tendance à boucler à l'infini ou à mal traiter les cas limites. On testera la recherche dichotomique dans les cas suivants :
   - l'élément cherché est tout à gauche ;
   - il est tout à droite ;
   - le tableau a 0, 1 ou 2 éléments.
2. Écrire une version OCaml.
3. Expliquer pourquoi la recherche dichotomique n'a aucun sens dans le cas d'une liste.

4 En vrac

Exercice 9

Une permutation est une structure en bijection avec un intervalle de la forme $[\![0, n-1]\!]$.

1. Écrire une fonction `perm : int array -> bool` qui vérifie qu'un tableau est une permutation.
2. Même chose avec une liste.

Exercice 10

Écrire une fonction `diviseurs : int -> int list array` qui prend un entier n et renvoie un tableau `t` indexé par $[\![0, n]\!]$ de telle sorte que `t.(i)` contienne les diviseurs de `i` (exception pour 0 : on laissera `t.(0) = []`).

Exercice 11

On a une liste ℓ contenant des entiers positifs, par exemple $\ell = [1; 0; 4; 0; 5; 2]$. On cherche à créer un tableau `t` tel que `t.(i)` renvoie « l'indice » de la première apparition de `i` dans ℓ. Si l'élément n'apparaît pas, on met -1. Par exemple, avec la liste précédente, on obtiendrait `[|1;0;5;-1;2;4|]`.
Écrire une fonction `numero : int list -> int array` renvoyant un tel tableau. Déterminer la complexité d'une telle fonction selon les paramètres pertinents du problème.

Exercice 12

La *paperfolding sequence* est une suite de listes binaires dont le n-ème terme se construit de la façon suivante :

— prendre une feuille blanche à deux mains, un bord dans chaque main;

— plier cette feuille en deux, tel que le pli s'éloigne de vous;

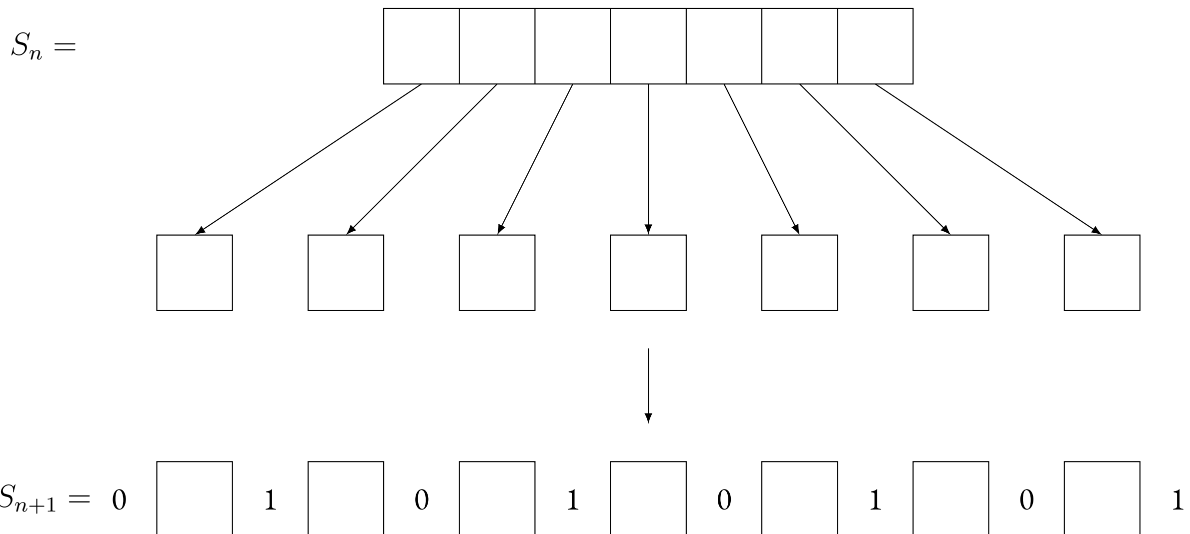
— récupérer les deux bords dans la main gauche, et récupérer le « bord du fond » avec la main droite;

— recommencer ce processus jusqu'à l'avoir fait n fois;

— sans jamais lâcher le bord gauche, déplier la feuille.

On observe au fond de la feuille des pics et des vallées. Une vallée est un 0; un pic est un 1. De gauche à droite, on lit le n-ème terme de la suite-de-pliage-de-papier (SDPP). On note S_n cette liste.

1. Vérifier, avec une feuille de papier, que $S_1 = [0]$; $S_2 = [0; 0; 1]$; $S_3 = [0; 0; 1; 0; 0; 1; 1]$.
2. Soit S_n un terme de la suite, on cherche à construire S_{n+1}. Observer le pliage d'une feuille, et en déduire que le schéma suivant permet d'obtenir S_{n+1}.



3. Écrire une fonction `intercale : int list -> int list` qui procède à l'intercalage final; conclure en écrivant une fonction `sdpp : int -> int list` renvoyant le n-ème terme de la SDPP.

Une autre construction de la SDPP suit une construction bien différente. Pour une liste ℓ à valeurs dans $\{0, 1\}$, on appelle (dans cet exo) le contraire de ℓ la liste où les 0 sont devenus des 1, les 1 des 0 : par exemple, le contraire de $[0; 1; 0; 1; 1]$ est $[1; 0; 1; 0; 0]$. On pourra remarquer que $1 - 0 = 1$ et $1 - 1 = 0$.

4. Écrire une fonction `contre : int list -> int list` renvoyant le contraire d'une liste opérant en temps linéaire.
5. Admettre que $S_{n+1} = S_n@[0]@(\text{miroir du contraire de } S_n)$.
6. Écrire une fonction `sdpp2 : int -> int list` renvoyant la SDPP selon ce dernier principe récursif.
7. (Bonus) Ne pas admettre la question 5 et la justifier.

Exercice 13

L'exercice suivant porte sur le taquin. Si vous ne connaissez, il vous est recommandé (très sérieusement) de jouer à quelques parties sur `taquin.net`.
On s'intéresse au jeu du taquin, selon une modélisation simple : on part par exemple de la grille de gauche, et on cherche à obtenir la grille de droite.

11	2	5	14
0	12	3	13
6	8	4	9
7	15	10	1

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

La grille est de taille $N \times N$, et les pièces numérotées de 1 à $N^2 - 1$. La « pièce » 0 est en fait un trou, sur lequel on peut faire glisser une pièce. On considère donc 4 types de coups possibles : Nord, Sud, Est et Ouest, suivant la direction dans laquelle le trou est déplacé lors d'un coup. À une grille on associe une permutation σ_B lue « en boustrophédon » : cela signifie qu'on lit la grille ligne par ligne en commençant de gauche à droite, puis de droite à gauche, puis de gauche à droite , ...
Par exemple, dans l'exemple du dessus, $\sigma_B = [11, 2, 5, 14, 13, 3, 12, 0, 6, 8, 4, 9, 1, 10, 15, 7]$.
Toute grille de taquin n'est pas résoluble : il suffit de considérer la grille suivante.

0	1
3	2

On note g_0 la grille finale à atteindre. On vous donne le théorème suivant :

> Une grille g est résoluble ssi la signature de la σ_B associée à g est égale à la signature de la σ_B associée à g_0.

Écrire une fonction `resoluble : int array array -> bool` qui détermine si une grille de taquin est résoluble.

5 Arbres binaires

On implémente les arbres binaires selon le type suivant :

```
type 'a arbre = V | N of 'a * 'a arbre * 'a arbre ;;
```

Cette définition est classique, mais tout le monde n'a pas la même, faites attention. Il est par exemple classique de placer l'étiquette **au milieu**, entre les deux fils :

```
type 'a arbre2 = V2 | N2 of 'a arbre2 * 'a * 'a arbre2 ;;
```

Faites attention en lisant un énoncé.

Exercice 14

Écrire une fonction `hauteur` et une fonction `taille` donnant respectivement la hauteur et la taille (= nombre de nœuds). On demande une complexité linéaire en la taille de l'arbre.

Exercice 15

Écrire une fonction `mini : 'a arbre -> 'a` renvoyant le minimum d'un arbre non vide.

Exercice 16

Écrire une fonction `est_abr : 'a arbre -> bool` qui teste si un arbre binaire est un ABR.

Exercice 17

Écrire une fonction `est_dedans : 'a arbre -> 'a -> bool` qui teste si un élément est dans un arbre.

Exercice 18

Écrire une fonction `chemin : 'a arbre -> 'a -> 'a list` qui prend un arbre en entrée et un élément x, et renvoie la liste des nœuds, à partir de la racine, conduisant à l'élément en question. Si l'élément n'est pas dans l'arbre, on plantera. Si l'élément apparaît plusieurs fois, on renvoie arbitrairement un chemin.

Déterminer la complexité de votre fonction.

Exercice 19 : parcours d'un arbre

Pour un arbre binaire A, un parcours est une façon d'en donner la liste des nœuds. On distingue trois principaux parcours : préfixe, infixe et postfixe.
Dans un parcours préfixe, on « affiche » d'abord un nœud, puis ensuite ceux du fils gauche, et enfin du fils droit.
On considère le code suivant :

```
let rec parc_pref a =
    match a with
    | V -> []
    | N(x,g,d) -> x::(parc_pref g)@(parc_pref d);;
```

1. Ce code permet d'obtenir le parcours préfixe d'un arbre binaire. Le tester sur un arbre à créer.
2. On aimerait que le parcours en linéaire en la taille de l'arbre : ce n'est pas le cas de cette fonction. En déterminer sa complexité.
3. Écrire une fonction `parc_pref_aux : 'a arbre -> 'a list -> 'a list` qui rajoute les éléments du parcours préfixe à la gauche d'une liste donnée en argument.
4. En déduire une fonction linéaire renvoyant le parcours préfixe.

Le parcours postfixe consiste à d'abord afficher le fils gauche, puis le fils droit, puis le père.

5. Écrire une fonction `parc_post : 'a arbre -> 'a list` renvoyant le parcours postfixe d'un arbre. Assurez-vous que la complexité obtenue est bien linéaire. *Indication : un miroir bien placé.*

Le parcours infixe consiste à d'abord parcours le fils gauche ; puis le père ; puis le fils droit.

6. Écrire une fonction `parc_inf : 'a arbre -> 'a list` renvoyant le parcours infixe d'un arbre.
7. Que dire sur le parcours infixe d'un ABR ?

6 Arbres binaires de recherche

On implémente les ABR par le même type que les arbre binaires. Pour tous les exercices suivants, identifier les complexités en $O(h)$ ou $O(n)$.

Exercice 20

Écrire une fonction `in_abr : 'a arbre -> 'a -> bool` qui détermine si un élément est dans un ABR.

Exercice 21

Écrire une fonction `mini_a : 'a arbre -> 'a` qui renvoie l'élément minimal d'un ABR.

Exercice 22

Écrire une fonction `insere : 'a arbre -> 'a -> 'a arbre` qui renvoie un nouvel ABR avec un nouvel élément inséré au niveau des feuilles.

Exercice 23

Écrire une fonction `fusion : 'a arbre -> 'a arbre -> 'a arbre` qui fusionne deux ABR.

Exercice 24 : un peu astucieux

Écrire une fonction `supprime : 'a arbre -> 'a -> 'a arbre` qui supprime un élément d'un arbre. Si l'élément n'est pas dans l'arbre, on ne plantera pas.
Indication : il faut réfléchir à une stratégie.

7 Pile et file

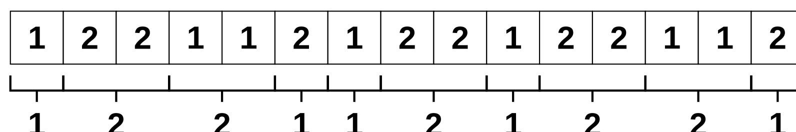
On utilisera les piles et files implémentées nativement par `Stack` et `Queue`. Vos fonctions peuvent être destructrices, sauf précision contraire.

Exercice 25

1. Écrire une fonction `parites : int queue -> (int queue) * (int queue)` qui prend une file et la sépare en deux files selon la parité des éléments, en préservant ce qui était l'ordre original de la file.
2. Même chose avec une pile.

Exercice 26 : suite de Kolakoski

La suite de Kolakoski $(k_i)_{i \in \mathbb{N}}$ est une suite à valeurs dans $\{1, 2\}$ qui vérifie une forme d'autodescription : si on découpe la suite en blocs de lettres identiques, la longueur du i-ème bloc est k_i.



1. La suite de Kolakoski est entièrement définie par ce procédé autodescriptif et par le fait de commencer par un 1 ; **à la main**, déterminer les 10 premiers termes de la suite.
2. En utilisant des files, écrire une fonction `kolakoski : int -> int list` renvoyant le résultat obtenu après n itérations du procédé de construction.