Pierre Béaur & Hugo Buscemi – CPGE Louis-le-Grand

Vacances à dos de chameau

Préambule

Les exercices suivants vous serviront à ne pas perdre la main pendant les vacances. L'objectif est de « picorer » dans cette feuille : nous vous conseillons de répartir votre travail tout le long de l'été, plutôt que de la faire dans un court laps de temps.

La difficulté n'est pas croissante, et les thèmes sont variés : n'hésitez pas à revenir sur un exercice précédent sur lequel vous avez buté, ou à travailler spécifiquement les sujets qui vous ont posé problème cette année.

Les conseils habituels perdurent : faire attention aux types manipulés, nommer vos variables judicieusement pour vous aider, ne pas hésiter à créer une/des fonction(s) annexe(s) afin de séparer le travail, lever une erreur à l'aide de `failwith` dans les cas non traités, faire attention au parenthésage, maintenir des `matchs` exhaustifs.

Et surtout : tester, tester, tester !

Exercice 0 : maximum

Écrire une fonction `maxl : 'a list -> 'a` qui renvoie le maximum d'une liste (peu importe le type).
Écrire une fonction `maxa : 'a array -> 'a` qui renvoie le maximum d'un tableau (peu importe le type).

Exercice 1 : évaluation d'un polynôme en un point

On représente un polynôme comme la liste de ses coefficients : par exemple, $P = 1 + 2X + 5X^2$ est représenté par la liste `[1.;2.;5.]` (de flottants !). Écrire une fonction `horner : float list -> float -> float` qui renvoie l'évaluation d'un polynôme en un point selon l'algorithme de Horner.

Exercice 2 : arbres binaires

On représente les arbres binaires par `type arbre_bin = Vide | Noeud of int * arbre_bin * arbre_bin`.
On rappelle que la taille de l'arbre vide est 0 *et que sa hauteur est* -1.
Écrire une fonction `taille_arbre : arbre_bin -> int` qui renvoie la taille d'un arbre binaire.
Écrire une fonction `hauteur_arbre : arbre_bin -> int` qui renvoie la hauteur d'un arbre binaire.

Exercice 3 : PGCD

Écrire une fonction `pgcd : int -> int -> int` renvoyant le PGCD de deux entiers selon l'algorithme d'Euclide.
Écrire une fonction `pgcdl : int list -> int` qui renvoie le PGCD d'une liste.

Exercice 4 : rotation d'une matrice

On considère l'opération de rotation d'une matrice carrée (un quart de tour dans le sens trigonométrique) :

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \to \begin{pmatrix} 3 & 7 & 11 & 15 \\ 2 & 6 & 10 & 14 \\ 1 & 5 & 9 & 13 \\ 0 & 4 & 8 & 12 \end{pmatrix}$$

Écrire une fonction `rotation : 'a array array -> unit()` qui procède à une rotation d'une matrice. On demande à ce que la fonction soit en place.

Exercice 5 : ensemble des parties

On s'intéresse à un ensemble $E = [\![0, N-1]\!]$; on cherche à manipuler les parties de E. Une partie de E est représentée comme la liste triée de ses éléments (par exemple, $\{0, 2, 5\}$ est représentée par `[0;2;5]`. Plus précisément, $\mathcal{P}(E)$ est en bijection avec $[\![0, 2^N - 1]\!]$: on cherche à expliciter cette bijection.
La construction classique est la suivante :

$$\varphi : \begin{pmatrix} \mathcal{P}(E) & \to & [\![0, 2^N - 1]\!] \\ F & \mapsto & \displaystyle\sum_{f \in F} 2^f \end{pmatrix}$$

Écrire une fonction `phi : int list -> int` qui renvoie l'image d'une partie par φ, de complexité $O(N)$.
Écrire une fonction `antiphi : int -> int list` qui est la bijection réciproque de φ, de complexité $O(N)$.

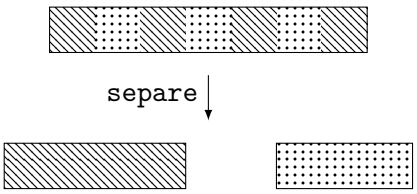
Exercice 6 : filtre

Écrire une fonction `filtre_liste : ('a -> bool) -> 'a list -> 'a list` telle que `filtre_liste f li` soit la liste des éléments `e` de `li` tels que `f e` vaille `true`.
La fonction imite le comportement de `List.filter` *mais ne doit bien sûr* ***pas*** *l'utiliser.*

Exercice 7 : tri fusion sur les listes

Le tri classique sur les listes est le tri fusion : on en présente la version la plus standard.

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 8 : renumérotation

On considère une liste d'entiers positifs ℓ, et on cherche à la renuméroter « le plus bas possible » : par exemple, considérons la liste `[4;5;4;0;4;4;5]`. Une renumérotation consisterait à considérer $4 \mapsto 0, 5 \mapsto 1, 0 \mapsto 2$: on obtiendrait alors la liste `[0;1;0;2;0;0;1]`.
Écrire une fonction `renumerote : int list -> int list` qui procède à une telle renumérotation.
Indication : il existe une réponse raisonnable opérant en $O(\max(\ell) + |\ell|)$.

Exercice 9 : liste des feuilles

Avec les arbres binaires représentés par `type arbre_bin = Vide | Noeud of int * arbre_bin * arbre_bin`, écrire une fonction `feuilles : arbre_bin -> int list` qui donne la liste des étiquettes des feuilles de l'arbre en entrée.

Exercice 10 : relations d'ordre classique

Pour deux listes ℓ_1 et ℓ_2, on s'intéresse aux relations suivantes :

- ℓ_1 est un préfixe de ℓ_2 : par exemple, `[0;5;2]` est un préfixe de `[0;5;2;7;3]` ;
- ℓ_1 est un facteur de ℓ_2 : ℓ_1 apparaît comme un bloc dans ℓ_2. Par exemple, `[0;5;2]` est un facteur de `[7;1;0;5;2;4]` ;
- ℓ_1 est une sous-liste de ℓ_2 : les éléments de ℓ_1 apparaissent dans leur ordre dans ℓ_2, mais pas forcément de façon contiguë. Par exemple, `[0;5;2]` est une sous-liste de `[0;1;5;4;2;3]`.

Écrire trois fonctions testant ces trois relations.

Exercice 11 : dichotomie

Écrire une fonction `dicho : 'a array -> 'a -> bool` qui teste si un élément est dans un tableau trié selon l'algorithme de dichotomie. On testera bien sur les tableaux de longueur 0, 1 et 2.

Exercice 12 : repérage dans une grille

On considère une grille avec n lignes et m colonnes. Raisonnablement, il existe deux manière de se repérer dans la grille : la première selon les coordonnées cartésiennes, et la seconde selon une numérotation de gauche à droite, puis de bas en haut.

(0,1)	(1,1)	(2,1)
(0,0)	(1,0)	(2,0)

3	4	5
0	1	2

Écrire une fonction `num : int*int -> int*int -> int` qui prend en entrée le couple (n, m), ainsi qu'un couple (x, y) correspondant aux coordonnées cartésiennes, et renvoie le numéro de la case. Écrire une fonction `coord : int*int -> int -> int*int` qui prend en entrée le couple (n, m) et le numéro de la case, et renvoie ses coordonnées cartésiennes.

Une autre numérotation est celle en boustrophédon :

(0,3)	(1,3)	(2,3)
(0,2)	(1,2)	(2,2)
(0,1)	(1,1)	(2,1)
(0,0)	(1,0)	(2,0)

11	10	9
6	7	8
5	4	3
0	1	2

Faire la même chose pour le boustrophédon !

Exercice 13 : plateaux

Dans une liste, un plateau est une succession de valeurs identiques : par exemple, pour la liste `['a';'b';'b';'b';'c']`, la succession de `'b'` est un plateau. Informatiquement, on la désignera par les indices extrémaux (ici `(1,3)`). Écrire une fonction `plateau : 'a list -> int*int` qui renvoie les extrémités du plus long plateau d'une liste (on demande une complexité linéaire en la longueur de la liste).

Exercice 14 : produit cartésien

On considère deux ensembles représentés par des listes sans doublons : écrire une fonction `prod_cart : 'a list -> 'b list -> ('a*'b) list` qui renvoie le produit cartésien des listes.

Exercice 15 : découpage quasi-carré

1. On cherche à écrire un entier n « le plus possible comme un carré ». Plus formellement, on cherche x et y entiers positifs tels que $n = x \times y$ et que $|x - y|$ soit minimal. Écrire une fonction `quasicarre : int -> int*int` qui renvoie le couple (x, y) (on demande à ce que $x \leqslant y$).
2. On considère un tableau `t` : on cherche à le transformer en la matrice la plus carrée possible.

1	0	3	2	5	4	7	6	9	8	0	1

1	0	3	2
5	4	7	6
9	8	0	1

   Écrire une fonction `decoup_quasicarre : 'a array -> 'a array array` qui découpe un tableau de la façon la plus carrée possible.

Exercice 16 : fusion de tableaux

On considère deux tableaux triés sans doublons : on cherche à les fusionner. Écrire une fonction `fusion_tab : 'a array -> 'a array -> 'a array` qui renvoie la fusion triée sans doublons des tableaux en entrée (et oui, ce sont bien des tableaux).

Exercice 17 : tableaux partiels

En OCaml, l'un des types natifs est `type 'a option = None | Some of 'a`, appelé « type option ».
On considère des tableaux remplis partiellement : plus formellement, il s'agit de `'a option array`, dont certaines cases sont remplies par des `None`, et les autres par des `Some x`.
Écrire une fonction `prem_case_vide : 'a tab -> int option` qui renvoie l'indice de la case vide la plus à gauche, et `None` s'il n'y en a pas.

Exercice 18 : ordre préfixe

On considère des arbres d'arité quelconque représentés par `type arbre = Feuille of int | Noeud_l of int * arbre list`. On pourra considérer qu'un nœud qui n'est pas une feuille possède au moins un enfant.
Écrire une fonction `ordre_prefixe : arbre -> int list` qui renvoie l'ordre **préfixe** de l'arbre. Les enfants d'un nœud seront à considérer dans l'ordre de la liste.

Exercice 19 : palindrome

Un palindrome est un élément qui se lit pareil dans les deux sens.
Écrire une fonction `est_palindrome_li : 'a list -> bool` qui dit si une liste représente ou non un palindrome.
Écrire une fonction `est_palindrome_tab : 'a array -> bool` qui dit si un tableau représente ou non un palindrome.

Exercice 20 : énumération

Écrire une fonction `enum_bool : int -> bool list list` qui prend en entrée un entier n, et renvoie la liste de toutes les listes booléennes de longueur n (a priori, il y en aura 2^n).

Exercice 21 : sous-maximum

Écrire une fonction `max2 : 'a list -> 'a` qui renvoie le deuxième élément maximal d'une liste. La valeur peut être la même que la valeur maximale et la fonction doit renvoyer une erreur dans les cas absurdes.

Exercice 22 : occurrences dans un texte

Écrire une fonction `occurrences : char list -> (char * int) list` qui renvoie une liste d'association de chacun des caractères de la liste d'entrée.
Quelle est la complexité de votre fonction ?

Le format liste d'association n'est pas des plus pratiques pour rechercher ou modifier des éléments. Pour cela, on se tournera vers les dictionnaires qui en Ocaml peuvent être implémenté à l'aide du module `Hashtbl`.
Pour créer un dictionnaire, on utilisera `Hashtbl.create n` où `n` sera une estimation du nombre final d'entrées du dictionnaire (une meilleure estimation donnera de meilleures performances).
Si `dico` est un dictionnaire ainsi créé, pour obtenir la valeur associée à la clé `cle`, on pourra utiliser `Hashtbl.find dico cle`. Pour ajouter ou modifier une association, on pourra utiliser `Hashtbl.replace dico cle valeur`. Pour vérifier si une clé est associée dans le dictionnaire, on pourra utiliser `Hashtbl.mem dico cle`.
À noter que toutes les clés doivent être d'un seul même type et il en va de même pour toutes les valeurs (qui peut être différent de celui des clés).
Écrire une fonction `occurrences_dict : string -> (char, int) Hashtbl.t`
En considérant que chaque opération sur les dictionnaire s'effectue en $O(\log n)$ avec n le nombre de clés du dictionnaire, quelle est la complexité de la fonction ?

Exercice 23 : matrice vs. liste d'adjacence

Écrire les fonctions `liste_de_matrice : int array array -> int list array` et `matrice_de_liste : int list array -> int array array` qui passent d'une matrice d'adjacence d'un graphe à une liste d'adjacence et inversement.

Exercice 24 : évaluation d'une formule

Implémenter un type cohérent `formule` permettant de représenter les formules logique utilisant les opérateurs : $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, ainsi qu'un type `valuation` représentant une valuation.
Écrire ensuite une fonction `eval : formule -> valuation -> bool` qui évalue une formule.

Exercice 25 : ABR

Les arbres binaires de recherches sont représentés par `type 'a abr = V | N of 'a * 'a abr * 'a abr`.
Écrire une fonction `recherche : 'a -> 'a abr -> bool` qui dit si un élément appartient à l'abr donné en entrée.
Faire de même avec les fonctions `insere : 'a -> 'a abr -> 'a abr` et `supprime : 'a -> 'a abr -> 'a abr` qui respectivement insère et supprime un élément d'un ABR.

Exercice 26 : complexes

On considère `type complexe = {re : float ; im : float}` pour représenter les nombre complexes. Écrire les fonctions suivantes :

- `conj : complexe -> complexe` qui renvoie le conjugué d'un nombre ;
- `plus : complexe -> complexe -> complexe` effectuant la somme, et une fonction similaire pour le produit ;
- `racines : complexe -> (complexe * complexe)` renvoyant les deux racines carrées d'un nombre complexe.

Exercice 27 : files par listes

On vous donne le type suivant :

```
type 'a file = {mutable entree : 'a list ;
                mutable sortie : 'a list};;
```

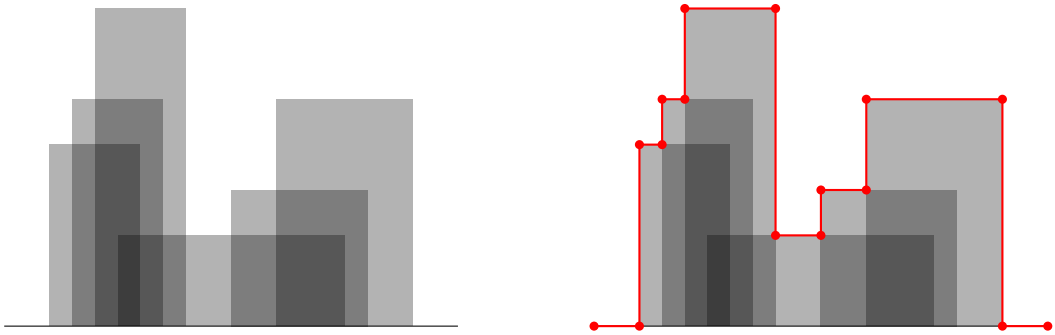
Écrire les trois fonctions `pop`, `push` et `is_empty` associées.

Exercice 28 : exponentiation rapide

1. Écrire une fonction `multmat : int array array -> int array array -> int array array` qui renvoie le produit matriciel (pas en place) de deux matrices. On attend une complexité en $O(n^3)$.
2. Écrire une fonction `expo_r : int array array -> int -> int array array` qui procède à l'exponentiation rapide matricielle d'une matrice. Pour rappel, si M est une matrice et p la puissance souhaitée :
   - si $p = 0$, alors $M^p = I_n$;
   - si $p = 2k$, alors $M^p = (M^k) \times (M^k)$;
   - si $p = 2k + 1$, alors $M^p = (M^k) \times (M^k) \times M$.

Exercice 29 : skyline

On s'intéresse à la ligne d'horizon d'une ville (*skyline*) : des bâtiments forment des rectangles, et on cherche à donner la suite des points se dessinant en haut.



L'ensemble des bâtiments sera représenté par le type `(int * int * int) list` où un bâtiment est représenté par le triplet `(g, d, h)` avec `g` et `d` les abscisses gauche et droite du bâtiment et `h` sa hauteur.
En utilisant le paradigme de diviser pour régner, écrire une fonction `skyline : (int * int * int) list -> (int * int) list`.