1 Pied à l'étrier

R. 1-1 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction lèvera une exception `Invalid_argument` dans le cas où la liste est vide. On rappelle que l'exception `Invalid_argument` est prédéfinie en OCaml, elle doit être accompagnée d'une chaîne de caractères. Par exemple : l'évaluation de l'expression `raise (Invalid_argument("ceci est un message d'erreur"))` lève l'exception `Invalid_argument("ceci est un message d'erreur")`.

R. 1-2 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction doit retourner `None` si la liste est vide et `Some(x)` si la liste contient au moins un élément et que le dernier élément est `x`.

R. 1-3 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On n'utilisera pas de fonctionnelle d'itération.

R. 1-4 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

R. 1-5 Écrire une fonction permettant de calculer la concaténation de deux listes. On n'utilisera pas de fonctionnelle d'itération.

R. 1-6 Écrire une fonction permettant de calculer la concaténation de deux listes. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

R. 1-7 Écrire une fonction prenant en argument une liste `l` et un élément `x` et calculant la liste `l` privée de toutes les occurrences de l'élément `x`.

R. 1-8 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction doit faire appel à une fonction auxiliaire récursive terminale.

R. 1-9 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction doit être récursive et ne pas faire appel à des fonctions récursives auxiliaires.

R. 1-10 Écrire une fonction prenant en argument une liste `l` et la découpant en deux listes, dans la première on rangera les éléments d'indices pairs dans `l`, dans la seconde on rangera les éléments d'indices impairs dans `l`. L'ordre des éléments dans les listes résultats doit être celui de la liste d'entrée.

2 Listes de listes

R. 1-11 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On utilisera la fonction de concaténation de listes `@`, en prenant garde à ne pas obtenir une complexité quadratique.

R. 1-12 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera deux fonctions mutuellement récursives.

R. 1-13 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera une unique fonction auxiliaire récursive terminale.

R. 1-14 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes, on se limitera à des itérations au moyen de la fonctionnelle `List.fold_left`.

3 Listes et comparaisons

R. 1-15 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive et ne pas utiliser de fonctions récursives auxiliaires. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-16 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive terminale et peut utiliser une fonction récursive auxiliaire. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-17 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit utiliser la fonctionnelle `List.fold_left` comme mécanisme d'itération. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-18 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) croissantes maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 5; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7]; [5]; [2; 3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`.

R. 1-19 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) monotones maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7; 5; 2]; [3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`

R. 1-20 Écrire une fonction prenant en arguments une liste `(l: 'a list)`, un entier `k`, une fonction `f: 'a -> int` à valeurs dans $[\![0, \mathtt{k}]\!]$ et retournant la liste des éléments de `l` triés par image par `f` croissante. On demande une implémentation en $\mathcal{O}(\max(n, k))$ où n est la taille de la liste `l`.

4 Quelques utilisations classiques des listes en algorithmique

R. 1-21 Écrire une fonction permettant de trier une liste au moyen de l'algorithme du tri fusion♣.

♣. https://en.wikipedia.org/wiki/Merge_sort

R. 1-22 Fournir une implémentation du type de données abstrait file au moyen de deux listes. On assurera une complexité amortie en $\mathcal{O}(1)$ pour chaque opération.

R. 1-23 Écrire une fonction prenant en argument une liste et retournant la liste des permutations de cette liste.

R. 1-24 Écrire une fonction prenant en argument un entier naturel n et retournant la liste des 2^n listes contenant n booléens. Ainsi dans la liste résultat on devra pouvoir trouver chaque liste de n booléens.

5 Produit de listes

R. 1-25 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, ainsi qu'une fonction de type `'a -> 'b -> unit` qui appelle cette fonction sur tous les couples d'éléments `(a, b)` avec `a`∈`l1` et `b`∈`l2`. Cette fonction ne doit utiliser aucune fonctionnelle d'itération dans cette fonction.

R. 1-26 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, ainsi qu'une fonction de type `'a -> 'b -> unit` qui appelle cette fonction sur tous les couples d'éléments `(a, b)` avec `a`∈`l1` et `b`∈`l2`. Cette fonction doit utiliser la fonctionnelle d'itération `List.iter`.

R. 1-27 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, et calculant une liste de type `('a * 'b) list` représentant le produit cartésien de `l1` et `l2`. Par exemple le produit de `[1; 2; 2]` et `['a'; 'b']` peut être représenté par `[(3, 'b'); (3, 'a'); (2, 'b'); (2, 'a'); (1, 'b'); (1, 'a')]` ou n'importe quelle permutation de cette liste. Cette fonction doit utiliser une référence et la fonctionnelle d'itération `List.iter`.

R. 1-28 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, et calculant une liste de type `('a * 'b) list` représentant le produit cartésien de `l1` et `l2`. Par exemple le produit de `[1; 2; 2]` et `['a'; 'b']` peut être représenté par `[(3, 'b'); (3, 'a'); (2, 'b'); (2, 'a'); (1, 'b'); (1, 'a')]` ou n'importe quelle permutation de cette liste. Cette fonction ne doit utiliser aucune fonctionnelle d'itération, en particulier ni `List.iter` ni `List.fold_left`.

1 Pied à l'étrier

R. 2-1 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

R. 2-2 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `for`, arrêtée au moyen d'une levée d'exception. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

R. 2-3 Écrire une fonction calculant l'indice du *dernier* `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

R. 2-4 Écrire une fonction testant l'existence d'un `0` dans un tableau d'entiers avec une boucle `while`, arrêtée dès que possible sans utiliser d'exception. La fonction doit retourner `true` s'il est possible de trouver la valeur 0 dans le tableau et `false` sinon.

R. 2-5 Écrire une fonction testant l'existence d'un `0` dans un tableau d'entiers avec une fonctionnelle `Array.iter`, arrêtée par une levée d'exception. La fonction doit retourner `true` s'il est possible de trouver la valeur 0 dans le tableau et `false` sinon.

R. 2-6 Écrire une fonction prenant en argument un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen d'une boucle `for`.

R. 2-7 Écrire une fonction prenant en argument un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen de la fonctionnelle `Array.fold_left`.

R. 2-8 Écrire une fonction calculant l'indice du minimum dans un tableau d'entiers parcouru par une boucle `for`. Cette fonction déclenchera l'exception `Invalid_argument` dans le cas où le tableau est de taille `0`.

R. 2-9 Écrire une fonction calculant l'indice du maximum dans un tableau d'entiers positifs parcouru par une fonctionnelle `Array.fold_left`. Cette fonction déclenchera l'exception `Invalid_argument` dans le cas où le tableau est de taille `0`.

R. 2-10 Écrire une fonction prenant en argument un tableau d'entiers `t` et calculant le tableau des sommes cumulées du tableau `t`. Ainsi la case d'indice `i` du tableau résultat doit contenir `t.(0) + t.(1) + ... + t.(i)`. On fournira une implémentation en $\mathcal{O}(n)$.

R. 2-11 Définir une fonction prenant en argument un tableau de listes d'entiers `t` et le modifiant en ajoutant à chaque liste l'entier indice de la case du tableau dans laquelle se trouve la liste. On itérera sur le tableau au moyen d'une boucle `for`

R. 2-12 Définir une fonction prenant en argument un tableau de listes d'entiers `t` et un entier `x` et retournant un tableau de listes d'entiers obtenu par ajout de l'entier en tête de chacune des listes se trouvant dans le tableau. On itérera sur le tableau au moyen d'un `Array.map`.

R. 2-13 Écrire une fonction prenant en argument un tableau d'entiers t, contenant des valeurs entières dans un intervalle $[\![0, m-1]\!]$ où m vous est passé en argument, et calculant l'entier de $[\![0, m-1]\!]$ ayant le plus d'occurrences dans le tableau. On s'efforcera d'itérer sur les tableaux aux moyens de fonctionnelles et non de boucle `for`/`while`. On fournira une implémentation en $\mathcal{O}(n+m)$.

2 Tris

R. 2-14 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri à bulles♣.

R. 2-15 Écrire une fonction qui prend en argument un tableau `t` et deux indices `g` et `d` valides dans `t`, et qui décale d'une case vers la droite, tous les éléments d'indices dans $[\![\mathtt{g}, \mathtt{d}-1]\!]$. Le contenu de la case d'indice `d` est écrasé par cette opréation

R. 2-16 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par insertion♡.

R. 2-17 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par sélection♠.

R. 2-18 Écrire une fonction partition prenant en argument un tableau `t`, deux indices `d` (début) et `f` (fin) et `ip` un indice du sous-tableau $\mathtt{t}[\mathtt{d}, \mathtt{f}]$ et qui permute $\mathtt{t}[\mathtt{d}, \mathtt{f}]$ de manière à placer d'abord les éléments inférieurs à la valeur pivot `t.(ip)`, puis cette valeur pivot, à l'indice `q` qu'elle retournera, et enfin ceux strictement supérieurs à cette valeur.

R. 2-19 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri rapide.

R. 2-20 Écrire une fonction permettant de calculer la médiane d'un tableau d'entiers sans d'abord trier ce tableau. On s'attend à une complexité en $\mathcal{O}(n^2)$.

3 Chaînes de caractères

R. 2-21 Écrire une fonction `string_depuis_char_list : char list -> string` prenant en paramètre une liste de caractères et calculant la chaîne de caractères constituées des caractères de cette liste, dans le même ordre. Par exemple `(string_depuis_char_list ['a'; 'b'; 'c'] = "abc")`.

R. 2-22 Écrire une fonction `char_list_depuis_string : string -> char list` prenant en paramètre une chaîne de caractères et calculant la liste des caractères la constituant, dans le même ordre. Par exemple `(string_depuis_char_list "abc" = ['a'; 'b'; 'c'])`.

R. 2-23 Écrire une fonction `int_depuis_string : string -> int option` prenant en paramètre une chaîne de caractères et calculant, s'il existe, l'entier représenté par cette chaîne de caractères. Par exemple `(int_depuis_string "0123" = Some 123)` mais `(int_depuis_string "01a23" = None)`.

4 Autres utilisations classiques des tableaux en algorithmique

R. 2-24 Écrire une fonction prenant en argument un tableau d'entiers, triés par ordre croissant, de taille n, un entier `p` et testant si l'entier `p` apparaît dans le tableau. L'algorithme devra être récursif et ne pas manipuler de boucles `while`/`for`. On assurera une complexité en $\mathcal{O}(\log(n))$.

R. 2-25 Écrire une fonction prenant en argument un tableau d'entiers, triés par ordre croissant, de taille n, un entier `p` et testant si l'entier `p` apparaît dans le tableau. L'algorithme utilisé ne devra pas être récursif. On assurera une complexité en $\mathcal{O}(\log(n))$.

♣. https://en.wikipedia.org/wiki/Bubble_sort
♡. https://en.wikipedia.org/wiki/Insertion_sort
♠. https://en.wikipedia.org/wiki/Selection_sort

R. 2-26 Écrire une fonction prenant en argument un tableau de booléens représentant un entier écrit en base 2 (les bits de poids faibles sont à droite) et modifiant le tableau pour qu'il représente l'entier suivant. On lèvera l'exception `Invalid_argument "dernier"` s'il n'est pas possible d'incrémenter l'entier.

R. 2-27 On représente une permutation de $[\![0, n-1]\!]$ par un tableau d'entiers `t` tel que l'image de tout entier `i` de $[\![0, n-1]\!]$ par la permutation est `t.(i)`. On représente un cycle (au sens des permutations) par la liste des éléments de son support, dans l'ordre du cycle, ainsi `[0; 1; 2]` est la permutation qui envoie 0 sur 1, 1 sur 2 et 2 sur 0. On rappelle qu'une permutation se décompose de manière unique en un produit de cycles à supports disjoints. On représente un produit de cycles à supports disjoints comme un liste des cycles qui le composent. Leurs supports étant disjoints, ces cycles commutent, ainsi l'ordre dans la liste n'importe pas. Donner une fonction calculant la décomposition d'une permutation en produit de cycles à supports disjoints.

R. 2-28 Écrire une fonction prenant en argument un tableau `t` de taille n et une valeur par défaut `x` de même type que celui des éléments du tableau et retournant un tableau de taille $2n$ dont les n premiers éléments sont ceux de `t` et les n derniers contiennent la valeur par défaut.

R. 2-29 Implémenter une structure de table dynamique, c'est-à-dire une structure de tableau telle que l'ajout et la suppression d'éléments en fin de tableau ont un coût amorti constant.

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation d'arbres binaires non vides, d'arbres généraux non vides.

```
type 'a btree =
  | E                              (* arbre vide *)
  | N of 'a * 'a btree * 'a btree (* noeud *)

type 'a gtree =
  | GN of 'a * 'a gtree list
```

1 Pied à l'étrier

R. 3-1 Définir une fonction permettant de tester si un arbre binaire est vide.

R. 3-2 Définir une fonction calculant la hauteur d'un arbre binaire. On rappelle que la hauteur de l'arbre vide est -1.

R. 3-3 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On itérera sur l'arbre général au moyen de deux fonctions mutuellement récursives.

R. 3-4 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On itérera dans l'arbre général au moyen d'une unique fonction récursive et d'un `List.fold_left`.

R. 3-5 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On fournira une implémentation récursive terminale.

R. 3-6 Définir une fonction permettant de tester si une étiquette apparaît dans un arbre binaire.

R. 3-7 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction sera récursive et ne devra pas faire usage d'une fonction récursive auxiliaire.

R. 3-8 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction ne sera pas récursive, elle devra faire appel à une fonction récursive auxiliaire qui sera récursive terminale.

R. 3-9 Définir une fonction prenant en paramètres deux arbres binaires et testant si ceux-ci ont la même forme, c'est-à-dire si les deux arbres sont égaux lorsqu'on omet les valeurs des étiquettes.

R. 3-10 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `Some(p)` où `p` est la profondeur d'un nœud d'étiquette paire s'il en existe et `None` sinon. On s'efforcera d'interrompre la recherche dès que cela est possible, en utilisant un mécanisme d'exception.

2 Parcours d'arbres

R. 3-11 Définir une fonction calculant le parcours infixe d'un arbre binaire. On pourra fournir une implémentation de complexité quadratique.

R. 3-12 Définir une fonction calculant le parcours infixe d'un arbre binaire. On fournira une implémentation de complexité linéaire.

R. 3-13 Définir une fonction prenant en paramètre un arbre binaire a de type `'a btree` et deux éléments x et y de type `'a` et calculant l'arbre binaire obtenu en remplaçant, dans a, toutes les occurences x par y.

R. 3-14 Définir une fonction calculant la bijection vue en première année entre l'ensemble des listes d'arbres généralisés (`'a gtree list`) et les arbres binaires (`'a btree`). Définir ensuite sa réciproque.

R. 3-15 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive n'utilisant pas de fonction récursive auxiliaire. Cette implémentation peut être non récursive terminale, de complexité quadratique.

R. 3-16 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire.

R. 3-17 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant le module `Stack`.

R. 3-18 Définir une fonction retournant un parcours en largeur d'un arbre binaire (la liste de ses étiquettes, triées par profondeur, et à profondeur équivalente de gauche à droite). On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant le module `Queue`.

R. 3-19 Définir une fonction retournant un parcours en largeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant deux listes : celle du niveau courant, celle du niveau suivant.

3 Utilisations classiques en algorithmique

R. 3-20 Définir une fonction permettant de tester l'appartenance d'un élément dans un arbre binaire de recherche. On fournira une implémentation ayant une complexité linéaire en la hauteur de l'arbre considéré.

R. 3-21 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et testant si l'arbre est de recherche, à savoir si pour tout nœud de l'arbre d'étiquette e, toute étiquette de son fils gauche est inférieure stricte à e, et toute étiquette de son fils droit est supérieur à e. On fournira une implémentation ayant une complexité en $\mathcal{O}(n^2)$.

R. 3-22 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et testant si l'arbre est de recherche, à savoir si pour tout nœud de l'arbre d'étiquette e, toute étiquette de son fils gauche est inférieure à e, et toute étiquette de son fils droit est supérieur à e. On fournira une implémentation ayant une complexité en $\mathcal{O}(n)$.

R. 3-23 Définir une fonction prenant en paramètre un arbre binaire et testant si celui-ci est parfait c'est à dire que tout nœud a soit deux fils vides (autrement dit est une feuille), soit deux fils non vides et que toutes les feuilles de l'arbre ont même profondeur.

R. 3-24 Définir une fonction prenant en paramètre un arbre binaire et testant si celui-ci est parfait c'est à dire que tout nœud a soit deux fils vides (autrement dit est une feuille), soit deux fils non vides et que toutes les feuilles de l'arbre ont même profondeur. On fournira une implémentation ayant une complexité en $\mathcal{O}(n)$.

R. 3-25 Implémenter une fonction de tri par arbre binaire de recherche : pour trier une liste d'éléments, on insère successivement ces éléments dans un arbre binaire de recherche, un parcours infixe de l'arbre binaire de recherche fourni alors un tri de la liste initiale.

R. 3-26 Implémenter une fonction de tri par tas : pour trier un tableau d'éléments, on insère successivement ces éléments dans une file de priorité implémentée par un tournoi complet, des extractions successives du maximum permettent alors d'obtenir un tri de la liste initiale.

1 Création de tableaux

Dans toutes les questions on utilise les types suivants.

```c
// struct pour les tableaux d'entiers munis de leur taille
struct tableau_s {
  int  taille ;
  int* tab    ;
};
typedef struct tableau_s tableau;
```

```c
// struct pour les tableaux de tableaux d'entiers munis de leur taille
struct tabtab_s {
  int      taille ;
  tableau* tab    ;
};
typedef struct tabtab_s tabtab;
```

```c
// struct pour les tableaux de booléens munis de leur taille
struct btableau_s {
  int   taille ;
  bool* tab    ;
};
typedef struct btableau_s btableau;
```

R. 4-1 Écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau de n booléens rempli de `false` à l'aide d'une boucle **`while`**.

R. 4-2 Écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau contenant les entiers de 1 à n à l'aide d'une boucle **`for`**.

R. 4-3 Écrire une fonction prenant en paramètre un entier naturel non nul n, et deux entiers relatifs a et b tels que $a < b$ et retournant un tableau de n entiers aléatoires tirés uniformément dans $[\![a, b[\![$.

2 Agrégation

R. 4-4 Écrire une fonction testant l'existence d'un **`0`** dans un tableau d'entiers parcouru par boucle **`while`**. La fonction doit retourner `true` s'il est possible de trouver la valeur **`0`** dans le tableau et `false` sinon.

R. 4-5 Écrire une fonction calculant l'indice du premier **`0`** dans un tableau d'entiers parcouru par boucle **`while`** arrêtée dès que possible, sans utiliser de **`break`**, ni de **`return`** dans la boucle. La fonction devra retourner **`-1`** si le tableau ne contient aucun **`0`**.

R. 4-6 Écrire une fonction calculant l'indice du premier **`0`** dans un tableau d'entiers parcouru par boucle **`while`** arrêtée dès que possible grâce à une instruction **`return`** dans la boucle. La fonction devra retourner **`-1`** si le tableau ne contient aucun **`0`**.

R. 4-7 Écrire une fonction calculant l'indice du premier **`0`** dans un tableau d'entiers parcouru par boucle **`for`** arrêtée dès que possible. La fonction devra retourner **`-1`** si le tableau ne contient aucun **`0`**.

R. 4-8 Écrire une fonction calculant l'indice du dernier `0` dans un tableau d'entiers parcouru par boucle `while` arrêtée dès que possible grâce à une instruction `return` dans la boucle. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

R. 4-9 Écrire une fonction prenant en paramètre un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen d'une boucle `for`.

R. 4-10 Écrire une fonction calculant l'indice du minimum dans un tableau d'entiers non vide parcouru par une boucle `for`. En cas d'ambiguïté, on retournera le plus petit indice du minimum.

R. 4-11 Écrire une fonction calculant la conjonction d'un tableau de booléens parcouru par une boucle `while`.

R. 4-12 Écrire une fonction prenant en paramètre un tableau d'entiers à valeurs dans $[\![0, m-1]\!]$ où m est donné en paramètre, et calculant l'entier de $[\![0, m-1]\!]$ ayant le plus d'occurrences dans le tableau. On fournira une implémentation en $\mathcal{O}(n+m)$.

3 Tableaux à partir d'un tableau

R. 4-13 Écrire une fonction qui copie un tableau d'entiers.

R. 4-14 Écrire une fonction prenant en paramètre `t` un tableau d'entiers de taille n et un entier par défaut `x` et retournant un tableau de taille $2n$ dont les n premiers éléments sont ceux de `t` et les n derniers contiennent la valeur par défaut.

R. 4-15 Écrire une fonction prenant en paramètre un tableau d'entiers `t` et retournant le tableau miroir de `t`.

R. 4-16 Écrire une fonction prenant en paramètre un tableau d'entiers `tailles` et retournant `t` un tableau de tableaux d'entiers initialisé à 0, de même taille que le tableau `tailles`, et tel que pour tout indice `i` valide pour ces tableaux, `t[i]` est de taille `tailles[i]`.

R. 4-17 Écrire une fonction prenant en paramètre un tableau d'entiers `t` et retournant le tableau des sommes cumulées du tableau `t`. Ainsi la case d'indice `i` du tableau résultat doit contenir `t[0] + t[1] + ... + t[i]`. On fournira une implémentation de complexité linéaire en la taille du tableau.

R. 4-18 Écrire une fonction qui prend en paramètres deux tableaux d'entiers triés et qui crée le tableau trié obtenu en interclassant les éléments des deux tableaux. Le nombre d'occurrence de chaque élément dans le tableau résultat est la somme de ceux dans chacun des tableaux en paramètre. Cette fonction devra être en complexité linéaire, c'est-à-dire en $\mathcal{O}(n_1+n_2)$ où n_1 et n_2 sont respectivement les tailles des deux tableaux pris en paramètre.

R. 4-19 Écrire une fonction permettant de distinguer les sous-séquences (non vides) croissantes maximales pour l'extension à gauche d'un tableau d'entiers non vide en indiquant où elles commencent, c'est-à-dire retournant le tableau trié des indices où commencent ces sous-séquences, muni de sa taille, *i.e.* le nombre de telles sous-séquences. Par exemple pour `{1, 4, 5, 8, 7, 5, 2, 3, 4, 9, 3, 3, 3, 5, 5, 7, 2}`, le tableau des indices de découpage est `{0, 4, 5, 6, 10, 16}` correspondant aux 6 sous-séquences `{1, 4, 5, 8}`, `{7}`, `{5}`, `{2, 3, 4, 9}`, `{3, 3, 3, 5, 5, 7}` et `{2}`. Cette fonction pourra allouer un tableau temporaire, mais ne devra parcourir qu'une fois de tableau pris en paramètre.

R. 4-20 Écrire une fonction permettant de distinguer les sous-séquences (non vides) croissantes maximales pour l'extension à gauche d'un tableau d'entiers non vide en indiquant où elles commencent, c'est-à-dire retournant le tableau trié des indices où commencent ces sous-séquences, muni de sa taille, *i.e.* le nombre de telles sous-séquences. Par exemple pour `{1, 4, 5, 8, 7, 5,`

`2, 3, 4, 9, 3, 3, 3, 5, 5, 7, 2}`, le tableau des indices de découpage est `{0, 4, 5, 6, 10, 16}` correspondant aux 6 sous-séquences `{1, 4, 5, 8}`, `{7}`, `{5}`, `{2, 3, 4, 9}`, `{3, 3, 3, 5, 5, 7}` et `{2}`. Cette fonction pourra parcourir plusieurs fois de tableau pris en paramètre, mais ne devra pas allouer d'espace mémoire en dehors de celui nécessaire pour enregistrer le tableau résultat.

R. 4-21 Écrire une fonction permettant de découper un tableau d'entiers en sous-séquences non vides de somme inférieure à 10 maximales pour l'extension à gauche. Par exemple `{1, 2, 8, 2, 3, 4, 1}` est découpé en `{{1, 2}, {8, 2}, {2, 3, 4, 1}}`.

4 Tableaux représentant des matrices

Dans toutes les questions on utilise les types suivants.

```c
//struct pour les matrices dans un tableau unidimensionnel
struct matrice_unidimensionel_s {
  int nbl;  //nombre de lignes
  int nbc;  //nombre de colonnes
  int* tab; //tableau des coeffs de taille nbl*nbc
};
typedef struct matrice_unidimensionel_s matrice_u;
```

```c
//struct pour les matrices dans un tableau bidimensionnel
struct matrice_bidimensionel_s {
  int nbl;  //nombre de lignes
  int nbc;  //nombre de colonnes
  int** tab;//tableau de nbl tableaux de nbc coeffs
};
typedef struct matrice_bidimensionel_s matrice_b;
```

R. 4-22 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètre une matrice M et les indices i et j d'un coefficient de M, et qui renvoie le coefficient $M_{i,j}$.

R. 4-23 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètres une matrice M et les indices i et j d'un coefficient de M, et une valeur entière x, et qui donne au coefficient d'indice (i, j) dans cette matrice la valeur x.

R. 4-24 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau d'entiers unidimensionnel représentant I_n, la matrice identité de dimensions $n \times n$.

R. 4-25 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètre une matrice M et les indices i et j d'un coefficient de M, et qui renvoie le coefficient $M_{i,j}$.

R. 4-26 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètres une matrice M et les indices i et j d'un coefficient de M, et une valeur entière x, et qui donne au coefficient d'indice (i, j) dans cette matrice la valeur x.

R. 4-27 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau d'entiers unidimensionnel représentant I_n, la matrice identité de dimensions $n \times n$.

5 Tris

R. 4-28 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri à bulles♣.

R. 4-29 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par insertion♡.

R. 4-30 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par sélection♠.

R. 4-31 Écrire une fonction partition prenant en arguments un tableau `t` et `ip` un indice de `t` et qui modifie `t` en place de manière à placer d'abord les éléments inférieurs à la valeur pivot `t[ip]`, puis cette valeur pivot, à un indice `q` qu'elle retournera, et enfin ceux strictement supérieurs à cette valeur. En déduire une fonction permettant de trier un tableau au moyen de l'algorithme du tri rapide.

6 Autres utilisations classiques des tableaux en algorithmique

R. 4-32 Écrire une fonction prenant en paramètre un tableau d'entiers, triés par ordre croissant, de taille n, un entier `p` et testant si l'entier `p` apparaît dans le tableau. Cette fonction devra être récursive et ne pas manipuler de boucles `while`/`for`. On assurera une complexité en $\mathcal{O}(\log(n))$.

R. 4-33 Écrire une fonction prenant en paramètre un tableau d'entiers, triés par ordre croissant, de taille n, un entier `p` et testant si l'entier `p` apparaît dans le tableau. Cette fonction ne devra pas être récursive. On assurera une complexité en $\mathcal{O}(\log(n))$.

R. 4-34 Écrire une fonction prenant en paramètre un tableau de booléens représentant un entier écrit en base 2 (les bits de poids faibles sont à droite) et modifiant le tableau pour qu'il représente l'entier suivant. Cette fonction renverra un booléen indiquant si l'incrémentation a bien été faite, de sorte qu'on renverra `false` s'il n'est pas possible d'incrémenter l'entier, en pareil cas le tableau sera modifié pour ne contenir que des `false`.

R. 4-35 Écrire une fonction permettant de calculer la médiane d'un tableau d'entiers deux à deux distincts sans trier ce tableau. On attend une fonction de complexité pire cas quadratique.

♣. https://en.wikipedia.org/wiki/Bubble_sort
♡. https://en.wikipedia.org/wiki/Insertion_sort
♠. https://en.wikipedia.org/wiki/Selection_sort

1 Différents types de listes chaînées

R. 5-1 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes.

R. 5-2 Définir une structure de cellule pour les listes *doublement* chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes, qui doivent pouvoir être parcourues depuis la tête ou depuis la queue.

R. 5-3 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier *et* d'une chaîne de caractères. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

R. 5-4 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier et d'un tableau. On peut imaginer par exemple qu'un élément est un sommet muni du tableau de ses voisins dans un graphe. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

2 Listes d'entiers simplement chaînées

Dans les questions suivantes on travaille avec le type ci-dessous.

```c
typedef struct s_cell cell;
struct s_cell {
  int val ;
  cell* next ; /* adresse de la cell. suivante ou NULL en fin de liste */
};

/* adresse de la 1ere cellule ou NULL pr la liste vide */
typedef cell* liste_c;
```

R. 5-5 Définir une fonction qui crée une liste chaînée d'entiers réduite à une cellule. L'entier à enregistrer dans cette cellule sera pris en paramètre.

R. 5-6 Définir une fonction qui affiche une liste chaînée d'entiers. Cette fonction doit pouvoir s'exécuter sur une liste vide, et fournir alors un affichage adéquat.

R. 5-7 Définir une fonction qui prend en paramètre un entier $n \in \mathbb{N}$ et qui renvoie une liste chaînée contenant les n premiers entiers naturels. L'itération de cette fonction devra être réalisée avec une boucle `for`.

R. 5-8 Définir une fonction qui libère l'espace mémoire alloué pour une liste chaînée d'entiers.

R. 5-9 Définir une fonction qui crée une liste chaînée d'entiers à partir d'un tableau d'entiers.

R. 5-10 Définir une fonction qui crée un tableau d'entiers à partir d'une liste chaînée d'entiers. L'itération de cette fonction devra être réalisée avec une boucle `for`.

R. 5-11 Définir une fonction qui ajoute un à un les éléments d'une liste `l1` dans une autre liste `l2`. Les listes `l1` et `l2` ne doivent pas être dégradées par cette procédure. On renverra la nouvelle liste, dans laquelle les éléments de `l1` apparaissent avant ceux de `l2`, en ordre inverse. Par exemple pour `l1` la liste $1/2/3$ et `l2` la liste $4/5/6$ le résultat attendu est la liste $3/2/1/4/5/6$.

R. 5-12 Définir une fonction qui calcule le miroir d'une liste `ll` sans la dégrader.

R. 5-13 Définir une fonction qui crée une copie d'une liste `ll` sans la dégrader.

R. 5-14 Définir une fonction qui prend en paramètre une liste et un entier naturel k inférieur à la taille de la liste, et qui supprime les éléments de cette liste au delà des k premiers. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

R. 5-15 Définir une fonction qui prend en paramètres une liste et un entier naturel k, et qui supprime les éléments de cette liste au delà des k premiers, s'il y en a. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

R. 5-16 Définir une fonction qui prend en paramètre une liste passée par référence et la transforme en place en la liste miroir. Cette fonction ne devra pas allouer d'espace mémoire supplémentaire, mais pourra en revanche faire usage de la pile d'appels.

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation de graphes (orientés ou non) par table de listes d'adjacences, par matrice d'adjacence.

```
type graphe_adj =  (* graphes par table de listes d'adjacence *)
  int list array

type graphe_mat =  (* graphes par matrice d'adjacence *)
  bool array array
```

1 Pied à l'étrier

R. 6-1 Définir une fonction OCaml prenant en paramètres un graphe `g` (représenté par matrice d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arête $\{\texttt{i}, \texttt{j}\}$ est une arête du graphe `g`.

R. 6-2 Définir une fonction OCaml prenant en paramètres un graphe `g` (représenté par table de listes d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arête $\{\texttt{i}, \texttt{j}\}$ est une arête du graphe `g`.

R. 6-3 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, \texttt{n}-1]\!]$ et les arcs sont les éléments de `l`. On peut supposer que `l` est sans doublons. On déclenchera une erreur si `l` contient un couple qui ne peut être un arc possible de ce graphe.

R. 6-4 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe non orienté (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, \texttt{n}-1]\!]$ et les arêtes sont données par les couples de `l`. On peut supposer que les couples de `l` représentent deux à deux des arêtes différentes. On déclenchera une erreur si `l` contient un couple qui ne correspond pas à une arête possible de ce graphe.

R. 6-5 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par matrice d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, \texttt{n}-1]\!]$ et les arcs sont données par les éléments de `l`.

R. 6-6 Définir une fonction OCaml prenant en paramètre un graphe orienté (représenté par table de listes d'adjacence) et retournant la liste de ses arcs.

R. 6-7 Définir une fonction OCaml prenant en paramètre un graphe orienté (représenté par matrice d'adjacence) et retournant la liste de ses arcs.

R. 6-8 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en paramètres un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^4)$.

R. 6-9 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement s'il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en paramètres un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^3)$, on pourra pour cela s'aider de l'algorithme de Roy-Warshall.

R. 6-10 On considère un graphe g orienté, à n sommets dont les arcs sont *pondérés* par des longueurs qui sont des entiers positifs. Un tel graphe est représenté en mémoire au moyen d'une matrice d'entiers de dimension $n \times n$ contenant dans la case d'indice (i, j) la pondération de l'arc (i, j) ou -1 si un tel arc n'existe pas. Définir une fonction `floyd_warshall : int array array -> int array array` appliquant l'algorithme de Floyd-Warshall au graphe qui lui est passé en paramètre et retournant donc la matrice des plus courts chemins. Ainsi La matrice résultat devra contenir dans sa case d'indice (i, j) ou bien un entier positif indiquant la longueur du plus court chemin de i à j dans le graphe g, ou bien -1 si un tel chemin n'existe pas.

2 Parcours de graphes

Dans cette partie les graphes seront tous représentés par table de listes d'adjacence (*i.e.* avec le type `graphe_adj`).

R. 6-11 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser la pile des appels récursifs, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n + m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-12 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Stack`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

R. 6-13 Définir une fonction OCaml calculant un parcours en largeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Queue`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

3 Application des parcours

Dans cette partie les graphes seront représentés par table de listes d'adjacence (ou table de listes de successeurs pour les graphes orientés), *i.e.* par le type OCaml `graphe_adj` défini plus haut.

R. 6-14 Définir une fonction OCaml prenant en paramètre un graphe non orienté et deux sommets de ce graphe, et testant si l'un est accessible depuis l'autre.

R. 6-15 Définir une fonction OCaml prenant en paramètre un graphe non orienté et testant si celui-ci est connexe.

R. 6-16 Définir une fonction OCaml prenant en paramètre un graphe non orienté et testant si celui-ci est un arbre.

R. 6-17 Définir une fonction OCaml prenant en paramètre un graphe non orienté et retournant sa décomposition en composantes connexes sous la forme d'un tableau associant à chaque sommet un identifiant unique de composante.

R. 6-18 Définir une fonction OCaml prenant en paramètre un graphe non orienté et retournant si ce graphe est biparti.

R. 6-19 Définir une fonction OCaml prenant en paramètres un graphe non orienté, deux sommets u et v et retournant la distance, en nombre d'arêtes séparant u et v.

R. 6-20 Définir une fonction OCaml prenant en paramètre un graphe orienté supposé sans circuit et retournant, *à l'aide d'un parcours* un tri topologique sous la forme d'une liste de sommets.

R. 6-21 Définir une fonction OCaml prenant en paramètre un graphe orienté et retournant un booléen indiquant s'il admet un circuit.

R. 6-22 Définir une fonction OCaml prenant en paramètre un graphe orienté et retournant un `int list option` qui donne un circuit si le graphe en admet un, et qui vaut `None` sinon.

Dans les questions suivantes on considère les types définis ci-dessous, permettant la représentation de graphes orientés et non orientés en C.

```c
struct list_s {
  int            elem ;         /* l'élément contenu dans le maillon */
  struct list_s* suivant ;      /* le maillon suivant */
};

/* Une liste est un pointeur vers son premier maillon */
typedef struct list_s* list;

struct graph_s {
  int   taille  ;               /* le nombre de sommets */
  list* contenu ;               /* le tableau des listes d'adjacence*/
};
typedef struct graph_s* graphe ;
```

1 Graphes en C

R. 7-1 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en profondeur de ce graphe. La gestion des sommets à visiter devra être faite par le mécanisme de récursivité.

R. 7-2 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en profondeur de ce graphe. La gestion des sommets à visiter devra être faite par une pile implémentée au moyen d'un tableau dynamique.

R. 7-3 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en largeur de ce graphe. La gestion des sommets à visiter devra être faite par une file implémentée au moyen de deux piles.

R. 7-4 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en largeur de ce graphe. La gestion des sommets à visiter devra être faite par une file implémentée dans un tableau de taille n^2.

R. 7-5 Donner une fonction C, prenant en paramètre un graphe orienté et testant si ce graphe contient un circuit.