Exercices de type B

Exercice 1. L'exercice suivant est à traiter dans le langage OCaml.

1. Écrire une fonction `somme : int array -> int -> int` telle que l'appel `somme t i` calcule la somme partielle $\sum_{k=0}^{i} t.(k)$ des valeurs du tableau t entre les indices 0 et i inclus.

Un tableau t de $n > 0$ éléments de $[\![0, n-1]\!]$ est dit *autoréférent* si pour tout indice $0 \leq i < n$, $t.(i)$ est exactement le nombre d'occurrences de i dans t, c'est-à-dire que

$$\forall i \in [\![0, n-1]\!], \quad t.(i) = \text{card}\left(\{k \in [\![0, n-1]\!] \mid t.(k) = i\}\right)$$

Ainsi, par exemple, pour $n = 4$, le tableau suivant est autoréférent :

i	0	1	2	3
$t.(i)$	1	2	1	0

En effet, la valeur 0 existe en une occurrence, la valeur 1 en deux occurrences, la valeur 2 en une occurrence et la valeur 3 n'apparaît pas dans t.

2. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour $n \in [\![1; 3]\!]$ et trouver un autre tableau autoréférent pour $n = 4$.
3. Écrire une fonction `est_auto : int array -> bool` qui vérifie si un tableau de taille $n > 0$ est autoréférent. On attend une complexité en $O(n)$.

On propose d'utiliser une méthode de retour sur trace (*backtracking*) pour trouver tous les tableaux autoréferents pour un $n > 0$ donné. Une fonction `gen_auto` qui affiche tous les tableaux autoréférents pour une taille donnée vous est proposée. Cette version ne fonctionne cependant que pour de toutes petites valeurs de n (instantané pour $n = 5$, un peu long pour $n = 8$, sans espoir pour $n = 15$). On pourra vérifier qu'il existe exactement deux tableaux autoréférents pour $n = 4$, un seul pour $n \in \{5, 7, 8\}$ et aucun pour $n = 6$.

Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

4. Que peut-on dire de la somme des éléments d'un tableau autoréférent ? En déduire une stratégie d'élagage pour accélérer la recherche. *Indication : utiliser la fonction* `somme` *de la première question pour interrompre par un échec l'exploration lorsque* `somme t i` *dépasse déjà la valeur maximale possible.*
5. Que peut-on dire si juste après avoir affecté la case $t.(i)$, il y a déjà strictement plus d'occurrences d'une valeur $0 \leq k \leq i$ que la valeur de $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 15$?
6. Après avoir affecté la case $t.(i)$, combien de cases reste-t-il à remplir ? Combien de ces cases seront complétées par une valeur non nulle ? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin ? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 30$?
7. Montrer qu'il existe un tableau autoréférent pour tout $n \geq 7$. *On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de $n \geq 7$. On ne demande pas de montrer que cette solution est unique.*

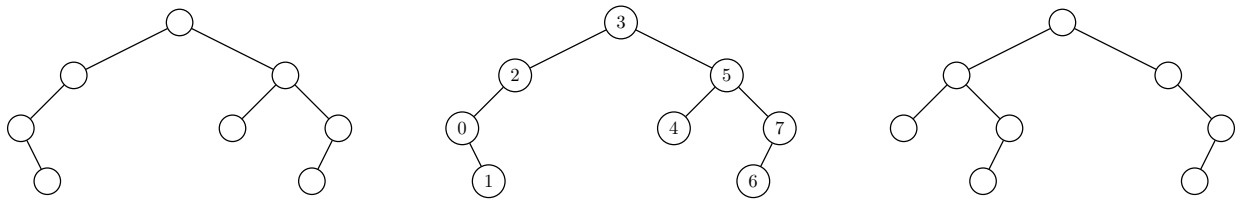
Exercice 2. L'exercice suivant est à traiter dans le langage OCaml.

Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$. Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit :

```
type arbre =
  | V
  | N of arbre * int * arbre
```

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à n nœuds par $[\![0; n-1]\!]$ en suivant l'*ordre infixe* de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par *ordre préfixe*. La figure 1 propose un exemple (on ne dessine pas les arbres vides).



(a) (b) (c)

FIGURE 1 – (a) un arbre binaire non étiqueté ; (b) son étiquetage en suivant un ordre infixe, la permutation associée est `[3; 2; 0; 1; 5; 4; 7; 6]` ; (c) un autre arbre binaire non étiqueté.

Un fichier source OCaml qui implémente ces exemples vous est fourni.

1. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
2. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur. On pourra utiliser l'opérateur `@` et on ne cherchera pas nécessairement à proposer une solution linéaire en la taille de l'arbre.
3. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur. *Indication : on pourra utiliser une fonction auxiliaire de type* `arbre -> int -> arbre * int` *qui prend en paramètres un arbre et la prochaine étiquette à mettre et qui renvoie le couple formé par l'arbre étiqueté et la nouvelle prochaine étiquette à mettre.*

Une permutation σ de $[\![0, n-1]\!]$ est *triable avec une pile* s'il est possible de trier la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ en utilisant uniquement une structure de pile comme espace de stockage interne. On considère l'algorithme suivant, énoncé ici dans un style impératif :

- Initialiser une pile vide ;
- Pour chaque élément en entrée :
  - ⋆ Tant que l'élément est plus grand que le sommet de la pile, dépiler le sommet de la pile vers la sortie ;
  - ⋆ Empiler l'élément en entrée dans la pile ;
- Dépiler tous les éléments restant dans la pile vers la sortie.

Par exemple, pour la permutation `[3; 2; 0; 1; 5; 4; 7; 6]`, on empile 3, 2, 0, on dépile 0, on empile 1, on dépile 1, 2, 3, on empile 5, 4, on dépile 4, 5, on empile 7, 6, on dépile 6, 7. On obtient la liste triée `[7; 6; 5; 4; 3; 2; 1; 0]` en supposant avoir ajouté en sortie les éléments dans une liste. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.

4. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure 1 et vérifier qu'elle est bien triable par pile.
5. Écrire une fonction `trier : int list -> int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, `trier [3; 2; 0; 1; 5; 4; 7; 6]` doit s'évaluer en la liste `[7; 6; 5; 4; 3; 2; 1; 0]`. On utilisera directement une liste pour implémenter une pile. *Indication : écrire une fonction auxiliaire de*

type `int list -> int list -> int list -> int list` *qui prend en paramètre une liste d'entrée, une pile et une liste de sortie, et qui, en fonction de la forme de la liste d'entrée et de la pile, applique une étape élémentaire avant de procèder récursivement.*

6. Montrer que s'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$, alors σ n'est pas triable par une pile.
7. On se propose de montrer que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n nœuds.
   (a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.
   (b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire. *Indication : on peut prendre σ_0 comme racine, puis procéder récursivement avec les $\sigma_0 - 1$ éléments pour construire le fils gauche et avec le reste pour le fils droit.*

Exercice 3. L'exercice suivant est à traiter dans le langage OCaml.

On s'intéresse au problème SAT pour une formule en forme normale conjonctive. On se fixe un ensemble fini $\mathcal{V} = \{v_0, v_1, \ldots, v_{n-1}\}$ de variables propositionnelles.

Un *littéral* ℓ_i est une variable propositionnelle v_i ou la négation d'une variable propositionnelle $\neg v_i$. On représente un littéral en OCaml par un type énuméré : le littéral v_i est représenté par `V i` et le littéral $\neg v_i$ par `NV i`. Une *clause* $c = \ell_0 \vee \ell_1 \vee \cdots \vee \ell_{|c|-1}$ est une disjonction de littéraux, que l'on représente en OCaml par un tableau de littéraux. On ne considérera dans cet exercice que des formules sous forme normale conjonctive, c'est-à-dire sous forme de conjonctions de clauses $c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$. On représente une telle formule en OCaml par une liste de clauses, soit une liste de tableaux de littéraux. On n'impose rien sur les clauses : une clause peut être vide et un même littéral peut s'y trouver plusieurs fois. De même une formule peut n'être formée d'aucune clause, elle est alors notée $\top$ et est considérée comme une tautologie. Une *valuation* $v : \mathcal{V} \to \{V, F\}$ est représentée en OCaml par un tableau de booléens.

Un programme OCaml à compléter vous est fourni. La fonction `initialise : int -> valuation` permet d'initialiser une valuation aléatoire.

1. Implémenter la fonction `evalue : clause -> valuation -> bool` qui vérifie si une clause est satisfaite par une valuation.

Étant donné une formule f constituée de m clauses $c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$ définies sur un ensemble de n variables, la fonction `random_sat` a pour objectif de trouver une valuation qui satisfait la formule, s'il en existe une et qu'elle y arrive. Cette fonction doit effectuer au plus k tentatives et renvoyer un résultat de type `valuation option`, avec une valuation qui satisfait la formule passée en paramètre si elle en trouve une et la valeur `None` sinon.

L'idée de ce programme est d'effectuer une assignation aléatoire des variables puis de vérifier que chaque clause est satisfaite. Si une clause n'est pas satisfaite, on modifie aléatoirement la valeur associée à un littéral de cette clause, qui devient ainsi satisfaite, puis on recommence.

2. Si ce programme renvoie `None`, peut-on conclure que la formule f en entrée n'est pas satisfiable ? De quel type d'algorithme probabiliste s'agit-il ?
3. Proposer un jeu de 5 tests élémentaires permettant de tester la correction de ce programme.
4. Ce programme est-il correct par rapport à sa spécification ? Si cela s'avère nécessaire, corriger ce programme pour qu'il remplisse ses objectifs.

On s'intéresse maintenant au problème MAX-SAT qui consiste, toujours pour une formule en forme normale conjonctive comme ci-dessus, à trouver le plus grand nombre de clauses de cette formule simultanément satisfiables. Un algorithme d'approximation probabiliste naïf pour obtenir une solution approchée consiste à générer aléatoirement k valuations et retenir celle qui maximise le nombre de clauses satisfaites.

5. Implémenter cette approche en OCaml et vérifier sur quelques exemples. Quelle est sa complexité dans le meilleur et dans le pire cas ?
6. Sous l'hypothèse P $\neq$ NP, peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT ? Justifier.

Exercice 4. L'exercice suivant est à traiter dans le langage C.

Dans tout l'exercice, on ne considère que des tableaux d'entiers de longueur $n \geq 0$.

Un squelette de programme C vous est donné, avec un jeu de tests qu'il *ne faut pas modifier*. Vous pouvez bien sûr ajouter vos propres tests à part.

1. Écrire une fonction de prototype `bool nb_occurrences(int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément `x` dans le tableau `tab` de longueur `n`. Quelle est la complexité de cette fonction ?

Dans toute la suite, on suppose que les tableaux sont *triés* dans l'ordre croissant. On va chercher à écrire une version plus efficace de la fonction ci-dessus qui exploite cette propriété. On cherche tout d'abord à écrire une fonction `int une_occurrence(int n, int* tab, int x)` qui permet de renvoyer un indice d'une occurrence quelconque de l'élément `x` s'il est présent dans le tableau et `-1` sinon. On procède par dichotomie.

2. Compléter le code de la fonction `int une_occurrence(int n, int* tab, int x)` qui vous est donnée dans le squelette. Cette fonction doit avoir une complexité en $O(\log n)$.
3. Écrire une fonction `int premiere_occurrence(int n, int* tab, int x)` qui renvoie l'indice de la première occurrence d'un élément `x` dans un tableau `tab` de longueur `n` si cet élément est présent et -1 sinon. Cette fonction doit avoir une complexité en $O(\log n)$.
4. Écrire une fonction `int nombre_occurrences(int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément `x` dans le tableau `tab` de longueur `n`. Cette fonction devra avoir une complexité en $O(\log n)$.
5. Justifier que la fonction `une_occurrence` termine et est correcte. On donnera un variant et un invariant de boucle que l'on justifiera.
6. Montrer que la complexité de la fonction `une_occurrence` est bien en $O(\log n)$.

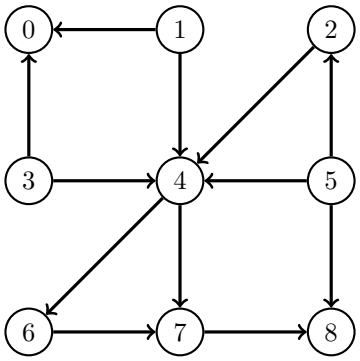
Exercice 5. L'exercice suivant est à traiter dans le langage C.

Dans cet exercice, tous les graphes seront orientés. On représente un graphe orienté $G = (S, A)$, avec $S = \{0, \dots, n-1\}$, en C par la structure suivante :

```
struct graph_s {
  int n;
  int degre[100];
  int voisins[100][10];
};
```

L'entier `n` correspond au nombre de sommets $|S|$ du graphe. On suppose que $n \leq 100$. Pour $0 \leq s < n$, la case `degre[s]` contient le degré sortant $d^+(s)$, c'est-à-dire le nombre de successeurs, appelés ici *voisins*, de s. On suppose que ce degré est toujours inférieur à 10. Pour $0 \leq s < n$, la case `voisins[s]` est un tableau contenant, aux indices $0 \leq i < d^+(s)$, les voisins du sommet s. Il s'agit donc d'une représentation par listes d'adjacences où les listes sont représentées par des tableaux en C.

Un programme en C vous est fourni dans lequel le graphe suivant est représenté par la variable `g_exemple`.



Pour $s \in S$ on note $\mathcal{A}(s)$ l'ensemble des sommets accessibles à partir de s. Pour $s \in S$, le maximum des degrés d'un sommet accessible à partir de s est noté $d^*(s) = \max\{d^+(s') \mid s' \in \mathcal{A}(s)\}$. Par exemple, pour le graphe ci-dessus, $\mathcal{A}(2) = \{2, 4, 6, 7, 8\}$ et $d^*(2) = 2$ car $d^+(4) = 2$. Dans cet exercice, on cherche à calculer $d^*(s)$ pour chaque sommet $s \in S$.

On représente un sous-ensemble de sommets $S' \subseteq S$ par un tableau de booléens de taille n, contenant `true` à la case d'indice s' si $s' \in \mathcal{A}(s)$ et `false` sinon.

1. Écrire une fonction `int degre_max(graph* g, bool* partie)` qui calcule le degré maximal d'un sommet $s' \in S'$ dans un graphe $G = (S, A)$ pour une partie $S' \subseteq S$ représentée par S, c'est-à-dire qui calcule $\max\{d^+(s') \mid s' \in S'\}$.
2. Écrire une fonction `bool* accessibles(graph* g, int s)` qui prend en paramètre un graphe et un sommet s et qui renvoie un (pointeur sur un) tableau de booléens de taille n représentant $\mathcal{A}(s)$. Une fonction `nb_accessibles` qui utilise votre fonction et un test pour l'exemple ci-dessus vous sont donnés dans le fichier à compléter.
3. Écrire une fonction `int degre_etoile(graph* g, int s)` qui calcule $d^*(s)$ pour un graphe et un sommet passé en paramètre. Quelle est la complexité de votre approche ?
4. Linéariser le graphe donné en exemple ci-dessus, c'est-à-dire représenter ses sommets sur une même ligne dans l'ordre donné par un tri topologique, tous les arcs allant de gauche à droite.
5. Dans cette question, on suppose que le graphe $G = (S, A)$ est acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $O(|S| + |A|)$.
6. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $O(|S| + |A|)$.

Exercice 6. L'exercice suivant est à traiter dans le langage C.

On dispose de $n \geq 1$ objets $\{o_0, \ldots, o_{n-1}\}$ de valeurs respectives $(v_0, ..., v_{n-1}) \in \mathbb{N}^n$ et de poids respectifs $(p_0, ..., p_{n-1}) \in \mathbb{N}^n$. On souhaite transporter dans un sac de poids maximum $p_{\max}$ un sous-ensemble d'objets ayant la plus grande valeur possible. Formellement, on souhaite donc maximiser

$$\sum_{i=0}^{n-1} x_i v_i$$

sous les contraintes

$$(x_0, ..., x_{n-1}) \in \{0,1\}^n \quad \text{et} \quad \sum_{i=0}^{n-1} x_i p_i \leq p_{\max}$$

Intuitivement, la variable x_i vaut 1 si l'objet o_i est mis dans le sac et 0 sinon.

On propose d'utiliser un algorithme glouton dont le principe est de considèrer les objets $o_0, o_1, \ldots, o_{n-1}$ dans l'ordre et de choisir à l'étape i l'objet i (donc poser $x_i = 1$) si celui-ci rentre dans le sac avec la contrainte de poids maximal respectée et ne pas le choisir (donc poser $x_i = 0$) sinon. *On remarque que les valeurs $v_0, v_1, \ldots, v_{n-1}$ ne sont pas directement utilisées par cet algorithme, elle le seront lors du tri éventuel des objets.*

1. Proposer un type de données pour implémenter, pour n objets, leurs valeurs, leurs poids et les indicateurs $x_0, x_1, \ldots, x_{n-1}$.
2. Écrire une fonction qui implémente la méthode gloutonne décrite ci-dessus à partir de n, $p_{\max}$ et $p_0, p_1, \ldots, p_{n-1}$ et qui permet de renvoyer les indicateurs $x_0, x_1, \ldots, x_{n-1}$ pour le choix glouton.
3. Écrire un programme complet qui permet de lire sur l'entrée standard (au clavier par défaut) un entier $n \geq 1$, puis un entier naturel $p_{\max}$, puis n entiers naturels correspondant aux valeurs $v_0, v_1, \ldots, v_{n-1}$, puis n entiers naturels correspondant aux poids $p_0, p_1, \ldots, p_{n-1}$ et qui affiche sur la sortie standard (l'écran par défaut) sous une forme de votre choix les indicateurs $x_0, x_1, \ldots, x_{n-1}$, la valeur de la solution $\sum_{i=0}^{n-1} x_i v_i$ et le poids utilisé $\sum_{i=0}^{n-1} x_i p_i$. On rappelle que le spécificateur de format pour lire ou écrire un entier est `%d`.
4. L'algorithme glouton ci-dessus donne-t-il toujours une solution optimale ?
   (a) si on ne suppose rien sur l'ordre des objets a priori ;
   (b) si les objets sont triés par ordre de valeur décroissante ;
   (c) si les objets sont triés par ordre de poids croissant ;
   (d) si les objets sont triés par ordre décroissant des quotients $\frac{v_i}{p_i}$.

   Justifier à chaque fois votre réponse à l'aide d'un contre-exemple ou d'une démonstration.
5. Le problème du sac à dos étudié dans cet exercice est un problème d'optimisation. Donner le problème de décision associé en utilisant une valeur seuil v_{seuil}. Montrer que ce problème de décision est dans la classe NP.
6. Quelle serait la complexité d'une méthode qui examinerait tous les choix possibles pour retenir le meilleur ? Quelles stratégies d'élagage pourrait-on mettre en œuvre pour réduire l'espace de recherche ?

On peut montrer que ce problème de décision est NP-complet.