Entraînements CCINP

Ce document contient l'intégralité des exercices mis à disposition par le CCINP (exercices 0 et une sélection d'exercices postés en 2023 et 2024). L'origine de chaque exercice est indiquée dans son titre. Attention, les modalités et le format évoluent légèrement au fil des ans. Pour savoir quelle forme auront les énoncés en 2025, il faut donc plus se fier aux exercices 2024.

Les exercices A commencent en page 15.

Exercices de type B

Exercice 1 *HORNSAT (exo 2024)*

Cet énoncé est accompagné d'un ou plusieurs codes compagnons en OCaml fournissant certaines des fonctions mentionnées dans l'énoncé : ils sont à compléter en y implémentant les fonctions demandées.On attend un style de programmation fonctionnel. L'utilisation des fonctions du module `List` *est autorisée ; celle des fonctions du module* `Option` *est interdite.*

Une formule du calcul propositionnel est une *formule de Horn* s'il s'agit d'une formule sous forme normale conjonctive (FNC) dans laquelle chaque clause (éventuellement vide, auquel cas la clause en question est la disjonction d'un ensemble vide de littéraux et est donc sémantiquement équivalente à $\bot$) contient au plus un littéral positif. Dans la suite, on considère qu'une clause d'une telle formule contient au plus une occurrence de chaque variable (en particulier, les clauses sont sans doublons).

1. Les formules suivantes sont-elles des formules de Horn ?
   a) $F_1 = (\neg x_0 \vee \neg x_1 \vee \neg x_3) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee x_0 \vee \neg x_3) \wedge (\neg x_0 \vee \neg x_3 \vee x_2) \wedge x_2 \wedge (\neg x_3 \vee \neg x_2)$.
   b) $F_2 = (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_0) \wedge \neg x_1 \wedge (x_1 \vee \neg x_1 \vee x_0) \wedge (\neg x_0 \vee x_2)$.
   c) $F_3 = (\neg x_1 \vee \neg x_4) \wedge x_1 \wedge (\neg x_0 \vee \neg x_3 \vee \neg x_4) \wedge (x_0 \vee \neg x_1) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_0 \vee \neg x_1)$.

On utilise le type suivant pour manipuler les formules de Horn : une formule de Horn est une liste de clauses de Horn ; une clause étant la donnée d'un `int option` valant `None` si la clause ne contient pas de littéral positif et `Some i` si x_i en est l'unique littéral positif et d'une liste d'entiers correspondants aux numéros des variables intervenant dans les littéraux négatifs.

```
type clause_horn = int option * int list
type formule_horn = clause_horn list
```

2. Écrire une fonction `avoir_clause_vide : formule_horn -> bool` qui renvoie `true` si et seulement si la formule en entrée contient une clause vide (donc ne contenant ni littéral positif, ni aucun littéral négatif).

On appelle *clause unitaire* une clause réduite à un littéral positif. Par ailleurs, *propager* une variable x_i dans une formule F sous FNC consiste à modifier F comme suit :

- Toute clause de F qui ne fait pas intervenir la variable x_i est conservée telle quelle.
- Toute clause de F qui fait intervenir le littéral x_i est supprimée entièrement.
- On supprime le littéral $\neg x_i$ de toutes les clauses de F qui font intervenir ce littéral.

On souligne que supprimer $\neg x$ d'une clause C qui ne fait intervenir que ce littéral ne revient pas à supprimer la clause C. On s'intéresse à l'algorithme $\mathcal{A}$ suivant dont on admet (pour le moment) qu'il permet de déterminer si une formule de Horn F est satisfiable :

tant que il y a une clause unitaire x_i dans F
  $F \leftarrow$ propager x_i dans F
si F contient une clause vide **alors**
  renvoyer faux
sinon
  renvoyer vrai

3. À l'aide de cet algorithme déterminer si les formules de Horn de la question 1 sont satisfiables. On utilisera ces formules pour tester les fonctions implémentées aux questions suivantes.

4. Écrire une fonction `trouver_clause_unitaire : formule_horn -> int option` renvoyant `None` si la formule en entrée n'a pas de clause unitaire et `Some i` où x_i est l'une des clauses unitaires sinon.

5. Justifier que propager une variable dans une formule de Horn donne une formule de Horn. Écrire une fonction `propager : formule_horn -> int -> formule_horn` qui prend en entrée une formule de Horn F et un entier i et calcule la formule résultat de la propagation de x_i dans F.

6. Déduire des questions précédentes une fonction `etre_satisfiable : formule_horn -> bool` renvoyant `true` si et seulement si la formule de Horn en entrée est satisfiable.

7. Quelle est la complexité de votre algorithme en fonction de la taille de la formule en entrée ? Que peut-on dire des problèmes de décision `SAT` et `HORN-SAT` (dont la définition est la même que celle de `SAT` à ceci près que les formules considérées sont supposées être des formules de Horn) ?

8. On s'intéresse à présent à la correction de l'algorithme $\mathcal{A}$.

   a) Si F est une clause de Horn sans clause unitaire ni clause vide, donner une valuation simple qui satisfait F.

   b) On admet que si F est une formule de Horn faisant intervenir une clause unitaire x_i et F' est le résultat de la propagation de x_i dans F, alors que F est satisfiable si et seulement si F' est satisfiable. En déduire la correction de l'algorithme $\mathcal{A}$.

9. Expliquer comment on pourrait modifier les fonctions précédentes afin de déterminer une valuation satisfaisant une formule de Horn dans le cas où elle existe plutôt que de juste dire si elle est satisfiable ou non. On ne demande pas d'implémentation.

Exercice 2 *Mots de Dyck (exo 2024)*

Cet énoncé est accompagné d'un ou plusieurs codes compagnons en C fournissant certaines des fonctions mentionnées dans l'énoncé : il sont à compléter en y implémentant les fonctions demandées.

La ligne de compilation `gcc -o main.exe -Wall *.c -lm` *vous permet de créer un exécutable* `main.exe` *à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire* `make`*. En ligne de commande, il suffit d'écrire* `make`*. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* `./main.exe`*.*

Il est possible d'activer davantage d'avertissements et un outil d'analyse de la gestion de la mémoire avec la ligne de compilation `gcc -o main.exe -g -Wall -Wextra -fsanitize=address *.c -lm` *ou en écrivant* `make safe`*. L'examinateur pourra vous demander de compiler avec ces options.*

Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant `make clean` *et relancer une compilation.*

La compilation du code compagnon initial avec `make safe` *provoque des warnings attendus qui seront résolus lors de l'implémentation des fonctions demandées par le sujet.*

On s'intéresse dans cet exercice aux mots de Dyck, c'est-à-dire aux mots bien parenthésés. Dans ce type de mots, toute parenthèse ouverte "(" est fermée ")" et une parenthèse ne peut être fermée si elle ne correspond pas à une parenthèse préalablement ouverte.

Par exemple pour deux couples de parenthèses, "(())" et "()()" sont des chaînes de parenthèses bien formées. "())(" et ")()(" ne le sont pas.

On admet que le nombre de mots bien parenthésés à n couples de parenthèses est donné par les nombres de Catalan définis par la formule suivante :

$$C_n = \frac{(2n)!}{(n+1)!n!} \text{ pour } n \geq 0$$

On rappelle que le type `uint64_t` est un type entier non signé codé sur 64 bits.

1. Complétez dans le code compagnon la fonction dont le prototype est `uint64_t catalan(int n)`. Vous pouvez utiliser une fonction auxiliaire si cela vous semble pertinent.
2. Que va-t-il se passer si on tente d'afficher `catalan(n)` pour `n` un peu grand ? Le constatez-vous ici ?

On cherche maintenant à afficher le nombre de mots (chaînes) bien parenthésés avec n fixé couples de parenthèses, ainsi que les mots eux-mêmes.

Un algorithme de force brute pour déterminer toutes les chaînes à n couples de parenthèses bien formées consiste à générer toutes les possibilités puis à ne garder que les chaînes bien formées.

3. Complétez dans le code compagnon la fonction dont le prototype est `bool verification(char * mot)`. Cette fonction renvoie `true` si le mot fourni en paramètre `mot` est bien parenthésé, `false` sinon.
4. Quelle est la complexité de cette vérification ?
5. Quelle est la complexité finale de l'algorithme de force brute ?

On appelle `n` le nombre de couples de parenthèses voulu. Dans le fichier compagnon fourni, le nombre de couples a été limité à 18.

On vous propose de coder l'énumération des chaînes de parenthèses bien formées en appliquant l'algorithme de backtracking suivant, dont on admet qu'il est correct : on compte le nombre de parenthèses ouvertes `o` et le nombre de parenthèses fermées `f` dans une chaîne de caractères courante (vide au départ).

- Si `o = f = n`, on a trouvé une chaîne bien formée.
- Si `o < n`, on ajoute une parenthèse ouvrante et on relance.
- Si `f < o`, on ajoute une parenthèse fermante et on relance.

Cet algorithme est à implémenter dans la fonction dont le prototype est `void dyck(char s[N], int o, int f, int n)` qui affiche sur la sortie standard les chaînes de parenthèses bien formées avec `n` couples de parenthèses lorsque `s` est la chaîne de caractère courante, `o` est son nombre de parenthèses ouvrantes et `f` est son nombre de parenthèses fermantes.

6. Compléter la fonction `dyck` pour afficher les chaînes bien parenthésées avec 5 couples de parenthèses.
7. Adapter la fonction `dyck` pour calculer le nombre de mots obtenus. Combien de mots trouvez-vous pour 16 couples de parenthèses ?
8. Adapter la fonction `dyck` pour stocker les mots bien parenthésés dans une liste chaînée et les afficher après l'appel à la fonction. Vous trouverez dans le code compagnon une structure qui peut vous aider.

Exercice 3 *Chemins simples sans issue (exo 2023)*

Consignes : Cet exercice est à traiter en OCaml. Le fichier chemins_simples.ml *est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin, et on donne leur représentation en OCaml.

Un *graphe orienté* est un couple (V, E) où V est un ensemble fini (ensemble des sommets), E est un sous-ensemble de $V \times V$ où tout élément $(v_1, v_2) \in E$ vérifie $v_1 \neq v_2$ (ensemble des arcs).

Étant donné un graphe $G = (V, E)$ un *chemin non vide* de G est une suite finie $s_0, \cdots, s_n$ de sommets de V avec $n \geq 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout s_{n+1} sommet tel que $(s_n, s_{n+1}) \in E$, s_{n+1} appartient à $\{s_0, \cdots, s_n\}$.

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on utilise le type suivant :

```
type graphe = int list array
```

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0, 1, 2, 3\}, \{(0, 1), (0, 3), (2, 0), (2, 1), (2, 3), (3, 1)\})$$

est représenté par le tableau `[|[1;3];[];[0;1;3];[1]|]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[|[];[0];[0;3;1];[1]|]` représente le graphe

$$G_2 = (\{0, 1, 2, 3\}, \{(1, 0), (2, 0), (2, 1), (2, 3), (3, 1)\})$$

On rappelle différentes fonctions pouvant être utiles :

- `List.filter : ('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter :('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0);(f a1); ...;(f an)` dans le cas où on a `l = a0::a1::...::an::[]`.
- `List.rev : 'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3;1;2;2;4]` est égal à `[4;2;2;1;3]`.
- `Array.length : 'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier chemins_simples.ml. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

1. Écrire une fonction `est_sommet : graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.
2. Écrire une fonction `appartient : 'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.
3. Écrire une fonction `est_chemin : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.
4. Compléter la fonction `est_chemin_simple_sans_issue : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.

5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier chemins_simples.ml et qui permet de générer la liste des chemins simples sans issue d'un graphe.

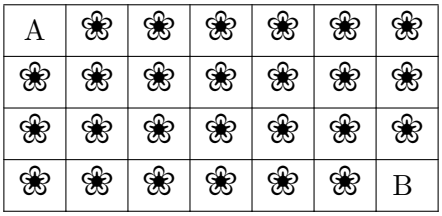
6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2.

7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

Exercice 4 *Récolte dynamique de fleurs (exo 2023)*

*Consignes : Cet énoncé est accompagné d'un code compagnon en C bouquet_enonce.c fournissant certaines des fonctions mentionnées dans l'énoncé : il est à compléter en y implémentant les fonctions demandées. La ligne de compilation gcc -o main.exe *.c -lm vous permet de créer un exécutable main.exe à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire make. En ligne de commande, il suffit de taper make. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande ./main.exe. Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant make clean et relancer make.*

Une petite fille se trouve en haut à gauche (case A) d'un champ modélisé par un tableau rectangulaire de taille $m \times n$ et doit se rendre dans la case B en bas à droite du champ où réside sa grand-mère (figure ci-dessous).



Chaque case du tableau, *y compris les cases A et B*, contient un certain nombre de fleurs. La petite fille, qui connaît depuis sa position initiale le nombre de fleurs de chaque case, doit se déplacer vers B de case en case, les seuls mouvements autorisés étant vers le bas ou vers la droite. À chaque déplacement, elle récolte les fleurs de la case atteinte. L'objectif pour elle est alors de faire le bouquet avec le plus de fleurs possible lors de son déplacement pour l'offrir à sa grand-mère.

1. On considère le champ suivant :

0 (A)	1	2	3
1	2	3	4
2	3	4	0
3	4	0	1 (B)

   Donner le nombre maximal de fleurs cueillies par la petite fille.

2. On note $n(i, j)$ le nombre maximum de fleurs que la petite fille peut récolter en se déplaçant de A à la case (i, j). Exprimer $n(i, j)$ en fonction de $n(i-1, j)$ et $n(i, j-1)$. En déduire une fonction récursive de prototype `int recolte(int champ[m][n], int i, int j)` qui, étant données les coordonnées `i`, `j` d'une case, calcule le nombre maximum de fleurs cueillies par la petite fille de A à la case (`i`, `j`).

3. On suppose $m = n = 4$ et on effectue donc un appel à `recolte(champ,3,3)` pour résoudre le problème posé. Donner le nombre de fois où votre fonction calcule le nombre de fleurs maximum cueillies dans la case $(1, 1)$ (deuxième case de la diagonale).

D'une manière générale, le nombre d'appels à la fonction récursive est important. On a donc intérêt à transformer l'algorithme récursif en algorithme dynamique. On propose de déclarer dans le programme principal un tableau `fleurs` dont la case (i, j) est destinée à contenir la récolte maximale que la petite fille peut obtenir en cheminant de A vers la case (i, j).

4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?
5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et retourne la cueillette maximale obtenue en parcourant le champ de A à la case (`i`, `j`).

La fonction `recolte_iterative` permet de déterminer la cueillette maximale en (`i`, `j`) mais ne précise pas le chemin parcouru pour l'obtenir.

6. Écrire la fonction de prototype `void deplacements(int fleurs[m][n], int i, int j)` qui affiche la suite des déplacements effectués par la petite fille sur un chemin permettant de récolter le nombre maximum de fleurs entre (0,0) et (`i`, `j`).
7. Insérer un appel de `deplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

Exercice 5 *Tri par pile (exo 0)*

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...).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entiers $[\sigma_0; \sigma_1; ...; \sigma_{n-1}]$. Un arbre binaire étiqueté est soit un arbre vide, soit un noeud 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 noeuds 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).

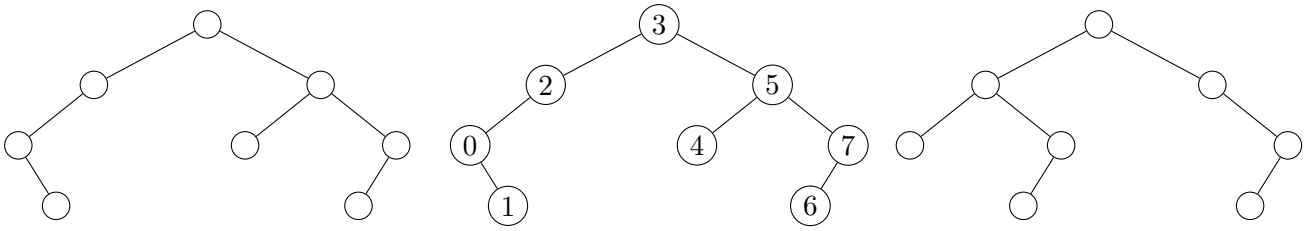


FIG. 1 : à gauche, un arbre binaire non étiqueté, au milieu, son étiquetage en suivant un ordre infixe, la permutation associée est $[3; 2; 0; 1; 5; 4; 7; 6]$; à droite, un autre arbre binaire non étiqueté

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

1. Étiqueter l'arbre de droite 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 $[\sigma_0; \sigma_1; ...; \sigma_{n-1}]$ 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 restants 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 alors 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 si cet algorithme permet de la trier correctement.

4. Dérouler l'exécution de l'algorithme sur la permutation obtenue en question 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 procéder récursivment.*
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 noeuds.
   (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 6 *Maximum des degrés d'un graphe (exo 0)*

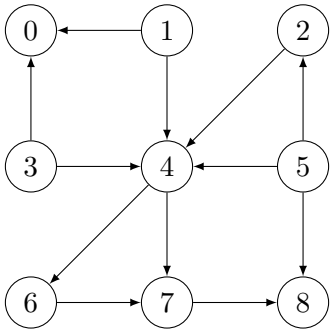
L'exercice suivant est à traiter dans le langage C.

Dans cet exercice, tous les graphes seront orientés. On représente un graphe orienté $G = (S, A)$ avec $S = \{0, ..., n-1\}$ 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'adjacence où les listes sont représentées par des tableaux en C.

Un programme C vous est fourni dans lequel le graphe suivant est representé 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 $S' \subset S$ par un tableau de booléens de taille n contentant `true` à la case d'indice s' si $s' \in S'$ 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 $S' \subset S$ 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 7 *Sac à dos (exo 0)*

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

On dispose de $n \geq 1$ objets $\{o_0, ..., o_{n-1}\}$ 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

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

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 considérer les objets $o_0, o_1, ...o_{n-1}$ 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 $v_0, v_1, ..., v_{n-1}$ 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 $x_0, x_1, ...x_{n-1}$.
2. Écrire une fonction qui implémente la méthode gloutonne décrite ci-dessus à partir de n, p_{max} et $p_0, p_1, ..., p_{n-1}$ et qui permet de renvoyer les indicateurs $x_0, x_1, .., x_{n-1}$ 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 $v_0, v_1, ..., v_{n-1}$, puis n entiers naturels correspondant aux poids $p_0, p_1, ..., p_{n-1}$ et qui affiche sur la sortie standard (l'écran par défaut) sous une forme de votre choix les indicateurs $x_0, x_1, ..., x_{n-1}$, 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 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 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 oeuvre pour réduire l'espace de recherche ?

Exercice 8 *Tableaux autoréférents (exo 0)*

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

1. Écrire une fonction `somme : int array -> int -> int` telle que l'appel à `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 :

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

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 autoréférents 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 $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en oeuvre. 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 oeuvre. 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 9 *Calcul avec les flottants (exo 2023)*

*Consignes : Cet énoncé est accompagné d'un code compagnon en C, flottants.c fournissant les structures de données et certaines fonctions mentionnées dans l'énoncé. Il est à compléter en y implémentant les fonctions demandées. La ligne de compilation gcc -o main.exe *.c -lm vous permet de créer un exécutable main.exe. Vous pouvez également utiliser l'utilitaire make. En ligne de commande, il suffit de taper make. Dans les*

deux cas, si la compilation réussit, le programme peut être exécuté avec la commande ./main.exe. Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant make clean et relancer make.

Un nombre réel x est représenté en machine en base 2 par un flottant qui a un signe s, une mantisse m et un exposant e tel que $x = s \times m \times 2^e$. Dans la norme IEEE 754, en convention normalisée la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits.

Dans cet exercice, on observe le résultat de calculs obtenus par un programme. On pourra utiliser la fonction de signature : `double pow(double v, double p)` qui calcule v^p.

1. Dans la fonction principale `main`, on a défini 3 variables a, b, c de type `double`. Compléter le code pour calculer et afficher le résultat des opérations $(a+b)+c$ et $a+(b+c)$. Que constatez-vous ?

2. Compte tenu des approximations faites lors du codage, on peut trouver plusieurs nombres x tels que $1+x=1$ après un calcul fait par la machine. Le plus petit nombre représentable exactement en machine et supérieur à 1 s'écrit $1+\epsilon$, avec ϵ un réel appelé ϵ machine. On admet que ϵ s'écrit sous la forme 2^{-n} avec n un entier naturel strictement positif. Écrire une fonction de signature `double epsilon()` qui renvoie la valeur de n. Justifier cette valeur.

3. On considère une suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}\times u_{n-2}} \text{ si } n \geq 2 \end{cases}$$

   Écrire une fonction de signature `double u(int n)` qui renvoie la valeur du terme u_n.

4. La limite théorique de la suite $(u_n)_{n\in\mathbb{N}}$ est 6. Compléter la fonction `main` afin d'afficher les 22 premiers termes de la suite. Vers quelle valeur semble tendre la suite ?

5. On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un double et un pointeur vers une structure `nb` définie ci-dessous

```
struct nb {double x; struct nb* suivant;};
```

   Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.

6. L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul d'une somme.

> **Entrée :** Une liste l de réels triée dans l'ordre croissant de taille au moins 2.
> **Sortie :** La somme des réels contenus dans la liste l.
> **tant que** la liste l contient strictement plus d'un élément
> | Calculer la somme $s = x + y$ des deux premiers éléments x et y de l
> | Supprimer x et y de l
> | Insérer s dans l de sorte à ce que l reste triée
> **renvoyer** l'unique élément de l

   - Compléter la fonction `somme2` qui implémente cet algorithme.
   - La fonction proposée ne prend pas en compte un cas d'insertion. Illustrer ce propos.

Exercice 10 *Langages locaux (exo 2023)*

Consignes : Cet énoncé est accompagné d'un code compagnon en OCaml localite.ml *fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires : il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.*

On considère un alphabet Σ. Si L est un langage sur Σ, on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des premières lettres des mots de L.
- $D(L) = \{a \in \Sigma \mid \Sigma^\star a \cap L \neq \emptyset\}$ l'ensemble des dernières lettres des mots de L.
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^\star m \Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des facteurs de longueur 2 des mots de L.
- $N(L) = \Sigma^2 \setminus F(L)$ l'ensemble des mots de taille 2 qui ne sont pas facteurs de mots de L.

On rappelle qu'un langage L est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$$

1. Calculer les ensembles $P(L), D(L), F(L)$ et $N(L)$ dans le cas où L est le langage dénoté par l'expression régulière $a^\star(ab)^\star + aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

On cherche dans la suite de l'exercice à concevoir un algorithme répondant à la spécification suivante :

> **Entrée :** Une expression régulière e sur un alphabet Σ ne faisant pas intervenir le symbole $\emptyset$.
> **Sortie :** Vrai si le langage dénoté par e est local, faux sinon.

Par défaut, dans la suite de l'énoncé, "expression régulière" signifie "expression régulière ne faisant pas intervenir le symbole $\emptyset$". Les expressions régulières seront manipulées en OCaml via le type somme suivant :

```
type regexp =
  | Epsilon
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles $P(L)$, $D(L)$ et $F(L)$ à partir d'une expression régulière dénotant L. Ces ensembles seront représentés en OCaml par des listes de chaînes de caractères qui vérifieront les propriétés suivantes :

- Elles sont triées dans l'ordre croissant selon l'ordre lexicographique.
- Elles sont sans doublons.

L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles $P(L)$, $D(L)$ et $F(L)$. C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot ε, justifier brièvement la correction de `compute_P`.
3. Implémenter la fonction `contains_epsilon`.
4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.

5. Expliquer en langage naturel comment calculer récursivement l'ensemble $F(L)$ étant donnée une expression régulière e dénotant le langage L. Si $e = e_1e_2$ on pourra exprimer $F(L)$ en fonction notamment de $P(L_2)$ et $D(L_1)$ où L_1 (resp. L_2) est le langage dénoté par e_1 (resp. e_2).

6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

```
prod ["a";"c";"e"] ["b";"c"] = ["ab";"ac";"cb";"cc";"eb";"ec"]
```

7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble $F(L)$ étant donnée une expression régulière dénontant le langage L.

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ étant donnée une expression régulière dénotant L.

9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.

10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir $\emptyset$? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée ?

Exercice 11 *Nombre d'occurrences (exo 0)*

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

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

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 `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 de l'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 qu'on justifiera.

6. Montrer que la complexité de `une_occurrence` est bien en $O(\log n)$.

Exercice 12 *SAT (exo 0)*

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 $\mathcal{V} = \{v_0, v_1, ...v_{n-1}\}$ 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 $c = l_0 \vee l_1 \vee ... \vee l_{|c|-1}$ est une disjonction de littéraux, que l'on représente en OCaml par un tableau de littéraux. On ne considèrera dans cet exercice que des formules sous forme normale conjonctive, c'est-à-dire sous forme de conjonctions de clauses $c_0 \wedge c_1 \wedge ... \wedge c_{m-1}$. 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 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 $c_0 \wedge c_1 \wedge ... \wedge c_{m-1}$ 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 f 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 à 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 le pire cas ?
6. Sous l'hypothèse P $\neq$ NP, peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT ? Justifier.

Exercices de type A

Exercice 13 *Une relation d'équivalence (exo 2024)*

On fixe $n \in \mathbb{N}^\star$. Soient $\varphi : [\![1,n]\!] \to [\![1,n]\!]$ et $\psi : [\![1,n]\!] \to [\![1,n]\!]$. Soit u et v deux éléments de $[\![1,n]\!]$. On dit que u et v sont (φ,ψ)-équivalents s'il existe $k \in \mathbb{N}$, un tuple $(w_0, w_1, \cdots, w_{k+1}) \in [\![1,n]\!]^{k+2}$ avec $w_0 = u, w_{k+1} = v$ et vérifiant :

$$\forall i \in [\![0,k]\!], \varphi(w_i) = \varphi(w_{i+1}) \text{ ou } \psi(w_i) = \psi(w_{i+1}).$$

L'objectif est d'écrire un algorithme en pseudo-code permettant de calculer les différentes classes d'équivalence engendrées par cette relation.

1. Justifier rapidement que "être (φ,ψ)-équivalent" est une relation d'équivalence sur l'ensemble $[\![1,n]\!]$.
2. Pour cette question, on considère les applications φ et ψ définies par :

$i =$	1	2	3	4	5	6	7	8	9
$\varphi(i) =$	3	2	2	9	6	4	9	5	7
$\psi(i) =$	5	1	3	4	5	1	7	7	4

   Calculer les différentes classes d'équivalence.
3. On revient au cas au général. On définit le graphe $G = (S, A)$ par :

$$S = [\![1,n]\!], A = \{(x,y) \in S^2 | x \neq y \text{ et } (\varphi(x) = \varphi(y) \text{ ou } \psi(x) = \psi(y))\}.$$

   On fixe x et y deux sommets différents de S. Traduire sur le graphe G le fait que les sommets x et y sont (φ,ψ)-équivalents et en déduire que le calcul des classes d'équivalence de G se traduit en un problème classique sur les graphes que l'on précisera.
4. Donner en pseudo-code un algorithme permettant de résoudre le problème correspondant sur les graphes.

On fixe n un nombre pair. On considère deux applications φ et ψ de $[\![1,n]\!]$ où tout élément de l'image de φ admet exactement deux antécédents par φ et où tout élément de l'image de ψ admet exactement deux antécédents par ψ.

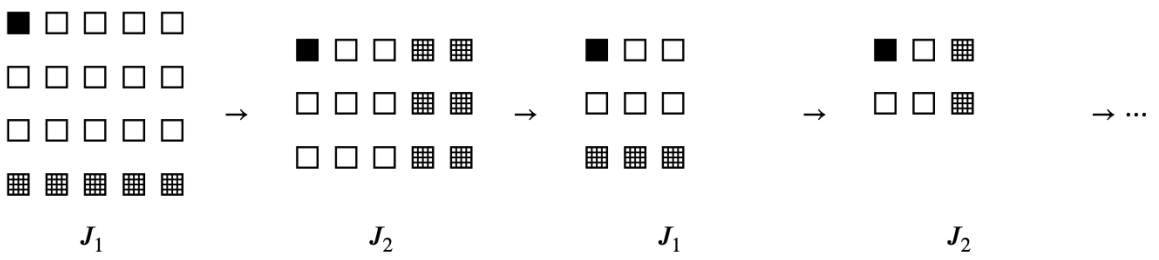
Pour $f \in \{\varphi,\psi\}$, on note G_f le graphe $(S, \{(x,y) \in S^2 | x \neq y \text{ et } f(x) = f(y)\})$.

5. Préciser la forme du graphe G_f pour $f \in \{\varphi,\psi\}$.
6. Expliciter la forme des différentes classes d'équivalence dans le graphe G correspondant.

Exercice 14 *Jeu de Chomp (exo 2024)*

On considère une variante du jeu de Chomp : deux joueurs J_1 et J_2 s'affrontent autour d'une tablette de chocolat de taille $l \times c$, dont le carré en haut à gauche est empoisonné. Les joueurs choisissent chacun leur tour une ou plusieurs lignes (ou une ou plusieurs colonnes) partant du bas (respectivement de la droite) et mangent les carrés correspondants. Il est interdit de manger le carré empoisonné et le perdant est le joueur qui ne peut plus jouer.

Dans la figure ci-dessous, matérialisant un début de partie sur une tablette de taille 4×5, le carré noir est le carré empoisonné, le choix du joueur J_i est d'abord matérialisé par des carrés hachurés, qui sont ensuite supprimés.



On associe à ce jeu un graphe orienté $G = (S, A)$. Les sommets S sont les états possibles de la tablette de chocolat, définis par un couple $s = (m, n), m \in [\![1, l]\!],\ n \in [\![1, c]\!]$. De plus, $(s_i, s_j) \in A$ si un des joueurs peut, par son choix de jeu, faire passer la tablette de l'état s_i à l'état s_j. On dit que s_j est un successeur de s_i et que s_i est un prédécesseur de s_j.

1. Dessiner le graphe G pour $l = 2$ et $c = 3$. Les états de G pourront être représentés par des dessins de tablettes plutôt que par des couples (m, n).

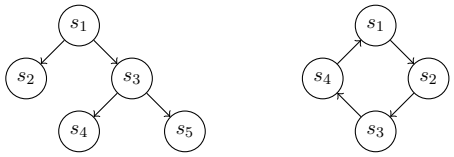
On va chercher à obtenir une stratégie gagnante pour le joueur J_1 par deux manières.

Utilisation des noyaux de graphe

Soit $G = (S, A)$ un graphe orienté. On dit que $N \subset S$ est un noyau de G si :

- pour tout sommet $s \in N$, les successeurs de s ne sont pas dans N,
- tout sommet $s \in S \setminus N$ possède au moins un successeur dans N

2. Donner tous les noyaux possibles pour les graphes suivants :



Dans la suite, on ne considère que des graphes acycliques.

3. Montrer que tout graphe acyclique admet un puits, c'est-à-dire un sommet sans successeur.

Dans le cas général, le noyau d'un graphe $G = (S, A)$ est souvent difficile à calculer. Si la dimension du jeu n'est pas trop importante, on peut toutefois le faire en utilisant l'algorithme suivant :

```
1 N = ∅
2 tant que il reste des sommets à traiter
3 |   Chercher un sommet s ∈ S sans successeur
4 |   N = N ⋃ {s}
5 |   Supprimer s de G ainsi que ses prédécesseurs
```

4. Justifier que cet algorithme termine et renvoie un noyau.
5. Démontrer que ce noyau est unique. Conclure que le graphe du jeu de Chomp possède un unique noyau N.
6. Appliquer cet algorithme pour calculer le noyau du jeu de Chomp à 2 lignes et 3 colonnes. Que peut-on dire du sommet (1,1) pour le joueur qui doit jouer ? En déduire à quoi correspondent les éléments du noyau.
7. Montrer que, dans le cas d'un graphe acyclique, tout joueur dont la position initiale n'est pas dans le noyau a une stratégie gagnante. Le joueur J_1 a-t-il une stratégie gagnante pour ce jeu dans le cas $l = 2$ et $c = 3$?

Utilisation des attracteurs

On modélise le jeu par un graphe biparti : pour ce faire, on dédouble les sommets du graphe précédent : un sommet s_i génère donc deux sommets s_i^1 et s_i^2, s_i^j étant le sommet i contrôlé par le joueur J_j. On forme alors deux ensembles de sommets $S_1 = \{s_i^1\}_i$ et $S_2 = \{s_i^2\}_i$, et on construit le graphe de jeu orienté $G = (S, A)$ avec $S = S_1 \bigcup S_2$ et $S_1 \bigcap S_2 = \emptyset$. De plus, $(s_i^1, s_j^2) \in A$ si le joueur 1 peut, par son choix de jeu, faire passer la tablette de l'état s_i^1 à l'état s_j^2. On raisonne de même pour $(s_i^2, s_j^1) \in A$.
On rappelle la définition d'un attracteur : soit F_1 l'ensemble des positions finales gagnantes pour J_1. On définit alors la suite d'ensembles $(\mathcal{A}_i)_{i\in\mathbb{N}}$ par récurrence : $\mathcal{A}_0 = F_1$ et

$$(\forall i \in \mathbb{N})\ \mathcal{A}_{i+1} = \mathcal{A}_i \cup \{s \in S_1 / \exists t \in \mathcal{A}_i, (s,t) \in A\} \cup \{s \in S_2 \text{ non terminal}, \forall t \in S, (s,t) \in A \Rightarrow t \in \mathcal{A}_i\}$$

et $\mathcal{A} = \bigcup_{i=0}^{\infty} \mathcal{A}_i$ est l'attracteur pour J_1.

8. Que représente l'ensemble $\mathcal{A}_i$?
9. Dans le cas du jeu de Chomp à deux lignes et trois colonnes (question 1), calculer les ensembles $\mathcal{A}_i$. Le joueur J_1 a-t-il une stratégie gagnante ? Comment le savoir à partir de $\mathcal{A}$?

Exercice 15 *Déduction naturelle (exo 0)*

Rappelons les règles de la déduction naturelle suivantes, où A et B sont des formules logiques et Γ un ensemble de formules logiques quelconque :

$$\frac{}{\Gamma, A \vdash A}\ \text{AX} \quad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B}\ \to_e \quad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg_e$$

1. Montrer que le séquent $\vdash \neg A \to (A \to \bot)$ est dérivable, en explicitant un arbre de preuve.
2. Montrer que le séquent $\vdash (A \to \bot) \to \neg A$ est dérivable, en explicitant un arbre de preuve.
3. Donner une règle correspondant à l'introduction du symbole $\wedge$ ainsi que deux règles correspondant à l'élimination du symbole $\wedge$. Montrer que le séquent $\vdash (\neg A \to (A \to \bot)) \wedge ((A \to \bot) \to \neg A)$ est dérivable.
4. On considère la formule $P = ((A \to B) \to A) \to A$ appelée *loi de Peirce*. Montrer que $\models P$, c'est-à-dire que P est une tautologie.
5. Pour montrer que le séquent $\vdash P$ est dérivable, il est nécessaire d'utiliser la règle d'absurdité classique $\bot_c$ (ou une règle équivalente), ce que l'on fait ci-dessous (il n'y aura pas besoin de réutiliser cette règle). Terminer la preuve du séquent $\vdash P$, dans laquelle on pose $\Gamma = \{(A \to B) \to A, \neg A\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{?}{\Gamma \vdash A}\ ? \quad \dfrac{}{\Gamma \vdash \neg A}\ \text{AX}}{\Gamma \vdash \bot}\ \neg_i}{(A \to B) \to A \vdash A}\ \bot_c}{\vdash ((A \to B) \to A) \to A}\ \to_i$$

Exercice 16 *SQL (exo 0)*

On considère le schéma de base de données suivant, qui décrit un ensemble de fabricants de matériel informatique, les matériels qu'ils vendent, leurs clients et ce qu'achètent leurs clients. Les attributs des clés primaires des six premières relations sont soulignés.

```
Production(NomFabricant, Modele)
Ordinateur(Modele, Frequence, Ram, Dd, Prix)
Portable(Modele, Frequence, Ram, Dd, Ecran, Prix)
Imprimante(Modele, Couleur, Type, Prix)
Fabricant(Nom, Adresse, NomPatron)
Client(Num, Nom, Prenom)
Achat(NumClient, NomFabricant, Modele, Quantite)
```

Chaque client possède un numéro unique connu de tous les fabricants. La relation `Production` donne pour chaque fabricant l'ensemble des modèles fabriqués par ce fabricant. Deux fabricants différents peuvent proposer le même matériel. La relation `Ordinateur` donne pour chaque modèle d'ordinateur la vitesse du processeur (en Hz), les tailles de la Ram et du disque dur (en Go) et le prix de l'ordinateur (en €). La relation `Portable`, en plus des attributs précendents, comporte la taille de l'écran (en pouces). La relation `Imprimante` indique pour chaque modèle d'imprimante si elle imprime en couleur (vrai/faux), le tye d'impression (laser ou jet d'encre) et le prix (en €). La relation `Fabricant` stocke les nom et adresse de chaque fabricant, ainsi que le nom de son patron. La relation `Client` stocke les noms et prénoms de tous les clients de tous les fabricants. Enfin, la relation `Achat` regroupe les quadruplets (client c, fabriquant f, modèle m, quantité q) tels que le client de numéro c a acheté q fois le modèle m au fabricant f. On suppose que l'attribut `Quantite` est toujours strictement positif.

1. Proposer une clé primaire pour la relation `Achat` et indiquer ses conséquences en terme de modélisation.
2. Identifier l'ensemble des clés étrangères éventuelles de chaque table.
3. Donner en SQL des requêtes répondant aux question suivantes :
   (a) Quels sont les numéros des modèles des matériels (ordinateur, portable ou imprimante) fabriqués par l'entreprise du nom de Durand ?
   (b) Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?
   (c) Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?
   (d) Quels sont les numéros des clients n'ayant acheté aucune imprimante ?

Exercice 17 *Activation de processus (exo 2023)*

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins k processus en même temps ?"

1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer la complexité de votre algorithme.

On souhaite montrer que `ACTIVATION` est NP-complet.

3. Donner un certificat pour ce problème.

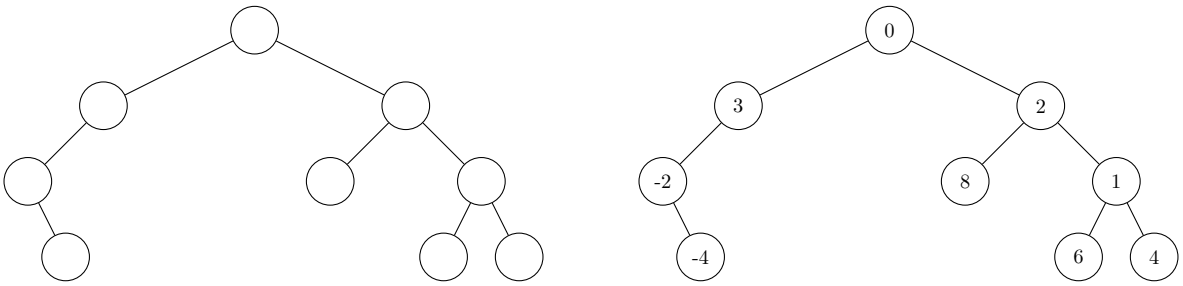
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
   (a) `appartient(c,i)` qui renvoie `Vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
   (b) `intersecte(P`$_i$`,R)` qui renvoie `Vrai` si le processus `i` utilise une ressource incluse dans un ensemble de ressources `R`.
   (c) `ajoute(P`$_i$`,R)` qui ajoute les ressources `P`$_i$ dans l'ensemble `R` et renvoie ce nouvel ensemble.

En théorie des graphes, le problème `STABLE` se pose la question de l'existence dans un graphe non orienté $G = (S, A)$ d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S,\ |S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$?

5. En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Exercice 18 *Minima locaux dans des arbres (exo 2023)*

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts. Un nœud est un minimum local d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils. Considérons par exemple l'étiquetage (b) de l'arbre binaire non étiqueté (a) :



(a) (b)

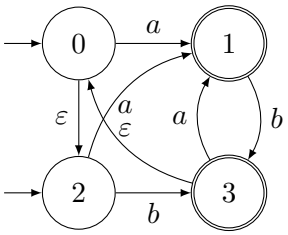
1. Déterminer le ou les minima locaux de l'arbre (b).
2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
3. Montrer que tout arbre non vide possède un minimum local.
4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.

5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
7. Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\left\lfloor \frac{2n+1}{3} \right\rfloor$. *On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.*

Exercice 19 *Langages réguliers (exo 0)*

1. Rappeler la définition d'un langage régulier.
2. Les langages suivants sont-ils réguliers ? Justifier.
   (a) $L_1 = \{a^n b a^m \mid n, m \in \mathbb{N}\}$
   (b) $L_2 = \{a^n b a^m \mid n, m \in \mathbb{N}, n \leq m\}$
   (c) $L_3 = \{a^n b a^m \mid n, m \in \mathbb{N}, n > m\}$
   (d) $L_4 = \{a^n b a^m \mid n, m \in \mathbb{N}, n + m \equiv 0 \bmod 2\}$
3. On considère l'automate non déterministe suivant :



   (a) Déterminiser cet automate.
   (b) Construire une expression régulière dénotant le langage reconnu par cet automate.
   (c) Décrire simplement avec des mots le langage reconnu par cet automate.

Exercice 20 *Mutex (exo 0)*

On considère le programme suivant, ici en OCaml, dans lequel n fils d'exécution incrémentent tous un même compteur partagé.

```
(* Nombre de fils d'exécution *)
let n = 100

(* Un même compteur partagé *)
let compteur = ref 0

(* Chaque fil d'exécution de numéro i va incrémenter le compteur *)
let f i = compteur := !compteur + 1

(* Création de n fils exécutant f associant à chaque fil son numéro *)
let threads = Array.init n (fun i -> Thread.create f i)

(* Attente de la fin de n fils d'exécution *)
let () = Array.iter (fun t -> Thread.join t) threads
```

On rappelle que l'on dispose en OCaml des trois fonctions `Mutex.create : unit -> Mutex.t` pour la création d'un verrou, `Mutex.lock : Mutex.t -> unit` pour le verrouillage et `Mutex.unlock : Mutex.t -> unit` pour le déverrouillage, du module `Mutex` pour manipuler des verrous.

1. Quelles sont les valeurs possibles que peut prendre le compteur à la fin du programme.
2. Identifier la section critique et indiquer comment et à quel endroit ajouter des verrous pour garantir que la valeur du compteur à la fin du programme soit n de manière certaine.

Dans la suite de l'exercice, on suppose que l'on ne dispose pas d'une implémentation des verrous. On se limite au cas de deux fils d'exécution, numérotés 0 et 1. Nous cherchons à garantir deux propriétés :

- *Exclusion mutuelle* : un seul fil d'exécution à la fois peut se trouver dans la section critique ;
- *Absence de famine* : tout fil d'excéution qui cherche à entrer dans la section critique pourra le faire à un moment.

On utilise pour cela un tableau `veut_entrer` qui indique pour chaque fil d'exécution s'il souhaite entrer en section critique ainsi qu'une variable `tour` qui indique quel fil d'exécution peut effectivement entrer dans la section critique. On propose ci-dessous deux versions modifiées `f_a` et `f_b` de la fonction `f`, l'objectif étant de pouvoir exécuter `f_a 0` et `f_a 1` de manière concurrente, et de même pour `f_b`.

```
let veut_entrer = [|false; false|]
let tour = ref 0

let f_a i =
  let autre  = 1 - i in
  veut_enter.(i) <- true;
  while veut_entrer.(autre) do () done;
  (* section critique *)
  veut_entrer.(i) <- false

let f_b i =
  let autre = 1 - i in
  veut_entrer.(i) <- true;
  tour := i;
  while veut_entrer.(autre) && !tour = autre do () done;
  (* section critique *)
  veut_entrer.(i) <- false
```

3. Expliquer pourquoi aucune de ces deux versions ne convient, en indiquant la propriété qui est violée.
4. Proposer une version `f_c` qui permet de garantir les deux propriétés.
5. Connaissez-vous un algorithme permettant de généraliser à n fils d'exécution ? Rappeler très succinctement son principe.

Exercice 21 *Grammaires algébriques (exo 2023)*

On considère la grammaire algébrique G sur l'alphabet $\Sigma = \{a, b\}$ et d'axiome S dont les règles sont :

$$S \to SaS \mid b$$

1. Cette grammaire est-elle ambiguë ? Justifier.
2. Déterminer (sans preuve pour cette question) le langage L engendré par G. Quelle est la plus petite classe de langages à laquelle L appartient ?
3. Prouver que $L = L(G)$.
4. Décrire une grammaire qui engendre L de manière non ambiguë en justifiant de cette non ambiguité.
5. Montrer que tout langage dans la même classe de langages que L peut être engendré par une grammaire algébrique non ambiguë.

Exercice 22 *ID3 (exo 0)*

On considère un problème d'apprentissage supervisé à deux classes dont les données d'apprentissage sont de la forme $Z = (x_i, y_i)_{i\in[\![1,n]\!]}$ avec $\forall i \in [\![1, n]\!]$, $x_i \in \mathbb{B}^d$, $y_i \in \{+, -\}$, où d est le nombre d'attributs binaires d'un exemple et $\mathbb{B} = \{\text{YES}, \text{NO}\}$, les valeurs possibles pour les attributs.

Par exemple, le tableau ci-dessous est un échantillon Z de données relatif aux infections à la COVID 19, extrait de IJCRD 2019. La première colonne indique l'identifiant d'un exemple x (qui comporte trois attributs F, T et R) et la dernière colonne son étiquette y (ici I).

ID	Fièvre (F)	Toux (T)	Problèmes respiratoires (R)	Infecté (I)
1	NO	NO	NO	-
2	YES	YES	YES	+
3	YES	YES	NO	-
4	YES	NO	YES	+
5	YES	YES	YES	+
6	NO	YES	NO	-
7	YES	NO	YES	+
8	YES	NO	YES	+
9	NO	YES	YES	+
10	YES	YES	NO	+
11	NO	YES	NO	-
12	NO	YES	YES	-
13	NO	YES	YES	-
14	YES	YES	NO	-

1. Rappeler le principe de l'apprentissage supervisé.
2. Dessiner l'arbre de décision obtenu en considérant successivement et dans l'ordre les attributs F, T et R. Commenter. *On rappelle qu'un arbre de décions est un arbre binaire dont les noeuds internes sont étiquetés par les attributs et les feuilles par $\{+, -\}$. Les fils gauches correspondent à une réponse NO et les fils droits à la réponse YES.*

L'entropie d'un ensemble S d'exemple est définie par :

$$H(S) = -\frac{n_+}{n}\log_2\left(\frac{n_+}{n}\right) - \frac{n_-}{n}\log_2\left(\frac{n_-}{n}\right)$$

où n_+, n_- et n désignent respectivement le nombre d'éléments de S dont l'étiquette est $+$, le nombre d'éléments de S dont l'étiquette est $-$ et enfin le nombre total d'éléments de S. Dans le cas où $k = 0$, on prend la convention que $k \log_2 k = 0$. Par exemple l'entropie de l'ensemble de toutes les données Z ci-dessus est 1.00.

Étant donné un attribut A, on définit le gain de A par rapport à S par :

$$G(S, A) = H(S) - \frac{n_{A=YES}}{n}H(S_{A=YES}) - \frac{n_{A=NO}}{n}H(S_{A=NO})$$

où $S_{A=YES}$ désigne le sous-ensemble des éléments de S dont l'attribut A est YES et $n_{A=YES}$ désigne son cardinal, de même pour NO et n désigne toujours le cardinal de S. Par exemple $G(Z, F) = 0.26$, $G(Z, T) = 0.07$ et $G(Z, R) = 0.26$ (les valeurs données sont approchées au centième).

Si l'on considère le sous-ensemble $Z_{F=YES}$ des individus qui ont eu de la fièvre, et en supprimant l'attribut F, on obtient le sous-tableau ci-dessous. Le gain d'information de l'attribut T est alors $G(Z_{F=YES}, T) = 0.20$.

ID	Toux (T)	Problèmes respiratoires (R)	Infecté (I)
2	YES	YES	+
3	YES	NO	-
4	NO	YES	+
5	YES	YES	+
7	NO	YES	+
8	NO	YES	+
10	YES	NO	+
14	YES	NO	-

3. Calculer le gain d'entropie $G(Z_{F=YES}, R)$ de l'attribut problèmes repiratoires pour le sous-ensemble des individus qui ont eu de la fièvre.

L'algorithme *Iterative Dichotomiser 3 (ID3)* (Algorithme 1) peut être utilisé pour construire un arbre de décision. Pour l'appel initial, il suffit de prendre l'ensemble de tous les exemples pour S_p et pour S, et l'ensemble de tous les attributs pour D.

Fonction `ID3`(S_p, S, D)
Entrée : S_p sous-ensemble des exemples du noeud parent, S sous-ensemble des exemples à considérer, D sous-ensemble des attributs à considérer
Sortie : Un arbre de décision
si l'ensemble des exemples S est vide **alors**
  renvoyer .....
si l'ensemble A des attributs est vide **alors**
  renvoyer .....
si tous les exemples de S ont une même étiquette y **alors**
  renvoyer .....
sinon
  Soit $A \in D$ l'attribut de plus grand gain $G(S, A)$
  Construire l'arbre de racine A et de sous-arbre gauche `ID3`$(S, S_{A=NO}, D \setminus \{A\})$ et de sous-arbre droit `ID3`$(S, S_{A=YES}, D \setminus \{A\})$

Algorithme 1 - Algorithme ID3

4. Indiquer comment compléter l'algorithme 1.

Exercice 23 *Formules propositionnelles croissantes (exo 2023)*

On fixe un entier $n \geq 1$ et $E = \{x_1, \cdots, x_n\}$ un ensemble de variables propositionnelles. Étant données deux applications $a : E \to \{V, F\}$ et $b : E \to \{V, F\}$ on dit que a est plus petite que b (que l'on note $a \leq b$) si :

$$\forall x \in E, a(x) = V \implies b(x) = V.$$

Dans un but de simplification des calculs, on pourra faire les abus de notation suivants : assimiler V à 1 et F à 0 et vice versa. Avec cet abus, la propriété $a \leq b$ se traduit par :

$$\forall x \in E, a(x) \leq b(x).$$

1. Étant donnée une valuation sur E, rappeler comment on l'étend naturellement en une valuation sur les formules propositionnelles.

On dit qu'une formule propositionnelle P est *croissante* si pour tout a, b des valuations vérifiant $a \leq b$, on a :

$$a(P) = V \implies b(P) = V.$$

2. Montrer que si P, Q sont des formules croissantes, alors $P \wedge Q$ et $P \vee Q$ sont des formules croissantes.
3. Soit C une clause conjonctive satisfiable contenant au moins un littéral. Montrer qu'elle est croissante si et seulement si elle ne contient aucun littéral de la forme $\neg x$ avec $x \in E$.
4. On considère une formule propositionnelle P qui n'est ni une tautologie, ni une antilogie.
   (a) Montrer que si P est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$, alors P est une formule propositionnelle croissante.
   (b) Réciproquement, montrer que si P est une formule propositionnelle croissante, alors elle est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$.

Exercice 24 *Arbres binaires de recherche (exo 0)*

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

1. Rappeler la définition d'un arbre binaire de recherche.
2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot *bacddabdbae* en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?
3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.
4. Proposer un algorithme qui permet de compter le nombre d'occurrences d'une lettre dans un arbre binaire de recherche de lettres. Quelle est sa complexité ?
5. On souhaite supprimer *une* occurrence d'une lettre donnée dans un arbre binaire de recherche de lettres. Expliquer le principe de l'algorithme permettant de résoudre ce problème et le mettre en oeuvre sur l'arbre obtenu à la question 2 en supprimant successivement une occurrence des lettres e, b, b, c et d. Quelle est sa complexité ?