Quelques exercices et problèmes, proposés en 1999/2000 ou avant.
Pour certains, très simples, il existe des prédicats prédéfinis faisant le même travail, mais les récrire constitue un bon exercice.
Certains sont des questions de sujets d'examens de cours Prolog et/ou intelligence artificielle.
Pour les exercices pas trop compliqués, écrire une version déclarative et une version plus efficace procédurale peut être intéressant (même si ce n'est pas précisé dans l'énoncé).
Tracer à la main des arbres de sous-buts (tous les sous-buts ou seulement les plus importants selon la taille du problème) est aussi un bon exercice.

Pour quelques problèmes, les amateurs de casse-têtes pourront chercher à les résoudre à la main, sans utiliser la force brute de la machine.

Premiers exercices : element – longueur – retirer – dernier

Écrire un prédicat Prolog `element(X,L)` qui est vrai si `X` est un élément de la liste `L`.
Exemples `element(c,[a,b,c,d])` est vrai,
`element(e,[a,b,c,d])` échoue,
`element(X,[a,b,c,d])` renvoie `X=a`, puis `X=b`, puis `X=c`, puis `X=d`,
Donner l'arbre des sous-buts pour ce dernier exemple
Tester et donner l'arbre des sous-but pour la question `element(a,L).`

Écrire un prédicat Prolog `longueur(L,N)` qui renvoie dans `N` la longueur de la liste `L`.

Écrire un prédicat Prolog `retirer3(L,Lmoins3)` qui retire les trois derniers éléments de la liste `L` et renvoie le résultat dans `Lmoins3`.
Exemple `retirer3([a,b,c,d,e,f,g,h],L)` renvoie `[L=[a,b,c,d,e].`
Aide : on pourra utiliser la concaténation des listes vue en cours.

Écrire un prédicat Prolog `retirer33(L,Lmoins33)` qui retire les ttrois premiers et les trois derniers éléments de la liste `L` et renvoie le résultat dans `Lmoins33`.
Exemple `retirer33([a,b,c,d,e,f,g,h],L)` renvoie `L=[d,e].`

Écrire un prédicat Prolog `der(X,L)` qui est vrai si `X` est le dernier élément de la liste `L`.
Exemples `der(d,[a,b,c,d])` est vrai,
`der(c,[a,b,c,d])` et `der(e,[a,b,c,d])` échouent,
`der(X,[a,b,c,d])` renvoie `X=d`.
Donner l'arbre des sous-buts pour ce dernier exemple.
Tester et donner l'arbre des sous-but pour la question `der(a,L).`
On écrira une version utilisant la concaténation des listes et une autre version ne l'utilisant pas. Comparer.

Calculs de maximum/minimum

Écrire un prédicat Prolog `max3` qui calcule le maximum de 3 nombres.
Exemple `max3(4,8,2,M)` renvoie `M=8`.
Votre programme est-il déclaratif ? Vous pouvez donner plusieurs versions et les comparer.

Ecrire un prédicat Prolog `max` qui calcule le maximum d'une liste de nombres.
Exemple `max([5,3,7,2],M)` renvoie `M=7`.

Ecrire un prédicat Prolog `minmax` qui, étant donnée une liste de listes de nombres, calcule le minimum des maxima des listes.
Exemple `minmax([[5,3,7,2],[4,2,3],[1,3,8]],M)` renvoie `M=7`.

Permutations

Ecrire un prédicat Prolog `permutation` qui donne toutes les permutations d'une liste donnée
a) 1ère version : une permutation d 'une liste `[X|L]` est obtenue en insérant `X` n'importe où dans une permutation de `L`; on écrira et utilisera donc un prédicat `inserer` qui insère un élément dans une liste (n'importe où) ;
b) 2ère version : une permutation d 'une liste `L` est obtenue en ôtant de `L` n'importe quel élément `X`, soit `L1` la liste ainsi obtenue, puis en ajoutant au début d'une permutation de `L1`; on écrira et utilisera donc un prédicat `oter` qui retire un élément d'une liste (n'importe lequel) .

Dans quel ordre les permutations sont-elles générées ?
Donner l'arbre des sous-buts pour la question `permutation(a,b,c).`

Tris

Ecrire des prédicats Prolog `tri(L,T)`, `bulle(L,T)` et `quick(L,T)` qui, recevant une liste de nombres `L`, renvoient dans `T` la liste triée.
Exemple, `tri([5,1,2,6],T)` *instancie* `T` à `[1,2,5,6]`.
Pour chacun des tris, on donnera une version déclarative et un version non déclarative évitant certains tests.

1- Pour le *tri par insertion* (`tri`) écrire et utiliser un prédicat `inserer_tr(X,L1,L2)` qui recevant un nombre `X` et une liste `L1` déjà triée, insère `X` à sa place et renvoie le résultat dans `L2`.
Exemple, `inserer_tr(6,[1,3,4,8],L)` *instancie* `L` à `[1,3,4,6,8]`.

2- Pour le *tri de la bulle* (`bulle`) écrire et utiliser un prédicat `echange(L1,L2)` qui, recevant une liste `L1` , échangent les deux premières valeurs consécutives mal placées et renvoie le résultat dans `L2` , et *échoue* s'il n'y a pas de valeurs mal placées.
Exemples, `echange([1,3,6,4,9,7],L)` *instancie* `L` à `[1,3,4,6,9,7]`; `echange([1,3,5,6],L)` *échoue*.
Pour la version déclarative, écrire et utiliser un prédicat `trie(L)` qui est *vrai* (c'est-à-dire *réussit*)

si `L` est déjà triée et *échoue* sinon.

3- Pour le *tri rapide* (`quick`), écrire et utiliser un prédicat `couper(X,L,L1,L2)` qui, recevant un pivot `X` et une liste `L`, renvoie dans `L1` la liste des éléments plus petits que `X` et dans `L2` la liste des éléments plus grands ou égaux à `X`.
Exemple, `couper(4,[6,2,8,1],L1,L2)` renvoie `L1=[2,1]` et `L2=[6,8]`.
Dire si votre programme est déclaratif ou non.
(On rappelle que la méthode du *tri rapide* consiste à choisir un *pivot*, par exemple le premier élément, et à construire la liste triée en prenant d'abord la liste des éléments plus petits que le pivot, elle-même triée par un appel récursif, puis le pivot, enfin la liste des éléments plus grands, également triée. On veillera à éviter tout bouclage !)

Les nombres de 1 à N

Ecrire un prédicat Prolog `gen(N,L)` qui construit, pour un entier `N` donné la liste `L` des `N` premiers entiers.
Exemple `gen(4,L)` donne `L=[1,2,3,4]`.
1ère version facile : construire `L=[4,3,2,1]` puis inverser.
Donner l'arbre des *sous-buts* pour la question `gen(3,L)`.
2ème version meilleure (dire pourquoi) : écrire et utiliser un prédicat auxiliaire `gen(N1,N2,L)` qui construit, pour `N1` et `N2` entiers donnés tels que `N1<=N2`, la liste `L` des entiers compris entre `N1` et `N2`.
Exemple `gen(3,7,L)` donne `L=[3,4,5,6,7]`.
Donner l'arbre des *sous-buts* pour la question `gen(3,L)`.

Dupliquer / Répéter

Ecrire un prédicat Prolog `dup` qui duplique tous les éléments d'une liste. La question `dup([a,b,c,d],L)` doit donner `L=[a,a,b,b,c,c,d,d])`.

Généralisation
Ecrire un prédicat Prolog `rep` qui répète N fois tous les éléments d'une liste. La question `rep([a,b,c,d],5,L)` doit donner `L=[a,a,a,a,a,b,b,b,b,b,c,c,c,c,c,d,d,d,d,d])`.

Et un autre
Ecrire un prédicat Prolog `repeter(L1,L2)` qui, à partir d'une liste `L1`=[x_1, x_2, ... , x_i, ...], construit la liste `L2`=[x_1, x_2, x_2, x_3, x_3, x_3, ... , x_i, ... , x_i] où x_i apparaît i fois.
La question `repeter([a,b,c,d],L)` doit donner `L=[a,b,b,c,c,c,c,d,d,d,d])`.
Donner l'arbre des *sous-buts* pour les questions suivantes `repeter([a,b,c],L)`, `repeter(L,[a,b,c])`, `repeter(L,[a,b,b],L)`.
Aide : on pourra écrire et utiliser un prédicat `repeter(L1,N,L2)` qui construit `L2` à partir de `L1` en répétant `N` fois le premier élément, `N+1` fois le deuxième, etc ... .

Retirer

Ecrire un prédicat Prolog `element(L,X,LX)` qui est vrai si `X` est un élément de `L` et `LX` la liste obtenue en retirant `X` de `L`.
Exemples :
`element([a,b,c,d],c,[a,b,d])` est vrai
`element([a,b,c],X,L)` renvoie `X=a` et `L=[b,c]`, puis `X=b` et `L=[a,c]`, puis `X=c` et `L=[a,b]`

Ecrire un prédicat Prolog `retire1sur2(L,L2)` qui retire un élément sur deux d'une liste à partir du deuxième, c'est-à-dire `retire1sur2(L,L2)` est vrai si `L2` est composée des 1^{er}, $3^{ème}$, $5^{ème}$, etc ... éléments de `L`.
Exemple `retire1sur2([a,b,c,d,e,f,g],L2)` renvoie `L2=[a,c,e,g]`.
Donner l'arbre des *sous-buts* pour cet exemple, ainsi que que pour le but `retire1sur2(L,[a,b,c])`.

Ecrire un prédicat Prolog `retire1sur3(L,L3)` qui retire un élément sur trois d'une liste, c'est-à-dire `retire1sur3(L,L3)` est vrai si `L3` est composée des 1^{er}, $2^{ème}$, $4^{ème}$,$5^{ème}$,$7^{ème}$, etc ... éléments de `L`.
Exemple `retire1sur3([a,b,c,d,e,f,g],L3)` renvoie `L3=[a,b,d,e,g]`.
Donner l'arbre des *sous-buts* pour cet exemple.

Ecrire un prédicat Prolog `retire1surN(L,N,LN)` qui retire un élément sur `N` d'une liste, c'est-à-dire `retire1surN(L,LN)` est vrai si `LN` est obtenue à partir de `L` en supprimant les $N^{ème}$, $(2N)^{ème}$, $(3N)^{ème}$, etc ... éléments de `L`.
Exemple `retire1surN([a,b,c,d,e,f,g],3,LN)` renvoie `LN=[a,b,d,e,g]`.
Aide : on pourra écrire et utiliser un prédicat `retire1surN(L,I,N,LN)` qui retire un élément sur `N` à partir du $I^{ème}$.

Additions

Ecrire un prédicat Prolog `add(L,A,L1)` qui, recevant une liste de nombres `L` et un nombre `A`, renvoie la liste `LplusA` où tous les nombres de `L` ont été augmentés de `A`.
Exemple `add([3,6,4],2,L)` renvoie `LplusA=[5,8,6]`.
Donner l'arbre des *sous-buts* pour cet exemple.

Ecrire un prédicat Prolog `add(L,L1)` qui, recevant une liste de nombres `L`, renvoie la liste `L1` où le premier nombre de `L` a été augmenté de `1`, le deuxième de `2`, ... le $i^{ème}$ de `i`.
Exemple `add([3,6,4],L)` renvoie `L=[4,8,7]`.
Aide : on pourra écrire et utiliser un prédicat `add(L,N,L1)` qui augment le premier nombre de `N`, le deuxième de `N+1`, etc ... .
Donner l'arbre des *sous-buts* pour cet exemple.

Mélanges de listes

Ecrire un prédicat Prolog `melange_tr` qui mélange deux listes de nombres déjà triées en une liste

triée.
Exemple `melange_tr([1,4,9,14],[2,11,12],L)` doit donner `L=[1,2,4,9,11,12,14].`
Donner l'arbre des *sous-buts* pour l'exemple précédent.
Que se passe-t-il si on appelle `melange_tr(L1,L2,[1,3,6,7])` ?

Ecrire un prédicat Prolog `melange1par1` qui mélange deux listes d'éléments en prenant alternativement un élément dans chaque liste et en complétant par ce qui reste de la plus longue.
Exemples `melange1par1([a,b,c],[x,y,z],L)` donne `L=[a,x,b,y,c,z]`, `melange1par1([a,b,c],[u,v,x,y,z],L)` donne `L=[a,u,b,v,c,x,y,z].`
Donner l'arbre des sous-buts pour ces deux exemples.
Que se passe-t-il si on demande `melange1par1(L1,L2,[a,b,c,d])`? Donner l'arbre des sous-buts.

Ecrire un prédicat Prolog `brouille` qui mélange deux listes de la façon suivante : on prend un élément de la première, puis deux éléments de la deuxième, puis trois de la première, puis quatre de la deuxième, etc, tant qu c'est possible. On compète par de qui reste dans les deux listes.
Exemples `brouille([a1,a2,a3,a4,a5,a6,a7],[b1,b2,b3,b4,b5,b6,b7,b8],L)` donne `L=[a1,b1,b2,a2,a3,a4,b3,b4,b5,b6,a5,a6,a7,b7,b8].`
Donner l'arbre des sous-buts pour cet exemple.

Ecrire un prédicat Prolog `melangetous` qui mélange deux listes d'éléments de toutes les façons possibles, en gardant l'ordre des éléments de chaque liste.
Exemple `melangetous([a,b,c],[d,e],L)` donne pour `L` les valeurs suivantes
`[d,e,a,b,c]`, `[d,a,e,b,c]`, `[d,a,b,c,e]`, `[d,a,b,c,e]`, `[a,d,e,b,c]` ... etc.
Donner toutes les valeurs de `L` dans l'ordre dans lequel votre programme les trouvera.
Que se passe-t-il si on appelle `melangetous(L1,L2,[a,b,c])` ?

Mise à plat

Ecrire un prédicat Prolog `plat` qui met une liste à plat, c'est-à-dire qui, à partir d'une liste qui peut être une liste de listes de llistes ..., renvoie la liste composée au premier niveau de tous les éléments.
Exemple `plat([[a,[b,c]],d,e,[[[f]]],g],L)` renvoie `L=[a,b,c,d,e,f,g].`
Donner l'arbre des *sous-buts* pour la question `plat([[a,b],[c]],L).`
(Si des prédicats auxiliaires sont utilisés, on ne développera pas les sous-arbres de sous-buts pour ces prédicats.)

Les mutants

Ecrire un prédicat Prolog qui crée de nouveaux noms d'animaux à partir de noms connus de la façon suivante : à partir d'un nom de la forme *nom1nom2* et d'un nom de la forme *nom2nom3*, on obtient le nom *nom1nom3*. Les noms seront des atomes.
Exemple : à partir de

```
animal(vache).
animal(cheval).
animal(chevre).
```

`mutant(A)` renvoie `A=vacheval` et `A=vachevre.`

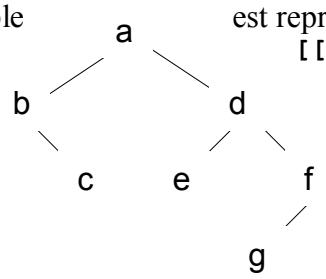
On pourra utiliser le prédicat prédéfini `name` qui donne la correspondance entre une nom et la liste des codes ASCII de ses caractères
(exemple `name(abc,L)` renvoie `L=[97,98,99]`, `name(A,[100,101,102]` renvoie `A=def`)
et travailler sur des listes de caractères, en écrivant et utilisant une variante de la concatenation des listes vue en cours.

On pourra aussi utiliser, s'il existe dansle Prolog que vous utilisez, et **avec précaution**, le prédicat prédéfini `concat` qui concatène des chaînes de caractères
(exemple `concat(abc,de,L)` renvoie `L = abcde`, mais attention, tester ce que fait `concat(A,B,abcde)`).

Arbres binaires de recherche

On représente un *arbre binaire* par une liste PROLOG de trois éléments qui sont
- la représentation du sous-arbre gauche,
- la racine,
- et la représentation du sous-arbre droit.

Un arbre vide est représente par `[]`

Par exemple 

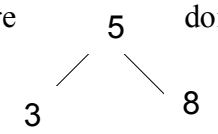
est représenté par la liste

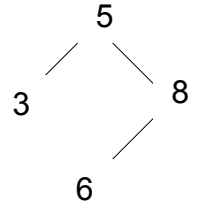
```
[[[],b,[[],c,[]]],a,[[[],e,[]],d,[[[],g,[]],f,[]]]]
                                                  ]
```

Tous les noeuds d'un *arbre binaire de recherche* sont étiquetés par des nombres. L'étiquette de sa racine est supérieure à toutes les étiquettes de son sous-arbre gauche et inférieure ou égale à tous les éléments du sous-arbre droit. Les sous-arbres sont également des arbres binaires de recherche.

1. Ecrire un prédicat PROLOG `rech(A)` qui vérifie si une liste `A` représente un arbre binaire de recherche.

2. Ecrire un prédicat PROLOG `ins(X,A,AR)` qui insère un élément `X` dans un arbre binaire de recherche `A` pour donner 1'arbre `AR`.
Méthode: si l'élément `X` est plus petit que la racine, on 1'insère dans 1e sous-arbre gauche, sinon on 1'insère dans le sous-arbre droit.

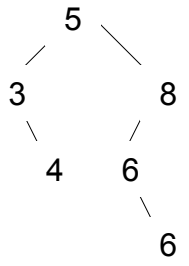
Exemple: 1'insertion de 6 dans 1'arbre 

donne l'arbre 

L'insertion d'un élément `X` dans 1'arbre vide donne 1'arbre `[],X,[]`

3. Utiliser un arbre binaire de recherche pour trier une liste de nombres, c'est-à-dire écrire un prédicat `tri(L,T)` qui, recevant une liste `L`, renvoie dans `T` la liste triée.
Méthode: construire un arbre binaire de recherche `A` à partir de la liste `L` en insérant, l'un après 1'autre, tous les éléments de `L` ; puis construire la liste `T` qui correspond à un parcours infixe de 1'arbre, c'est-à-dire *à plat* et sans les `[]` intermédiaires, par un prédicat `p1at(A,T)`.

Exemple: La liste `L=[4,6,6,8,3,5]` donne 1'arbre



représenté par la liste
`A=[[[],3,[],4,[]]],5,[[[],6,[],6,[]]],8,[]]`
puis la liste `T=[3,4,5,6,6,8]`

Vos ensembles de clauses sont-i1s déclaratifs? Justifier votre réponse.

Chemin minimal dans un graphe

Ecrire un ensemble de clauses PROLOG qui cherche un chemin minimal d'un noeud à un autre dans un graphe.
On ne cherchera pas l'efficacité et on écrira un ensemble de clauses le plus déclaratif possible, tout usage d'une *coupe*, d'un *échec* ou d'une *négation* devra être justifié.
On suppose que les arcs valués sont donnés au moyen de clauses de la forme `arc(X,Y,C)` qui signifie qu'il y a un arc de coût `C` de `X` à `Y`.
Un chemin sera représenté par une liste. Pour plus de commodité, les chemins seront construits *à l'envers*, le chemin `ABC` étant représenté par la liste `[C,B,A]`.

1. Ecrire d'abord un prédicat PROLOG `chemin(D,A,CH)` qui recevant deux noeuds de départ et d'arrivée `D` et `A`, renvoie un chemin de `D` à `A`.
On écrira et utilisera un prédicat auxiliaire `chemin1(A,CH1,CH2)` qui, recevant l'arrivée `A` et un chemin `CH1`, prolonge `CH1` en un chemin `CH2` arrivant en `A` de la facon suivante: si `CH1` commence par `A`, alors on renvoie `CH2=CH1`, sinon on prolonge `CH1` par un successeur `Y` du premier élément de `CH1`, qui ne se trouve pas déjà dans `CH1` et on appelle de nouveau `chemin1`.

2. Modifier les programmes précédents de façon à faire intervenir le coût des chemins, `chemin(D,A,CH,CO)` renvoyant dans `CO` le coût du chemin `CH`. `chemin1` comportera comme paramètres supplémentaires le coût de `CH1` (connu à l'appel), et le coût de `CH` (que l'on cherche). Le coût du chemin `[Y,X|CH]` sera bien sûr égal à la somme des coûts de `[X|CH]` et de l'arc `XY`.

3. On veut maintenant un chemin minimal: pour cela on cherche un chemin tel qu'il n'y ait pas d'autre chemin de coût inférieur, au moyen du prédicat `chemin_min` suivant:

```
chemin_min(D,A,CH,CO):- chemin(D,A,CH,CO),ppch(D,A,CH,CO).
```

`ppch(D,A,CH,CO)` étant un prédicat qui échoue s'il existe un chemin de coût < CO.
Ecrire `ppch`.
(On ne se préoccupera pas de l'efficacité d'un tel programme avec un interpréteur PROLOG standard !)

Remplissage de cruches

Problème:

Disposant de deux cruches de contenances respectives 3 litres et 7 litres, initialement vides, comment obtenir 8 litres d'eau, sachant que l'on peut remplir ou vider chacune des cruches, ou transvaser le contenu (ou une partie du contenu) de l'une dans l'autre.

Remarque:

Les différentes parties sont indépendantes, mais la réflexion pour chacune d'elles peut aider a résoudre les autres.

I - Résolution graphique

Résoudre graphiquement ce problème
1°) en utilisant un graphe dont les sommets sont toutes les situations possibles, c'est-à-dire les couples des contenances respectives des deux cruches, et les arcs les mouvements permettant de passer d'une situation à une autre.
2°) en utilisant la représentation planaire suivante: la situation *(x,y)* est représentée dans le plan par le point de coordonnées *x* et *y* ; et après avoir décrit géométriquement les mouvements possibles à partir d'une situation quelconque *(x,y)*.

II - Quelques propriétés

1°) Combien y a-t-il de situations possibles (à partir de la situation initiale *(0,0)*)?
2°) On appelle *raisonnable* une solution sans cycle dans laquelle on ne va jamais en plus de un mouvement à une situation que l'on pourrait atteindre en un seul mouvement à partir de la situation initiale. Combien y a-t-il de solutions raisonnables? Que peut-on dire des nombres de mouvements des solutions raisonnables pour obtenir une quantité quelconque d'eau?
3°) Généraliser les deux questions précédentes pour deux cruches de contenances quelconques *c1* et *c2* (nombres entiers de litres).

III - Description en PROLOG

On définit un prédicat `cruches(X,Y,L,N)` qui est vrai si l'on peut obtenir `X` litres dans la première cruche, `Y` litres dans la deuxième, en `N` mouvements, `L` décrivant la suite des mouvements (par exemple la liste, à l'envers pour plus de facilité, des couples de contenances successives).
En utilisant uniquement ce prédicat `cruches` (et des prédicats et fonctions prédéfinis PROLOG), écrire un ensemble déclaratif de clauses PROLOG décrivant le problème. On ne demande pas dans cette question que le problème puisse effectivement être résolu par un interpréteur PROLOG classique.

Que faudrait-il ajouter à l'interpréteur PROLOG ou aux clauses pour que le problème puisse être effectivement resolu ?

IV - Résolution en PROLOG

1°) Ecrire un programme PROLOG qui résout effectivement le problème en simulant une démarche avant. On pourra définir et utiliser les prédicats suivants:
`suivant(S1,S2)` est vrai si la situation `S2` peut être obtenue à partir de `S1` en un mouvement;
`trouver(Sol,Sol1,But,N)` est vrai si `Sol` est une solution prolongeant `Sol1` pour obtenir la situation `But` en `N` mouvements.
On pourra être amené à définir d'autre(s) prédicat(s) PROLOG.
On pourra représenter une situation par le couple des contenances respectives des deux cruches (liste de deux éléments), une solution par la liste, à l'envers pour plus de facilité, des situations successives qui permettent de passer de la situation initiale `[0,0]` à la situation `But`.
2°) Donner l'arbre des sous-buts `trouver(...)` que l'interpréteur devra résoudre, et la solution trouvée par le programme. Est-elle *raisonnable* ? (voir partie II)

Les nombres chanceux

On considère la liste de tous les entiers de `1` à `N`. On en supprime un sur deux (à partir du 2ème), puis un sur trois (à partir du 3ème) et un sur quatre (à partir du 4ème). On recommence : un sur deux, puis trois, puis quatre, puis de nouveau un sur deux, puis trois, puis quatre, etc. On s'arrête si lors d'une opération, on n'a pas supprimé de nombre. On appelle alors nombres *chanceux* d'ordre `N`, s'ils existent, ceux qui restent, en dehors de 1. Le dernier nombre à être rayé de la liste est le nombre *malchanceux* d'ordre `N`.
Exemples :
`[1,2,3,4,5,6,7,8,9]` devient `[1,3,5,7,9]` puis `[1,3,7,9]` puis `[1,3,7]` puis `[1,7]` ; `7` est le seul nombre chanceux d'ordre `9`.
`[1,2,3,4,5,6,7]` devient `[1,3,5,7]` puis `[1,3,7]`; `3` et `7` sont les nombres chanceux d'ordre `7`.
`[1,2]` devient `[1]`; il n'y a pas de nombre chanceux d'ordre `2`.

1. Ecrire un prédicat Prolog `ch` qui détermine les nombres chanceux d'ordre `N`. Pour cela, on écrira les prédicats suivants :
`lis(N,L)` renvoie dans `L` la liste de tous les entiers de `1` à `N`
exemple : `lis(7,L)` renvoie `L=[1,2,3,4,5,6,7]`
prédicat auxiliaire `lis1(I1,I2,L)` renvoie dans `L` la liste de tous les entiers de `I1` à `I2`
exemple : `lis1(3,6,L)` renvoie `L=[3,4,5,6]`
`supp(L1,K,L2)` renvoie dans `L2` la liste obtenue à partir de `L1` en supprimant les `K`ème, `2K`ème, `3K`ème, etc ... éléments de `L1` .
exemple : `supp([a,b,c,d,e,f,g],3,L)` renvoie `L=[a,b,d,e,g]`
prédicat auxiliaire `supp1(L1,I,K,L2)` renvoie dans `L2` la liste obtenue à partir de `L1` en supprimant de `L1` un élément sur `K` à partir du `I`ème
exemple : `supp1([a,b,c,d,e,f,g],2,3,L2)` renvoie `L2=[a,c,d,f,g]`
`c(L1,K,L2)` retire les `K`ème, `2K`ème, `3K`ème, etc ... éléments de `L1`, puis
- renvoie dans `L2` la liste obtenue `L3` si celle-ci est égale à `L1` ;
- sinon appelle récursivement `c` pour `L3` et pour `K+1` (si `K<4`) ou `2` (si `K=4`), qui renvoie `L2`

`ch(N,L)` renvoie dans `L` la liste des nombres chanceux d'ordre `N`.

2. Construire l'arbre des sous-buts pour la recherche des nombres chanceux d'ordre `9`. On ne développera pas sur l'arbre des appels à `lis` et à `supp` .

3. Modifier et enrichir votre programme pour obtenir aussi le nombre malchanceux d'ordre `N`.

Zigzag

1. Ecrire un prédicat Prolog `element(L,X,LX)` qui est vrai si `X` est un élément de `L` et `LX` la liste obtenue en retirant `X` de `L`.
Exemples :
`element([a,b,c,d],c,[a,b,d])` est vrai
`element([a,b,c],X,L)` renvoie `X=a` et `L=[b,c]`, puis `X=b` et `L=[a,c]`, puis `X=c` et `L=[a,b]`

2. Ecrire un prédicat Prolog `gen(N,LN)` qui recevant un nombre entier `N`, crée une liste constituée des entiers de `1` à `N`.
Exemple : `gen(4,L4)` renvoie `L4=[1,2,3,4]` ou, si l'on préfère `L4=[4,3,2,1]`

3. On appelle *liste-zigzag* une liste de nombres telle que chaque élément de la liste est alternativement plus grand ou plus petit que son prédecesseur.
Exemple : `[21,4,8,1,5]` est une liste-zigzag .
On appelle *permutation-zigzag d'ordre N* une permutation des nombres de `1` à `N` qui est une liste-zigzag telle que le premier élément de la liste est plus petit que le deuxième.
Exemple : `[4,6,1,3,2,5]` est une permutation-zigzag d'ordre `6`.
On pourra, pour plus de commodité et en le précisant clairement, représenter cette permutation-zigzag par la liste "à l'envers" `[5,2,3,1,6,4]`

Ecrire un prédicat Prolog `zigzag(N,ZZ)` qui renvoie dans `ZZ` toutes les permutations-zigzag d'ordre `N`.
On pourra écrire et utiliser comme prédicats auxiliaires, outre les prédicats des questions 1 et 2, un prédicat `p(L1,L2,A,Z)` qui, recevant un début de liste zigzag `L1` et une liste `L2`, prolonge `L1` en une liste-zigzag `Z` constituée des éléments de `L1` et `L2`, le paramètre `A` indiquant si l'on doit ajouter à `L1` un élément plus petit ou plus grand que son dernier élément.

4. Ecrire un predicat `compter(N,NZ)` qui donne le nombre `NZ` de permutations-zigzag d'ordre `N`.

Représentations de suites – sommes terme à terme – traductions

Les questions, de difficulté croissante, sont indépendantes.
Tous les ensembles de clauses demandés seront déclaratifs.

1. Ecrire un prédicat Prolog `somme1` qui fait la somme terme à terme des éléments de deux listes de nombres de même longueur et échoue si les deux listes ne sont pas de même longueur.

Exemple : `somme1([4,7,8],[1,9,2],L) renvoie L=[5,16,10]`

2. Plutôt que de représenter une suite d'éléments a_1, a_2, … , a_n par la liste $[a_1,a_2,\ldots,a_n]$, on peut la représenter par la liste des couples (i, a_i), ordonnée par rapport à i, où a_i est le $i^{ème}$ élément de la suite.
Exemple : 4, 7, 8 est représentée par `[(1,4),(2,7),(3,8)]`
Ecrire un prédicat `somme2` qui, en utilisant cette représentation des suites, fait la somme terme à terme de deux suites de même longueur et échoue si les deux suites ne sont pas de même longueur.
Exemple : `somme2([(1,4),(2,7),(3,8)],[(1,1),(2,9),(3,2)],L)` renvoie
`L=[(1,5),(2,16),(3,10)]`
Remarque : on ne demande pas de vérifier que la liste est bien ordonnée par rapport à i.

3. On représente maintenant la suite par une liste composée de la longueur de la suite et des couples (i, a_i), ordonnés par rapport à i, où a_i est le $i^{ème}$ élément de la suite et est différent de 0.
Exemple : 0, 4, 0, 0, 7, 0, 8, 0 est représentée par `[8,(2,4),(5,7),(7,8)]`
(L'intérêt de cette représentation est évidemment pour les suites comportant beaucoup de zéros.)
Remarque : on ne demande pas de vérifier que la liste est bien ordonnée par rapport à i.
Ecrire un nouveau prédicat `somme3` pour cette représentation.
Exemple : `somme3([8,(2,4),(5,7),(7,8)],[8,(2,3),(5,-7)],L)` renvoie
`L=[8,(2,7),(7,8)]`
Aide : On pourra écrire un prédicat auxiliaire `s3` qui fait le travail sans la longueur.
Exemple : `s3[(2,4),(5,7),(7,8)],[(2,3),(5,-7)],L)` renvoie `L=[(2,7),(7,8)]`

4. On considère les trois représentations suivantes d'une suite de nombres a_1, a_2, … , a_n :
(1) la liste des a_i, *exemple* `[0,9,0,6,0]`
(2) la liste des couples (i, a_i), ordonnée par rapport à i, où a_i est le $i^{ème}$ élément de la suite *exemple* `[(1,0),(2,9),(3,0),(4,6),(5,0)]`
(3) la liste composée de la longueur de la suite et des couples (i, a_i), ordonnée par rapport à i, où a_i est le $i^{ème}$ élément de la suite et est différent de 0, exemple `[5,(2,9),(4,6)]`

Ecrire les 6 prédicats Prolog `trad`$_{ij}$ qui permettent de passer de la représentation (i) à la représentation (j). Tous les ensembles de clauses seront déclaratifs.
Exemple : `Trad31([5,(2,9),(4,6)],L1)` donne `L1=[0,9,0,6,0]`
Chacun des 6 prédicats doit être écrit indépendamment des autres, c'est-à-dire, on **n**'écrira **pas**, par exemple ~~`trad13(L1,L3) :- trad12(L1,L2),trad23(L2,L3).`~~
Aide : du plus facile au plus difficile, les listes `Li` désignant les représentations (i) d'une suite, les listes `L'3` désignant des représentations (3) sans la longueur (*exemple* : `[(2,9),(4,6)]`)

`trad21(L2,L1)`

`trad12(L1,L2)` on pourra écrire un prédicat auxiliaire `t12(L1,I,L2)` qui, recevant une liste `L1` et un indice `I`, donne une liste `L2` commençant à l'indice `I`

`trad23(L2,L3)` on pourra écrire un prédicat auxiliaire `t23(L2,L'3,N)` qui, recevant une liste `L2`, donne une liste `L'3` et une longueur `N`

`trad13(L1,L3)` on pourra écrire un prédicat auxiliaire `t13(L1,I,L'3,N)` qui, recevant une liste `L1` et un indice `I`, donne une liste `L'3` et une longueur `N`

`trad32(L3,L2)` on pourra écrire un prédicat auxiliaire `t32(L'3,I,N,L2)` qui, recevant une liste `L'3`, un indice `I` et une longueur `N`, donne une liste `L2` allant de `I à N`

`trad31(L3,L1)` on pourra écrire un prédicat auxiliaire `t32(L'3,I,N,L1)` qui, recevant une liste `L'3`, un indice `I` et une longueur `N`, donne une liste `L1`

Le taquin

On veut résoudre le problème du "taquin", c'est-à-dire obtenir l'état final suivant à partir d'un état quelconque en faisant glisser les carrés horizontalement ou verticalement vers la case vide.

1	2	3
4	5	6
7	8	

Exemple :

1		3
4	2	5
7	8	6

donne

1	2	3
4		5
7	8	6

puis

1	2	3
4	5	
7	8	6

enfin

1	2	3
4	5	6
7	8	

On représentera un état par une liste de triplets, chaque triplet `(I,J,N)` représentant la case de numéro `N` aux coordonnées `I` et `J`. La case vide sera représentée par `0`. L'état initial précédent sera ainsi représenté par la liste

```
[(1,1,1),(1,2,0),(1,3,3),(2,1,4),(2,2,2),(2,3,5),(3,1,7),(3,2,8),(3,3,6)].
```

1. Ecrire un prédicat Prolog `suivant(T,T1)` qui donne les états suivants `T1` d'un état donné `T` par un seul mouvement.

Exemple :

1		3
4	2	5
7	8	6

a pour suivants

	1	3
4	2	5
7	8	6

,

1	3	
4	2	5
7	8	6

et

1	2	3
4		5
7	8	6

2. Le prédicat suivant `sol(T,TT)` décrit une liste d'états `TT` aboutissant à un état final à partir de `T`.

```
sol(T,[T]) :- final(T).
sol(T,[T|TT]) :- suivant(T,T1),sol(T1,TT).
```

Néanmoins, cet ensemble de clauses, correct d'un point de vue logique ne l'est pas du point de vue d'une exécution Prolog car, les mouvements état réversibles, on obtiendra, sauf pour quelques cas exceptionnels, un bouclage. On v donc définir un autre prédicat `sol1` obtenu en remplaçant le premier paramètre `T` par la liste des états depuis l'état initial jusqu'à `T` et en testant si l'état suivant `T1` n'appartient pas à cette liste, c'est-à-dire :

`sol1(TT,TTF)` est vrai si `TTf` est une liste d'états prolongeant a liste `TT`, ne contenant pas deux fois le même état et aboutissant à un état final.

Ecrire le prédicat `sol1` et un nouveau prédicat `sol` à l'aide de `sol1`.

3. Cette recherche, bien que correcte, est très longue. Pour accélérer la recherche, on va limiter la profondeur des appels successifs.

Ecrire un prédicat `sol2` identique à `sol1` excepté qu'il comporte un troisième paramètre donnant la profondeur maximale autorisée :

`sol2(TT,TTf,PMAX)` est vrai si `TTf` est une liste d'états prolongeant la liste `TT` et aboutissant à un état final en au plus `PMAX` coups.

Ecrire, à l'aide de `sol2`, un prédicat `sol3` résolvant le problème, si possible, en un nombre de coups au plus égal à un nombre donné.

Que se passe-t-il si le problème n'est soluble qu'en un nombre de coups supérieur à ce nombre ?