Chapitre 3

La coupure

On présente en ce chapitre le prédicat prédéfini très important qui aura pour effet de stopper la recherche de Prolog dès qu'il aura trouvé une solution dont on veut qu'elle soit unique. Ce prédicat permet de contrôler l'espace de recherche, d'économiser le travail de Prolog et de définir une négation.

Contrôle de l'exploration de l'arbre par la primitive « coupure »

Le prédicat prédéfini *coupure* permet de contrôler l'exploration de l'arbre de recherche. En effet, lorsque cette coupure (notée *cut*, slash « / » ou plus souvent point d'exclamation « ! ») s'efface, celui-ci est évalué à vrai et tous les choix en attente sont supprimés entre ce nœud et celui où est apparue la coupure.
« ! » signifie donc que la première solution trouvée sur sa gauche suffira, mais que Prolog cherchera toutes les solutions aux prédicats sur la droite de « ! ». Lorsque l'on est certain de l'unicité d'une solution, la coupure empêche les retours en arrière inutiles, voire néfaste, par exemple dans :

min(X, Y, X) :- X <= Y, !.
min(X, Y, Y) :- X > Y.

On pourrait supprimer la condition *X > Y* de la seconde clause au cas où le troisième argument (le résultat du *min*) est inconnu, ce qui est l'utilisation la plus courante, mais alors *min(3, 5, 5).* réussirait, ce qu'il vaut mieux éviter.

Cependant, cela peut convenir dans la mesure où *min* sera utilisé de façon fonctionnelle pour connaître le troisième argument. Aussi, la plupart du temps les clauses suivantes suffiront :

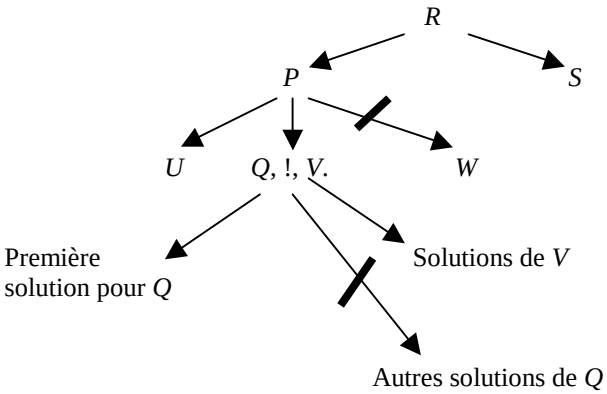
min(X, Y, X) :- X <= Y, !.
min(X, Y, Y).

Prenons maintenant l'exemple d'un programme Prolog où *P, Q, R* sont des propositions. Dans l'arbre de recherche, deux sous-arbres vont être coupés (trait gras).

Là où apparaît la coupure, c'est-à-dire pour trouver les solutions de *P*, la recherche des solutions par *W* est supprimée au cas où une première solution a été trouvée par *Q* et les autres solutions de *Q* ne sont par recherchées.

Par contre, la coupure, telle qu'elle est placée, ne limite pas les solutions de *R*, ce qui fait que même si la clause *R :- S.* était placée plus bas, toutes les solutions de *R* sont recherchées. On demande si *R* est prouvable, il le sera si *P* ou *S* le sont.

R :- P.
R :- S.
P :- U.
P :- Q, !, V.
P :- W.



La description de l'effet de la coupure dans l'arbre de recherche n'est pas facile à donner, aussi, pour un second exemple, on considère quatre programmes ne variant que par la dernière clause p_i mise à la place de *p*. Les axiomes sont :

q(a).
q(b).
q(c).
r(b, b1).
r(c, c1).
r(a, a1).
r(a, a2).
r(a, a3).

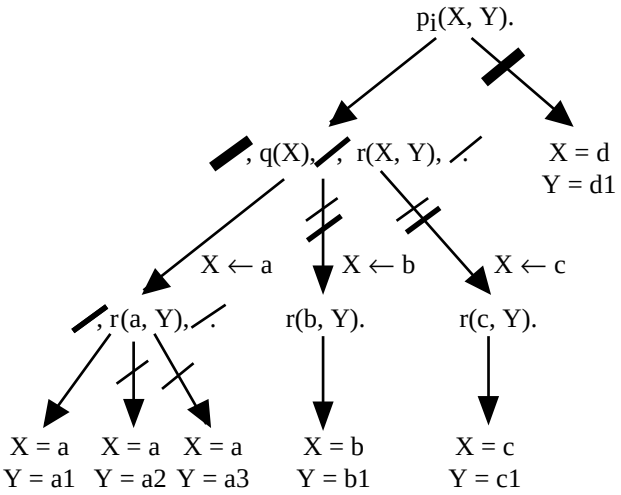
Les règles sont successivement :

p0(X, Y) :- q(X), r(X, Y).
p1(X, Y) :- q(X), r(X, Y), /.
p2(X, Y) :- q(X), //, r(X, Y).
p3(X, Y) :- ///, q(X), r(X, Y).
pi(d, d1).

Avec *i* successivement 0, 1, 2, 3.

Soient donc, à présent, les quatre programmes différents constitués avec les clauses relatives aux prédicats *q*, *r*, *p* avec trois possibilités de positions de la coupure (vues par différentes épaisseurs) pour *p(X, Y).*

Suivant le but $p_i(X, Y)$. demandé on a le schéma :



Pour p_0 les 6 solutions sont données alors qu'il n'y en a qu'une (a, a_1) pour p_1, les 3 (a, a_i) pour p_2, et 5 pour p_3, (d, d_1) n'étant pas regardée sauf pour p_0.

Le prédicat « faux » ou « impasse »

Lorsque c'est le tour du prédicat prédéfini *fail* (l'impasse) d'être effacé, alors la recherche continue en remontant au nœud supérieur. Tout simplement parce que *fail* est toujours évalué à faux. Ce prédicat ne sert quasiment que pour la négation.
Cette négation n'est pas toujours présente dans les prédicats prédéfinis des différentes versions de Prolog ; en ce cas, elle est facile à redéfinir (paragraphe suivant).
Que se passe-t-il, dans l'exemple précédent, si l'on remplace les clauses *p*, successivement par les clauses *p4 p5 p6* correspondant à *p1 p2 p3* dans lesquelles « impasse » prend la place de « coupure » ? La réponse sera *no* dans tous les cas puisque *fail* est évaluée à faux, la différence sera qu'elle sera immédiate pour *p6*, surviendra après recherche des solutions de *q* pour *p5* et après recherche des solutions de *q* et de *r* pour *p4*.

Négation

Le problème qui se pose pour la négation est que le « faux » est remplacé par le concept de « non prouvable ». Cette négation n'est donc qu'une « négation par l'absence », elle peut être simulée par les deux clauses dont l'ordre est indispensable :

```
not(P) :- P, !, fail.
not(P).
```

En effet, la première clause signifie que si *P* est attesté, alors la coupure cessera toute recherche ultérieure et l'impasse *fail* forcera la tête *not(P)* à être fausse. La seconde clause indique que si *P* n'est pas présent dans la base de connaissance, alors *not(P)* est vrai.
Par exemple « différent » (prédéfini par \==) peut être redéfini par :

```
dif(X, X) :- !, fail.
dif(X, Y).
```

Il est également possible de construire un opérateur conditionnel :

```
if(P, Q, R) :- P, !, Q.
if(P, Q, R) :- R.
```

Supposons en effet que *P* soit vérifié, alors, le *if*, sans aller voir la seconde clause, va chercher à résoudre *Q*. Par contre, si *P* n'est pas satisfait, alors Prolog passe à la seconde clause qui l'oblige à résoudre *R*.

Par ailleurs, *not(P)* n'instancie rien, *P* peut être résolu en donnant des solutions s'il contient des variables, alors que *not(P)* sera un succès sans donner d'instanciation. Un exemple extrêmement réduit aidera à comprendre ce mécanisme.

```
etudiant(max).
etudiant(luc).
mineur(jean).
etudiantmajeur(X) :- not(mineur(X)), etudiant(X).
```

La question *etudiantmajeur(luc).* donnera vrai dans la mesure où le fait *mineur(luc)* n'existe pas dans la base, mais la question *etudiantmajeur(X).* ne donnera rien, il n'y a pas d'instanciation pour *X*, sauf si on inverse l'ordre des hypothèses de *etudiantmajeur*, car alors *X = max* puis *X = luc* prouveront d'abord *etudiant(X).*

Voyons maintenant l'effet de la double négation, si on reprend le petit programme :

```
femme(eve).
petit(eve).
femme(irma).
taille(irma, 155).
femme(julie).
taille(julie, 165).
petit(X) :- taille(X, T), T < 160.
```

femme(*X*). → *X* = eve ; *X* = irma ; *X* = julie ; *X* = carmela ; no
not(femme(eve)). → no
homme(eve). → no
not(not(femme(eve))). → yes
petit(*X*). → *X* = eve ; *X* = irma ; *X* = carmela ; no
not(petit(*X*)). → no
not(not(femme(*X*))). → *X* = _1039
not(not(petit(*X*))). → *X* = _1685

On voit qu'il n'y a pas d'instanciation à ces deux dernières questions.

Utilisation de la coupure pour réduire l'espace de recherche

En utilisation pour l'appartenance, la coupure est ici, accessoire, mais empêche d'aller voir si un élément appartenant à une liste va lui appartenir plusieurs fois :

```
app(X, [X | _]) :- !.
app(X, [_ | L]) :- app(X, L).
```

Par contre, comme ce sera le cas de nombreux exemples, notamment pour les algorithmes à essais avec retour en arrière, chercher tous les éléments d'une liste donnée interdit au contraire d'utiliser cette coupure.
Après avoir défini le prédicat *liste* on peut écrire *no(liste(X))* mais aussi définir directement :

```
noliste ([]) :-  !, fail.
noliste ([_ | _]) :-  ! , fail.
noliste(_).
```

1 Construire un instrument répétitif pour toute proposition

On affirme par une première clause que la répétition zéro fois consiste à ne rien faire, et la coupure est précisément là pour empêcher Prolog d'aller voir la seconde clause.

```
repeat(0, _) :- !.
repeat(X, P) :- P, Y is X − 1, repeat(Y, P).
```

> Exemple, on répète 5 fois un affichage :
> repeat(5, write('abc ')).
> → abc abc abc abc abc yes

2 Tête et queue d'une liste
Définir les prédicats *tete(L, N, T)* où *T* est la liste formée des *N* premiers éléments de *L* et *queue(L, N, Q)* où *Q* est la liste des derniers éléments de *L* sauf les *N* premiers.

Nous marquons grâce à des coupures, les deux conditions d'arrêt pour $N = 0$ et lorsqu'il n'y a plus rien à parcourir dans la liste *L*. Dans le cas contraire, nous pourrions avoir des branches infinies dans la recherche, étant entendu qu'ici,

comme dans beaucoup de cas, les relations sont des applications au sens mathématique, seule l'unique solution pour le dernier argument nous intéresse.

```
tete(_, 0, []) :- !.
tete([], _, []) :- !.
tete([X | L], N, [X | T]) :- M is N − 1, tete(L, M, T).

queue(L, 0, L) :- !.
queue([], _, []) :- !.
queue([_ | L], N, Q) :- M is N − 1, queue(L, M, Q).
```

queue([a, b, c, d, e, f, g], 3, *Q*). → *Q* = [d, e, f, g]
tete([a, b, c, d, e, f, g], 3, *T*). → *T* = [a, b, c]
tete([a, b, c, d, e, f, g], 3, *T*), queue(*T*, 2, *Q*). → *T* = [a, b, c] *Q* = [c]

3 Listes disjointes

On voudrait écrire le prédicat indiquant si oui ou non deux listes n'ont pas d'élément commun. Une façon consiste à chercher si tous les éléments *E* de *X* peuvent appartenir à la seconde liste *Y*, si oui, il y a échec.

En utilisant l'appartenance, le prédicat « être disjoints » pour deux listes devra échouer dès qu'on trouve un élément commun et il sera possible « d'arrêter les frais » de recherche grâce à une coupure. La seconde solution est plus élégante.

```
disjoints(L, M) :- not(elcommun(X, L, M)).
elcommun(X, L, M) :- app(X, L), app(X, M).
```

Ou bien :

```
disjoints(X, Y) :- app(E, X), app(E, Y), !, fail.
disjoints(X, Y).
```

Exemples

elcommun(*X*, [a, b, c], [d, e, f]). → no
elcommun(*X*, [a, b, c], [d, c, b]). → *X* = b ; *X* = c ; no
disjoints([a, b, c], [q, w, e, r, t, y]). → yes
disjoints([a, z, e, r, t, y], [q, w, e, r, t, y]). → no

4 Retraits
On reprend cet exercice en voyant toutes les possibilités. Il faut que soit vrai *eff(X, L, R)* où *R* est la liste *L* dans laquelle la première occurrence de *X* est effacée, *eff2* pour effacer toutes les occurrences, puis *eff3* pour effacer à tous les niveaux de *L*.

```
eff(_, [], []).
eff(A, [A | L], L) :- !.
eff(A, [B | L], [B | M]) :- eff(A, L, M).
```

Mais, pour ne pas avoir la situation :

eff(a, [f, a, d, a], X). →*X = [f, d, a] ; X = [f, a, d] ; X = [f, a, d, a] ; no*

il faut rajouter la prémisse *coupure* dans la seconde clause.
Alors *: eff(a, [f, a, d, a], X).* →*X = [f, d, a] ; no*

Pour *eff2*, on remplace la seconde clause par *eff2(A, [A | L], M) :- eff2(A, L, M)*. Mais là aussi *eff2(a, [b, a, b, a], X*). donnerait *X = [b, b]* puis *X = [b, a, b]* puis *X = [b, b, a]* d'où la nécessité de la coupure. Pour « effacer » toutes les occurrences de *A*, il suffit de changer la seconde clause, en indiquant que les effacements se poursuivent :

```
eff2(_, [], []).
eff2(A, [A | L], M) :- eff2(A, L, M), !.
eff2(A, [B | L], [B | M]) :- eff2(A, L, M).
```

Maintenant, pour retirer les occurrences de *X* à tous les niveaux, il faut les retirer en profondeur dans *Y* et en largeur dans *Z*, ce que fait la troisième clause. Cependant il faut veiller à prévoir l'arrêt en largeur, ce que fait la première clause et en profondeur, ce que fait la seconde où on rencontre cet *X* comme une feuille de l'arbre, mais aussi la dernière clause. En effet, dans cette dernière, les coupures qui précèdent obligent cette clause à n'être atteinte que dans le cas où ce qui reste de l'arbre n'est pas une liste mais un atome *A* qui est forcément différent de *X*.

```
eff3(_, [], []).
eff3(X, [X | L], M) :- eff3(X, L, M), !.
eff3(X, [Y | L], [Z | M]) :- eff3(X, Y, Z), eff3(X, L, M), !.
eff3(_, A, A).
```

Exemples

eff3(a, [a, b, [b, a, c], [b, [a, c], b], a], *R*).
→ *R* = [b, [b, c], [b, [c], b]] ; no

Une seule solution, celle où tous les « a » sont « effacés ».
Dans la demande suivante, tout est normal, car rien n'indique que l'on devrait effacer les listes vides :

eff3(a, [a, b, [a, a, a], [a, b, [a, a], b, a], d], *R*). → *R* = [b, [], [b, [], b], d]

5 Substitution
subst(X, Y, L, R) où *R* est le résultat de la liste *L* dans laquelle *X* a été substitué par *Y*. Construire de même le prédicat *sb* de substitution à toutes les profondeurs.

```
subst(_, _, [], []).
subst(X, Y, [X | L], [Y | M]) :- !, subst(X, Y, L, M).
subst(X, Y, [Z | L], [Z | M]) :- subst(X, Y, L, M).
```

subst(a, x, [a, a, x, a, a, x], *R*). → *R* = [x, x, x, x, x, x]
subst(a, x, [a, b, a, c, b, a, a, b, d, c], *R*).
→ *R* = [x, b, x, c, b, x, x, b, d, c]

Sans la coupure, on aurait des solutions parasites dues au fait que *Y* et *Z* peuvent s'unifier dans la troisième clause.
Pour les substitutions à toutes les profondeurs, la première clause constitue la règle générale ; on substitue *X* par *Y* au sein du premier élément *T* de la liste (donc, en profondeur) et dans la queue *Q* de cette liste (donc en largeur). La seconde clause arrête les appels récursifs en largeur, la troisième traitant le cas de l'objet *X* lui-même. La coupure interdisant les solutions parasites que pourrait donner la dernière clause.

```
sb(X, Y, [T | Q], [TS | QS]) :- sb(X, Y, T, TS), sb(X, Y, Q, QS).
sb(_, _, [], []).
sb(X, Y, X, Y) :- !.
sb(X, Y, Z, Z).
```

sb(a, x, [a, [b, a, c], [[b, a, a]], b, [], c], *R*).
→ *R* = [x, [b, x, c], [[b, x, x]], b, [], c]

6 Différence ensembliste
diff(A, B, R) est vrai si la liste *R* est *A* privée des éléments présents dans *B*. En déduire la relation *être égaux en tant qu'ensemble*, c'est-à-dire sans tenir compte de l'ordre ou des répétitions.

Sans la coupure, le problème admet des solutions parasites.

```
diff([], M, []).
diff([X | L], M, R) :-  app(X, M), !, diff(L, M, R).
diff([X | L], M, [X | R]) :-  diff(L, M, R).

app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).

egaux(L, M) :- diff(L, M, []), diff(M, L, []).
```

diff([a, b, c, d, e, f], [c, d, f, h, g, k], *D*). → *D* = [a, b, e] ; no
diff([a, b, c, d], [d, g, h, b], *R*). → *R* = [a, c] ; no
egaux([a, b, c, d], [d, b, b, a, c]). → yes
egaux([a, b, c, d], [d, a, b, a, c, b, d]). → yes
egaux([a, b, c], [c, b, c, a, e]). → no

7 Aplatir une liste
On veut, par exemple, que *aplatir(*[[a, z, e, r], t, y,[u, i, [o, p], q, s], d, f], *R).* renvoie *R* = [a, z, e, r, t, y, u, i, o, p, q, s, d, f] ce qui constitue une suppression des parenthèses intérieures.

```
liste([]).
liste([_|_]).
inv([], L, L).
inv([X | L], T, R) :- inv(L, [X | T], R).

aplatir(L, R) :- apbis(L, [], LA), inv(LA, [], R).
apbis(X, T, [X | T]) :- not(liste(X)).
apbis([X | L], T, R) :- apbis(X, T, XA), apbis(L, XA, R).
apbis([], L, L).

aplat([], []).                  % une autre solution
aplat([X | L], R) :- !, aplat(X, XA), aplat(L, LA), conc(XA, LA, R).
aplat(X, [X]).
```

8 Eléments de trois en trois
Ecrire, sans la coupure, les clauses nécessaires pour obtenir la liste des éléments de 3 en 3 dans une liste.

```
sl3([X, Y, Z | L], [X | R]) :- sl3(L, R).
sl3([X, _ ], [X]).
sl3([X], [X]).
sl3([], []).
```

> sl3([a, b, c, d, e, f, g, h, i, j, k, l, m, n], *L*). → *L* = [a, d, g, j, m]
> sl3([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], *L*). → *L* = [0, 3, 6, 9]

9 Eléments de *N* en *N*
Ecrire plus généralement, avec les prédicats d'égalité, d'affectation et la coupure, les clauses nécessaires pour obtenir la liste des éléments de *N* en *N* dans une liste.

```
sl(N, K, [X | L], [X | R]) :- (K = N ; K = 0), M is N − 1, sl(N, M, L, R), !.
sl(N, K, [X | L], R) :- M is K − 1, sl(N, M, L, R), !.
sl(_, _, [], []).
sl(N, L, R) :- sl(N, N, L, R).
```

Ou encore avec un prédicat de retrait des *N* premiers éléments d'une liste :

```
sl(_, [], []).
sl(N, [X | L], [X | R]) :- M is N − 1, supr(M, L, LL), sl(N, LL, R).
supr(_, [], []).
supr(0, L, L).
supr(N, [X | L], R) :- !, M is N − 1, supr(M, L, R).
```

> sl(4, [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t], *L*).
> → *L* = [a, e, i, m, q]
>
> sl(5, [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t], *L*).
> → *L* = [a, f, k, p]
>
> sl(6, [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t], *L*).
> → *L* = [a, g, m, s]

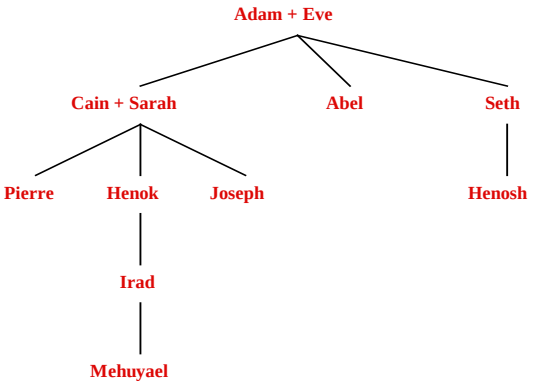
10 Automate fini reconnaissant les mots (ab)
Un automate fini est un ensemble d'états, un alphabet ici réduit à A = {a, b} et un ensemble de « transitions » qui, à chaque état et lettre de l'alphabet, peut faire passer à un autre état. Ici, seules deux transitions sont possibles. Un mot sera « reconnu » s'il permet de faire passer d'un état initial à un état terminal, ici, le même $q0$.

```
init(q0).                terminal(q0).
trans(q0, a, q1).        trans(q1, b, q0).
reconnu(L) :- init(Q), accessible(Q, L), fini(L, S), terminal(S).
accessible(_, []).
accessible(Q, [X | L]) :- trans(Q, X, S), accessible(S, L).
fini([X], S) :- trans(_, X, S), !.
fini([_ | L], S) :- fini(L, S).
```

reconnu([a, b, a, b, a, b]). → yes
reconnu([a, b, a]) . → no
reconnu(*X*). → *X* = [a, b] ; *X* = [a, b, a, b] ; *X* = [a, b, a, b, a, b] ; *X* = [a, b, a, b, a, b, a, b] ; etc

11 Définir le degré de parenté
On peut imaginer beaucoup de relations comme *frere(X, Y)* vérifiée si *X* est un frère de *Y* (donc *X* est un homme), *oncle(O, X)*, *demifrere(X, Y)*, puis surtout les relations de parenté au degré *n*, sachant qu'il s'agit de la distance dans l'arbre généalogique, ainsi père et fils sont au degré 1, frères et sœurs au degré 2, oncle et neveu au degré 3, etc.

```
epoux(adam, eve).
epoux(cain, sarah).
mere(eve, abel).
mere(eve, seth).
pere(adam, cain).
pere(adam, sarah).
pere(seth, henosh).
pere(cain, pierre).
pere(cain, henok).
pere(cain, joseph).
pere(henok, irad).
pere(irad, mehuyael).
```




```
pere(P, E) :- epoux(P, M), mere(M, E).
```

parent(P, X) :- pere(P, X).
parent(M, X) :- mere(M, X).
branche(X, X, 0).
branche(V, E, N) :- parent(P, E), branche(V, P, M), N is M + 1.
degre(X, Y, D) :- branche(A, X, N), branche(A, Y, M),!, D is N + M.

La coupure est nécessaire pour ne pas aller chercher d'ancêtre commun plus haut que le premier ; en effet, c'est la plus courte distance que l'on souhaite obtenir.

branche(adam, irad, *N*). → *N* = 3
branche(*X*, irad, *Y*). → *X* = irad *Y* = 0 ; *X* = henok *Y* = 1 ;
X = cain *Y* = 2 ; *X* = adam *Y* = 3
degre(irad, seth, *D*). → *D* = 4
degre(pierre, joseph, *X*). → *X* = 2
degre(mehuyael, joseph, *X*). → *X* = 4
degre(adam, irad, *X*). → *X* = 3
degre(mehuyael, adam, *X*). → *X* = 4
degre(irad, henosh, *X*). → *X* = 5

12 Tri par insertion
Le premier élément d'une liste étant mis de côté, le reste est trié (suivant la même méthode), puis l'élément est inséré à sa place.

tri([X | L], R) :- tri(L, LT), insert(X, LT, R).
tri([], []).

insert(X, [], [X]).
insert(X, [Y | L], [X, Y | L]) :- X < Y, !.
insert(X, [Y | L], [Y | M]) :- insert(X, L, M).

insert(4, [0, 1, 2, 3, 5], *R*). → *R* = [0, 1, 2, 3, 4, 5]
insert(0, [2, 3], *R*). → *R* = [0, 2, 3]
insert(5, [1, 2, 3], R). → *R* = [1, 2, 3, 5]
tri([1, 5, 6, 2, 3, 8, 0, 4, 1, 9, 2, 5], *L*).
→ *L* = [0, 1, 1, 2, 2, 3, 4, 5, 5, 6, 8, 9]
En modifiant « < » en « @< », il est possible de trier des chaînes de caractères et d'obtenir ce dernier exemple :
tri([christelle, tauty, marie, anne, sophie], *R*).
→ *R* = [anne, christelle, marie, sophie, tauty]

13 Tri par fusion
La liste à trier est séparée en deux, ces deux parties sont triées suivant le même algorithme, puis fusionnées. On peut construire et utiliser un prédicat de « scission » consistant à parcourir une liste en rangeant alternativement ses éléments en deux listes *P* et *I* suivant les rangs pair et impair.

```
trifus([], []).
trifus([X], [X]).
trifus(L, R) :- scission(L, G, D), trifus(G, GT), trifus(D, DT), fusion(GT, DT, R).

fusion([], L, L).
fusion(L, [], L).
fusion([X | P], [Y | Q], [X | R]) :- X < Y, !, fusion(P, [Y | Q], R).
fusion([X | P], [Y | Q], [Y | R]) :- fusion([X | P], Q, R).
```

Très directement, on peut définir :

```
scission([], [], []).
scission([X], [], [X]).
scission([X, Y | L], [Y | P], [X | I]) :- scission(L, P, I).
```

Il est possible de simplifier « fusion » en :

```
fusion(L, [], L).
fusion([X | P], [Y | Q], [X | R]) :- X < Y, !, fusion(P, [Y | Q], R).
fusion(L, [Y | Q], [Y | R]) :- fusion(L, Q, R).
```

Exemples

scission([a, b, c, d, e, f, g, h, i, j, k], *P*, *I*).
→ *P* = [b, d, f, h, j] *I* = [a, c, e, g, i, k]
fusion([0, 1, 2, 3, 5], [1, 2, 2, 4, 5, 6], *R*).
→ *R* = [0, 1, 1, 2, 2, 2, 3, 4, 5, 5, 6]
trifus([6, 5, 8, 0, 2, 3, 7, 5], *L*).
→ *L* = [0, 2, 3, 5, 5, 6, 7, 8]

Toujours avec « @< », il est possible de trier des chaînes de caractères :

tri([viviane, marie-therese, sylvie, marie-francoise, kylia, marie-christine, corinne, marie-renee, akima, christine], *R*).
→ *R* = [akima, christine, corinne, kylia, marie-christine, marie-francoise, marie-renee, marie-therese, sylvie, viviane]

14 Tri-pivot
Un élément (pivot) est choisi dans la liste, puis tous les éléments sont versés dans deux listes *G, D* suivant qu'ils sont inférieurs ou supérieurs au pivot. Ces listes sont triées de la même façon puis concaténées de part et d'autre du pivot.

```
partition([X | L], P, [X | G], D) :- X =< P,!, partition(L, P, G, D).
partition([X | L], P, G, [X | D]) :- partition(L, P, G, D).
partition([], _, [], []).

triseg([], []).
triseg([P| L], R) :- partition(L, P, G, D), triseg(G, GT), triseg(D, DT),
                     conc(GT, [P| DT], R).
```

Retirer la coupure obligerait à rajouter $X > P$. Le pivot P sera, en l'absence de toute information sur le désordre de la liste, le premier élément. Mais on peut faire mieux, pour se dispenser de la concaténation, le second argument étant un tampon, le troisième, le résultat :

```
tri(LD, LT) :- tribis(LD, LT, []).
tribis([], L, L).
tribis([P | R], LT, LR) :- partition(R, P, G, D),
                           tribis(G, LT, [P | Q]), tribis(D, Q, LR).
```

> partition([2, 3, 0, 5, 4, 2, 7, 6, 1], 5, *G*, *D*).
> → *G* = [2, 3, 0, 5, 4, 2, 1] *D* = [7, 6]
> tri([2, 5, 0, 3, 9, 4, 7, 6, 3, 5, 1, 2, 8, 0], *X*).
> → *X* = [0, 0, 1, 2, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9]

15 Un tri stupide
Le tri de *L* doit être une permutation de *L* qui est ordonnée.

```
tri(L, LT) :- permut(L, LT), ordonne(LT), !.
permut([], []).    % indique que les deux arguments sont des permutations
permut([X|L], [Y|M]) :- ret(Y, [X | L], N), permut(N, M).
ordonne([]).       % indique si la liste est ordonnée ou non
ordonne([_]).
ordonne([X, Y | L]) :- X =< Y, ordonne([Y | L]).
ret(X, [X | L], L).
ret(X, [Y | L], [Y | R]) :- ret(X, L, R).
```

> tri([5, 7, 5, 1, 4, 6, 8], *L*). → *L* = [1, 4, 5, 5, 6, 7, 8]

16 Racine carrée entière
On cherche le plus grand entier dont le carré est inférieur ou égal à un entier positif donné. (Les fonctions *sqrt* et *round* existent dans la plupart des Prolog).

C'est un exemple « d'itération », à savoir que la solution voulue *M* doit avoir un carré *P* inférieur ou égal à *N*. Pour cela, *M* est initialement 1, puis est « incrémenté » sous le nom de *MS*. Le résultat *Q* est $M - 1$ dès que $P = M^2$ a dépassé *N*, car c'est le précédent qui était le bon.

```
sqrt(N, Q) :- sqrbis(N, 1, Q).
sqrbis(N, M, Q) :- P is M*M, N < P, !, Q is M – 1.
sqrbis(N, M, Q) :- MS is M + 1, sqrbis(N, MS, Q).
```

sqrt(49, *Q*). → *Q* = 7
sqrt(48, *Q*). → *Q* = 6

17 Puissance et racine *N*-ième entière d'un entier *X*
Reprendre la même démarche qu'en 16

```
puiss(1, X, X) :- !.
puiss(N, X, P) :- K is N – 1, puiss(K, X, PS), !, P is PS*X.
rac(N, X, Q) :- racbis(N, X, 1, Q).
racbis(N, X, M, Q) :- puiss(N, M, P), X < P, !, Q is M – 1.
racbis(N, X, M, Q) :- MS is M + 1, racbis(N, X, MS, Q).
```

puiss(5, 3, *P*). → *P* = 243
puiss(9, 2, *P*). → *P* = 512
rac(4, 81, *Q*). → *Q* = 3
rac(4, 2401, *Q*). → *Q* = 7

18 Nombres parfaits
Il s'agit des entiers égaux à la somme de leurs diviseurs propres comme 28 = 1 + 2 + 4 + 7 + 14.

```
interval(1, [1]) :- !.
        % Produit l'intervalle ordonné décroissant des entiers de N à 1
interval(N, [N | L]) :- M is N – 1, interval(M, L).
som([], 0).         % calcule la somme des éléments d'une liste
som([X | L], S) :- som(L, SP), S is SP + X.
```

```
divprop(_, [], R, R).
divprop(N, [D | L], [D | LD], R) :-  0 is N mod D, !, divprop(N, L, LD, R).
divprop(N, [_ | L], LD, R) :- divprop(N, L, LD, R).
parfait(N) :- M is N // 2, interval(M, I), divprop(N, I, LD, []), som(LD, N).
chercheparfait :- cherch(6).
cherch(N) :- NS is N + 1, parfait(N), !, write(N), write(' est parfait '), nl,
cherch(NS).
cherch(N) :- NS is N + 1, N < 10000, cherch(NS).
```

> chercheparfait. → 6 est parfait 28 est parfait 496 est parfait 8128 est parfait (en 24 s)

19 Entiers premiers entre eux et indicateur d'Euler

L'indicateur d'Euler d'un entier N est le nombre E d'entiers inférieurs à N premier avec lui. S'il se décompose en facteurs premiers $N = p^{\alpha}q^{\beta}r^{\gamma}\ldots$, on montre que son indicateur d'Euler est $\phi(N) = p^{\alpha-1}(p-1)q^{\beta-1}(q-1)r^{\gamma-1}(r-1)\ldots$

Deux entiers sont premiers entre eux s'ils n'ont pas de diviseur commun (autre que 1) ; on examine donc un diviseur potentiel *D* pour les deux nombres *A* et *B*. Dès qu'il y a divisibilité, la deuxième clause de *aux* répond *faux*. Sinon, on examine *D* à partir de 2, tant qu'il est inférieur à *A* et à *B*. Dans *eulerbis*, *A* est un entier de 2 à *N* – 1 et *B* le compteur d'entiers premiers avec *N*. Quand la suite des recopies est achevée, la valeur de *E* est donc *B*.

```
premiers(A, B) :- aux(A, B, 2).
aux(A, B, D) :- A < D, B < D.
aux(A, B, D) :- 0 is A mod D, 0 is B mod D, !, fail.
aux(A, B, D) :- DS is D + 1, aux(A, B, DS).
euler(N, E) :- eulerbis(N, 2, 1, E).
eulerbis(N, A, E, E) :- N =< A, !.
eulerbis(N, A, B, E) :- premiers(N, A), !, AS is A + 1,
                        BS is B + 1, eulerbis(N, AS, BS, E).
eulerbis(N, A, B, E) :- AS is A + 1, eulerbis(N, AS, B, E).
```

> premiers(12, 16). → no
> premiers(18, 49). → yes
> premiers(512, 81). → yes
> euler(17, *E*). → *E* = 16
> euler(12, *E*). → *E* = 4
> euler(91, *E*). → *E* = 72

20 Crible d'Eratosthène
En prenant la liste des entiers de 2 à *N*, on passe au « crible », c'est-à-dire on supprime tous les multiples du premier de la liste (d'où son nom), ainsi les multiples de 2 sont retirés, puis ceux du premier restant 3, puis ceux du premier restant 5, etc.

Dans les clauses du prédicat *retmul*, *P* représente le premier entier à conserver, dont chaque multiple *M* et son suivant *MS* sont à retirer de la liste *L*. *PS* est le premier suivant et *LP* désigne la liste *L* sans les multiples de *P*.

```
retmul(_, [], _, []).
retmul(P, [M | L], M, R) :- !,  MS is M + P, retmul(P, L, MS, R).
retmul(P, [X | L], M, [X | R]) :- X < M, !, retmul(P, L, M, R).
retmul(P, [X | L], M, [X | R]) :- MS is M + P, retmul(P, L, MS, R).

eratos(N, R) :- compte(N, [], C), crible(C, 2, R).

compte(X, L, C) :- X > 1, !, Y is X – 1, compte(Y, [X | L], C).
compte(_, C, C).

crible([], _, []).
crible([P | L], P, [P | R]) :- M is 2*P, retmul(P, L, M, CS), crible(CS, PS, R).
```

compte(19, [], *C*).
→ *C* = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

retmul(2, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], 2, *R*).
→ *R* = [3, 5, 7, 9, 11, 13, 15, 17]
retmul(3, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], 9, *R*).
→ *R* = [2, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14, 16, 17]
% retire les multiples de 3 à partir de 9

eratos(12, *R*). → *R* = [2, 3, 5, 7, 11]
eratos(64, *R*). →
R = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]

21 Profondeur d'un arbre
Ecrire les clauses nécessaires au calcul de la profondeur d'un objet écrit correctement avec des crochets (il s'agit en fait de dénombrer le nombre de niveaux de crochets emboîtés).

C'est un des premiers exemples très intéressants où l'exploration d'une liste non régulière doit se faire non seulement en largeur, mais en profondeur dans les niveaux de parenthèses.
La première clause de *prof* stoppe la recherche en largeur, la dernière la stoppe en profondeur.
La seconde indique récursivement qu'il faut chercher dans le premier élément *X*, comme dans la queue *L*.

```
max(A, B, B) :- A < B, !.
max(A, B, A).

prof([], 1).
prof([X | L], N) :- prof(X, XP), prof(L, LP), P is XP + 1, max(LP, P, N),!.
prof(_, 0).
```

> prof(a, *N*). → *N* = 0
> prof([i, i, e, j], *N*). → *N* = 1
> prof([a, [b], [c, [d, [b, [], c]], [e], a]], *N*). → *N* = 5

22 Nombre de feuilles d'un arbre
C'est le nombre d'atomes d'une structure écrite avec des crochets.

```
nbfeuilles([X | L], N) :- nbfeuilles(X, NX), nbfeuilles(L, NL), N is NX + NL, !.
nbfeuilles([], 0) :- !.
nbfeuilles(_, 1).
```

Exemples

> nbfeuilles(i, *N*). → *N* = 1
> nbfeuilles([x, m, l], *N*). → *N* = 3
> nbfeuilles([[e, n, s, i, i, e], e, n, s, i, i, e, [e, n, s, i, i, e]]], *N*). → *N* = 18
> nbfeuilles([c, o, r, i, n, n, e], *N*). → *N* = 7
> nbfeuilles([a, [b], [c, [d, [b, [], c]], [e], a]], *N*). → *N* = 8

23 Somme totale figurant dans une liste
Calculer la somme de tous les éléments entiers figurant dans une liste.

Problème tout à fait analogue, la coupure permet de considérer que la troisième clause ne sera regardée que pour un premier argument qui n'est pas une liste comme dans les deux exercices précédents.

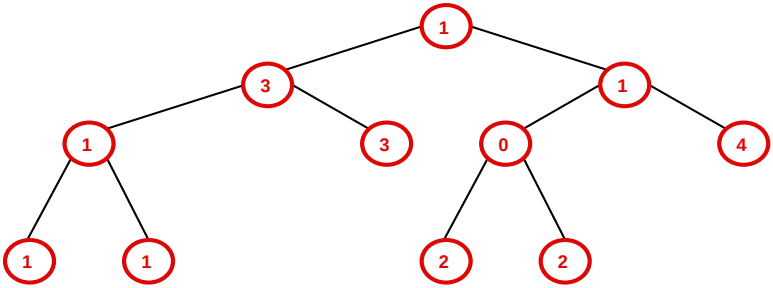
```
nb([X | L], N) :- nb(X, NX), nb(L, NL), N is NX + NL, !.
nb([], 0) :- !.
nb(X, X).
```

> nb([2, [4, [5, 0], 1], [[2]], 0, [1, 2], 5], *N*). → *N* = 22

24 Mobile suspendu
Il s'agit d'un arbre binaire étiqueté où de chaque côté figure un entier ou un mobile. Ils seront représentés par une liste [poids-central, mobile gauche, mobile-droit]. Donner les clauses d'un prédicat permettant de calculer le poids total d'un mobile, seulement dans le cas où il est équilibré.
On pourra se servir du prédicat prédéfini *integer(X)* qui est vrai si et seulement si *X* est un entier, mais il est possible de s'en passer avec une coupure bien située.

```
poids([M, MG, MD], P) :- !, poids(MG, PG), poids(MD, PG), P is 2*PG + M.
poids(M, M).
```

> poids([2, [3, 2, 2], [1, 2, 3]], *P*). → no
> poids([2, [3, 2, 2], [1, [1, 1, 1], 3]], *P*). → *P* = 16
> poids([1, [3, [1, 1, 1], 3], [1, [0, 2, 2], 4]], *P*). → *P* = 19
> schéma ci-dessous :



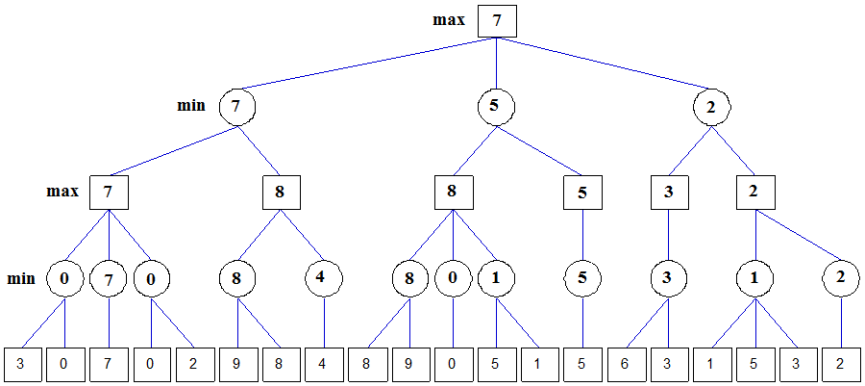
25 L'algorithme du minimax

Cet algorithme s'applique pour tout jeu où deux joueurs choisissent un coup parmi plusieurs coups possibles et ceci à tour de rôle. Les chances d'un « état » du jeu sont évaluées par une certaine fonction heuristique mesurée positivement lorsqu'il favorise un des joueurs (nommé *max*) et négativement pour son adversaire (nommé *min*).

L'algorithme consiste à faire un choix sur la plus grande valeur d'une fonction « *minimax* » lorsque c'est à *max* de jouer (car c'est ce qu'il jouerait pour gagner ou perdre le moins) et un choix sur la plus petite (algébriquement, car c'est ce que *min* jouerait pour perdre le moins) lorsque c'est à *min* de jouer.

Cette fonction qui doit calculer la valeur espérée pour un joueur grâce à un arbre fini de coups à jouer, est définie récursivement par ce qui vient d'être dit, et par l'heuristique lorsqu'elle arrive sur un état terminal ou au niveau de profondeur fixé à l'avance.

Dans l'exemple du schéma ci-dessous, les nœuds dessinés en ronds correspondent au joueur *min* et les carrés au joueur *max*. En commençant par la ligne du bas, il est très facile de remonter jusqu'au sommet.



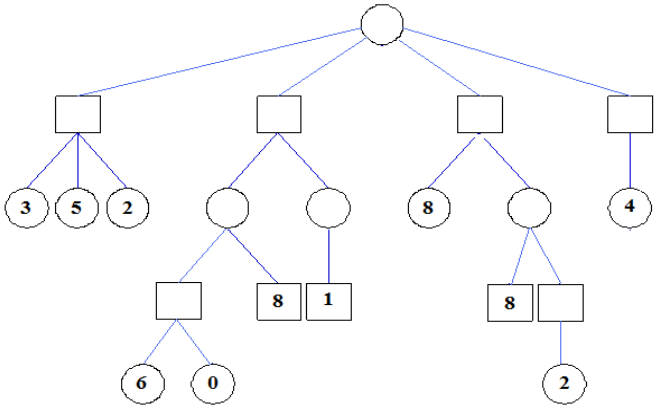
Concernant la représentation de l'arbre, il est commode que chaque crochet ouvrant corresponde à une descente et chaque crochet fermant à une remontée. Prenons l'exemple plus simple de la liste [[[1, 2], [3, 5, 2]], [[3, 4], [[1]]] où *max* est au niveau de la racine ; comme l'arbre est de profondeur 3, c'est *min* qui se trouve encore au niveau des feuilles.

En admettant que le prédicat prédéfini *integer(X)* est vrai si et seulement si *X* est un entier et que les prédicats ternaires *min* et *max* déjà vus existent (sinon, on les réécrit), on va définir un prédicat unique *minimax* qui donnera l'entier résultat *R* pour un nom de joueur *N* et un arbre *A* représenté comme ci-dessus.

Si on convient que les valeurs sont entières et uniquement entre 0 et 100, alors ces dernières valeurs peuvent être prises comme éléments neutres respectivement pour *max* et *min*.

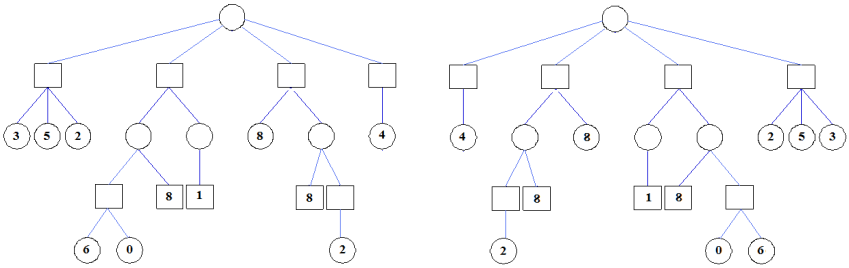
```
min(A, B, A) :- A < B, !.
min(_, B, B).
max(A, B, A) :- A > B, !.
max(_, B, B).

minimax(_, R, R) :- integer(R).
minimax(max, [], 0).
minimax(min, [], 100).

minimax(max, [T | Q], R) :- minimax(min, T, X), minimax(max, Q, Y),
                            max(X, Y, R).
minimax(min, [T | Q], R) :- minimax(max, T, X), minimax(min, Q, Y),
                            min(X, Y, R).
```

> minimax(max, [[[1, 2], [3, 5, 2]], [[3, 4], [1]]], *R*). → *R* = 2
>
> Et pour le premier exemple :
> minimax(max, [[[[3, 0], [7], [0, 2]], [[9, 8], [4]]], [[[8, 9], [0], [5, 1]], [[5]]], [[[6, 3]], [[1, 5, 3], [[2]]]]], *R*).
> → *R* = 7
> minimax(min, [[3, 5, 2], [[[6, 0], [8]], [[1]]], [8, [8, [2]]], [[4]]], *R*).
> → *R* = 4 % c'est l'exemple ci-dessous



Naturellement, comme dans ce troisième exemple, l'arbre peut être irrégulier c'est-à-dire non seulement avec un nombre de fils différent à chaque nœud mais avec des branches de longueurs différentes, comme dans tout jeu où la partie peut se terminer plus ou moins rapidement.
On vérifiera que *min* possède la valeur 4 pour ce dernier exemple.

26 Opération miroir
On cherche à décrire l'opération qui renvoie le symétrique d'un arbre comme dans le schéma ci-dessous.



miroir(A, AM) :- miroir(A, [], AM).
miroir([T | Q], A, R) :- !, miroir(T, [], TM), miroir(Q, [TM | A], R).
miroir([], R, R) :- !.
miroir(X, _, X).

Cette dernière clause indique qu'un élément qui n'est pas une liste est son propre miroir.

Comme pour le miroir d'une liste et bien d'autres problèmes, il est possible de prendre le même nom de prédicats pour deux prédicats d'arités différentes.

> miroir([a, b, c, d, e, f], *R*). → *R* = [f, e, d, c, b, a]
>
> miroir([3, 5, [1, 6], 2], *R*). → *R* = [2, [6, 1], 5, 3]
>
> miroir([[3, 5, 2], [[[6, 0], [8]], [[1]]], [8, [8, [2]]], [[4]]], *R*).
> → *R* = [[[4]], [[[2], 8], 8], [[[1]], [[8], [0, 6]]], [2, 5, 3]]

27 Nombre de dérangements
Soit *derang*(*n*) le nombre de permutations de *n* objets n'ayant aucun point fixe, ce qui signifie par exemple pour $n = 3$ que si abc, acb, bac, bca, cab, cba sont les 6 permutations de a, b, c, en revanche, seules les permutations bca et cab sont des dérangements. On calcule que pour une somme allant de $p = 0$ à n, $derang(n) = n\,!\,\Sigma(-1)^p / p\,! = \Sigma(-1)^p\, n(n-1)(n-2)\ldots(p+1)$.
La formule de récurrence facile à trouver est :
$derang(n) = n.derang(n-1) + (-1)^n$ et les premières valeurs sont :
$d_0 = 1$, $d_1 = 0$, $d_2 = 1$, $d_3 = 2$, $d_4 = 9$, $d_5 = 44$, $d_6 = 265$.

Prendre la formule de récurrence telle quelle n'est pas très astucieux, il est plus intéressant d'avoir un jeu de réécritures où le signe *S* passe à *–S*, l'indice *K* allant de 0 à *N* passe donc de *K* à *K* + 1, et le coefficient *D* passe à (*K* + 1)*D* + *S* en suivant la formule de récurrence. Ce coefficient démarre à 1 et sera le résultat *R* lorsque *K* = *N* (seconde clause).

```
derang(N, R) :- derbis(N, 0, 1, –1, R).
derbis(N, N, R, _, R) :- !.
derbis(N, K, D, S, R) :- KS is K + 1, DS is KS*D + S, SS is –S,
                         derbis(N, KS, DS, SS, R).
```

derang(0, *R*). → *R* = 1
derang(4, *R*). → *R* = 9
derang(5, *R*). → *R* = 44

28 Suites fibonaciennes généralisées
Soit la suite fibonaccienne d'ordre 3 du chapitre 1 définie par les trois premiers termes $u_0 = 2$, $u_1 = -3$, $u_2 = -3$ et la relation de récurrence :
$u_{n+3} = 4u_{n+2} - u_{n+1} - 6u_n$.
On veut généraliser aux suites fibonaciennes définies par une liste *LI* des premiers termes et une liste *LC* de même longueur, des coefficients de la combinaison linéaire qui définit la relation de récurrence.

Prenons d'abord l'exemple où trois termes initiaux sont donnés, ils sont initialisés à l'envers. Puis la liste des coefficients de l'exemple pour lequel le quatrième terme est calculé par : $u_3 = 4(-3) - 1(-3) - 6*2 = -21$

```
init([–3, –3, 2]).
coef([4, –1,–6]).
```

```
u(N, R) :- init(LI), aux1(LI, [], N, R).
aux1([X | L], LI, N, R) :- aux1(L, [X | LI], N, R).
aux1([], L, N, R) :- aux2(L, N, R).

aux2([R | _], 0, R) :- !.
aux2([X | L], N, R) :- M is N − 1, aux2(L, M, R).
```

Le prédicat *aux2* renvoie l'élément *R* de rang *N* à partir de la droite dans la liste initiale et si *N* est trop grand, alors *u(N, R)* va lancer le calcul des suivants

```
u(N, R) :- init(LI), coef(LC), length(LC, LG), prev(LG, N, LI, LI, LC, 0, R).
```

LP est la liste des termes précédents, *LC* celle des coefficients, le terme suivant *X* est calculé par étapes, c'est donc le résultat *R* à la fin, *R = X* quand on a épuisé la liste des coefficients.

```
prev(LG, N, [R | _], _, _, _, R) :- N < LG, !.
prev(LG, N, LU, [U | LP], [C | LC], X, R) :-
        Y is X + U*C, prev(LG, N, LU, LP, LC, Y, R).

% cette clause exécute le terme suivant X que la clause suivante range dans LU

prev(LG, N, LU, _, [], X, R) :-
        coef(LC), M is N − 1, prev(LG, M, [X | LU], [X | LU], LC, 0, R).
```

u(2, *X*). → *X* = –3
u(3, *X*). → *X* = –21
u(4, *X*). → *X* = –63
u(5, *X*). → *X* = –213
u(6, *X*). → *X* = –663
u(7, *X*). → *X* = –2061

En changeant les données, on initialise les premiers termes à l'envers et les coefficients de la combinaison linéaire, par exemple :

```
init([3, 1, 2, –1, 5, 0, 1]).
coef([5, –1, –2, 3, 1, –2, 1]).
```

On obtient u(8, *X*). → *X* = 55

29 Problème du voyageur de commerce
a) Des villes : Paris, Belmopan (sortez votre atlas), Ouagadougou... sont reliées par des « arcs » à sens unique pour l'instant.
b) Définir le prédicat *chemin* puis, en introduisant des distances, le modifier de *X* à *Y* par le chemin *L* de longueur *N*.
c) Introduire des boucles et définir des chemins sans points doubles.

On aura bien sûr un programme composé d'abord de 10 faits du type :

arc(paris, tirana).
arc(oulan-bator, kuala-Lumpur).
arc(paris, reykjavik).
arc(tirana, oulan-bator).
arc(oulan-bator, kuala-lumpur).
arc(paris, oulan-bator).
arc(belmopan, reykjavik).
arc(belmopan, paris).
arc(belmopan, ouagadougou).
arc(paris, ouagadougou).
arc(ouagadougou, kuala-lumpur).

Ne pas oublier de mettre des minuscules à ces villes qui sont des constantes, puis on définira un chemin comme une liste de villes telles que deux villes consécutives soient des *arcs* :

chemin(X, Y, [X, Y]) :- arc(X, Y).
chemin(X, Y, [X | L]) :- arc(X, Z), chemin(Z, Y, L).

a) On pourra alors poser les buts :
Comment se rendre de Belmopan à Kuala-Lumpur ? Quels sont les chemins arrivant à Reykjavik, etc. ?

chemin(belmopan, kuala-lumpur, *X*).
→ *X* = [belmopan, paris, tirana, oulan-bator, kuala-lumpur] ;
X = [belmopan, paris, oulan-bator, kuala-lumpur] ;
X = [belmopan, paris, ouagadougou, kuala-lumpur] ; no

chemin(*V*, reykjavik, *L*).
→ *V* = paris *L* = [paris, reykjavik] ;
V = belmopan *L* = [belmopan, reykjavik] ;
V = belmopan *L* = [belmopan, paris, reykjavik] ; no

b) On peut introduire des distances ou des coûts en remplaçant les clauses *arc* par une relation donnant en plus la distance, *dist(A, B, 5).* où *A, B,* ... sont des villes, par exemple, et en remplaçant *chemin* par :

```
dist(paris, tirana, 10).
dist(oulan-bator, kuala-lumpur, 25).
dist(paris, reykjavik, 15).
dist(paris, oulan-bator, 30).
dist(tirana, oulan-bator, 25).
dist(belmopan, reykjavik, 20).
dist(belmopan, paris, 30).
dist(belmopan, ouagadougou, 35).
dist(paris, ouagadougou, 15).
dist(ouagadougou, kuala-lumpur, 20).
dist(oulan-bator, kuala-lumpur, 15).
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).

route(X, Y, [X, Y], N) :- dist(X, Y, N).
route(X, Y, [X | L], N) :- dist(X, Z, K), route(Z, Y, L, M) , N is K + M.
```

Les chemins issus de *A* de longueur < 22 seront demandés par :
route(belmopan, *V, L, N*), *N* < 22.
→ *V* = reykjavik *L* = [belmopan, reykjavik] *N* = 20 ; no

Les chemins de Paris à *F* ne passant pas en Tirana, par :
route(paris, *F, L, N*), not(app(tirana, *L*)).
→ *F* = reykjavik *L* = [paris, reykjavik] *N* = 15 ;
F = oulan-bator *L* = [paris, oulan-bator] *N* = 30 ;
F = ouagadougou *L* = [paris, ouagadougou] *N* = 15 ;
F = kuala-lumpur *L* = [paris, oulan-bator, kuala-lumpur] *N* = 45 ;
F = kuala-lumpur *L* = [paris, ouagadougou, kuala-lumpur] *N* = 35 ; no

Mais ceci a l'inconvénient de construire *L* pour ensuite vérifier si *D* y est présent ou non, il vaut mieux construire un autre prédicat *route-sans(X, Y, Z, L, N).*

c) S'il y a des boucles comme par exemple un arc de Ouagadougou vers Belmopan, on peut définir :

```
arc(ouagadougou, belmopan).
chemin-injectif(X, Y, [X, Y], _) :- arc(X, Y).
                                    % L sert de tampon, R sera le résultat
chemin-injectif(X, Y, [X | R], L) :- arc(X, Z), not(app(Z, L)),
                                    chemin-injectif(Z, Y, R, [Z | L]).
```

La question (qui donne une infinité de solutions)
chemin(belmopan, tirana, *C*).
→ *C* = [belmopan, paris, tirana] ; *C* = [belmopan, paris, ouagadougou, belmopan, paris, tirana] ; et ainsi de suite...

Par contre, respectivement une et deux solutions :
chemin-injectif(belmopan, tirana, *C*, [belmopan]).
→ *C* = [belmopan, paris, tirana] ; no

chemin-injectif(paris, reykjavik, *C*, [paris]).
→ *C* = [paris, reykjavik] ; *C* = [paris, ouagadougou, belmopan, reykjavik] ; no

30 Les cruches
Disposant de deux cruches, la première pleine contenant 5 litres et la seconde, vide, de capacité 2 litres, on voudrait savoir comment obtenir la première vide et la seconde avec 1 litre exactement. Les opérations possibles sont seulement de vider entièrement une des cruches, ou bien de transvaser l'une dans l'autre de façon à la vider ou à l'emplir entièrement.

On appelle a et b les deux cruches, on définit leur capacité, et un prédicat *etat* rendant compte du temps *T* et des emplissages à cet instant. L'état à l'instant *T* est réécrit en un autre état possible à l'instant suivant *TS* = *T* + 1 dans l'ordre chronologique. Ce qui explique que l'état final demandé soit placé en première clause avec une coupure pour arrêter la recherche, et que le départ se fasse avec l'état initial. Ce genre de problème est vu de différentes façons dans les chapitres suivants. Ici, nous présentons une solution très simple où le détail des actions à effectuer est placé à chaque fois dans une liste *H* appelée « historique ».

On remarquera l'emploi des « , » et « ; » pour décrire les conditions par des conjonctions et disjonctions.

```
capacite(a, 5).
capacite(b, 2).

etat(T, 0, 1, H) :- !, write(T), write(' mouvements '), inverse(H, HI), write(HI).
etat(T, X, Y, H) :- Y > 0, TS is T + 1, etat(TS, X, 0, ['vider b' | H]).
etat(T, X, Y, H), X > 0 :- TS is T + 1, etat(TS, 0, Y, ['vider a' | H]).

etat(T, X, Y, H) :- X > 0, capacite(b, N), Y < N,
        ((X =< N − Y, XS is 0, YS is X + Y);
        (X > N − Y, YS is N, XS is X − N + Y)),
        TS is T + 1, etat(TS, XS, YS, ['transvaser a dans b' | H]).
etat(T, X, Y, H) :- Y > 0, capacite(a, N), X < N,
        ((Y =< N − X, YS is 0, XS is X + Y);
        (Y > N − X, XS is N, YS is Y − N + X)),
        TS is T + 1, etat(TS, XS, YS, ['transvaser b dans a' | H]).

inverse(L, R) :- inverse(L, [], R).
inverse([], R, R).
inverse([X | L], T, R) :- inverse(L, [X | T], R).
```

```
etat(0, 5, 0, []).
→ 5 mouvements
[transvaser a dans b, vider b, transvaser a dans b, vider b, transvaser a
dans b]
```

Autre essai, en modifiant les capacités initiales :

```
capacite(a, 7).
capacite(b, 3).
```

```
etat(0, 7, 3, []).
→ 6 mouvements
[vider b, transvaser a dans b, vider b, transvaser a dans b, vider b,
transvaser a dans b]
```

31 Animaux mutants
Grâce à une base d'animaux tels que cheval, lapin, pintade, albatros, alligator, tortue, caribou, ours... on construira par concaténation des mutants comme « lapintade »...
Se servir d'un prédicat que l'on peut définir par *nonvide([_ | _]).*

```
animal([c, h, e, v, a, l]).
animal([a, l, b, a, t, r, o, s]).
animal([l, a, p, i, n]).
animal([p, i, n, t, a, d, e]).
animal([m, o, u, c, h, e]).
animal([p, i, n, s, o, n]).
animal([v, a, c, h, e]).

concat([], L, L).
concat([X | L], M, [X | N]) :-  concat(L, M, N).

nonvide([_ | _]).
```

On décrit le fait que deux listes *A* et *B* ont une intersection *I* non vide, *FB* étant la fin de *B*.

```
mutant(M) :- animal(A), animal(B), A \== B,
             concat(_, I, A), nonvide(I),
             concat(I, FB, B), concat(A, FB, M).
```

mutant(*X*). →
X = [c, h, e, v, a, l, b, a, t, r, o, s] ;
X = [c, h, e, v, a, l, a, p, i, n] ; *X* = [l, a, p, i, n, t, a, d, e] ;
X = [l, a, p, i, n, s, o, n] ; *X* = [m, o, u, c, h, e, v, a, l] ;
X = [v, a, c, h, e, v, a, l]

32 Animaux supermutants
Avec les mêmes animaux, on veut trouver des « trimutants » formés de l'enchaînement de trois animaux dont les intersections sont composées d'au moins deux lettres comme « chevalligatortue ».

On construit le prédicat « avoir au moins deux éléments » :

```
aumoinsdeux([_, _ | _]).
```

```
trimutant(M) :-  animal(A), animal(B), A \== B, animal(C),
                 A \== C, B \== C, concat(_, I1, A), aumoinsdeux(I1),
                 concat(I1, FB, B), concat(MB, I2, FB), aumoinsdeux(I2),
                 concat(I2, FC, C), concat(A, FB, DM), concat(DM, FC, M).
```

trimutant(*X*). → *X* = [m, o, u, c, h, e, v, a, l, b, a, t, r, o, s] ;
X = [v, a, c, h, e, v, a, l, b, a, t, r, o, s]

33 Numération japonaise

Traduire en nombre une expression japonaise sachant que la numération est complètement régulière (13 se dit dix-trois et 30 se dit trois-dix) et que :

itchi = 1,	ni = 2,	san = 3,	shi = 4,	go = 5,
roku = 6,	shitchi = 7,	hatchi = 8,	ku = 9,	giu = 10,
hyaku = 100,	sen = 1000,	man = 10 000.		

Il y a par contre plusieurs séries de mots pour la numération en japonais suivant ce que l'on compte.

```
donnees([itchi, 1, ni, 2, san, 3, shi, 4, go, 5, roku, 6, shitchi, 7, hatchi, 8, ku, 9,
        giu, 10, hyaku, 100, sen, 1000, man, 10000]).

present(X, U, [X, U | _]).
present(X, U, [_ , _ | L]) :- present(X, U, L).

val(X, U) :- donnees(D), present(X, U, D).

jap([X, Y | L], N) :- !, val(X, U), val(Y, V), jap(L, M), N is (M + (U*V)).
jap([X], N) :- !, val(X, N).
jap([], 0).
```

Cette relation ne fonctionne que dans un sens.

jap([ni, giu, hatchi], *N*). → *N* = 28
jap([shitchi, giu, go], *N*). → *N* = 75
jap([go, hyaku, itchi, giu, ni], *N*). → *N* = 512
jap([itchi, sen, shitchi, hyaku, hatchi, giu, ku], *N*). → *N* = 1789
jap([ni, sen, hatchi, hyaku, go, giu, san], *N*). → *N* = 2853
jap([ni, man, ku, sen, itchi, hyaku, roku, giu, shitchi], *N*). → *N* = 29167

34 Numération anglaise
Ecrire un prédicat *nombre* établissant la relation entre une liste de mots anglais traduisant un nombre inférieur à 100 et la valeur de ce nombre. On s'arrangera pour que la relation puisse être interrogée dans les deux sens.

On décrit d'abord les équivalents, puis les nombres à un, deux ou trois chiffres.

```
chiffre(one, 1).
chiffre(two, 2).
chiffre(three, 3).
chiffre(four, 4).
chiffre(five, 5).
chiffre(six, 6).
chiffre(seven, 7).
chiffre(eight, 8).
chiffre(nine, 9).
gdchif(ten,10).
gdchif(eleven,11).
gdchif(twelwe, 12).
gdchif(thirteen, 13).
gdchif(forteen, 14).
gdchif(fifteen, 15).
gdchif(sixteen, 16).
gdchif(seventeen, 17).
gdchif(eighteen, 18).
gdchif(nineteen, 19).
diz(twenty, 20).
diz(thirty, 30).
diz(forty, 40).
diz(fifty, 50).
diz(sixty, 60).
diz(seventy, 70).
diz(eighty, 80).
diz(ninety, 90).

nombre(X, N) :- trois(X, N),!.
nombre(X, N) :- deux(X, N),!.
nombre([U], N) :- chiffre(U, N),!.
nombre([zero], 0).
```

Un prédicat spécial pour les nombres à trois chiffres et un autre pour les nombres à deux chiffres qui sont soit des « grands chiffres » soit des composés.

```
trois([C, hundred |L], N) :- chiffre(C, V), restetrois(L, M), N is (100*V + M).
restetrois([], 0).
restetrois([and | L], N) :- deux(L, N).

deux([D|L], N) :- diz(D, V), restedeux(L, M), N is (V + M).
deux([C], N) :- gdchif(C, N).
deux([U], N) :- chiffre(U, N).
restedeux([U], N) :- chiffre(U, N).
restedeux([], 0).
```

Exemples

nombre([three, hundred], *X*). → *X* = 300
nombre([sixty, five], *X*). → *X* = 65
nombre([one, hundred, four], *X*). → no
nombre([five, hundred, and, four], *X*). → *X* = 504
nombre([two, hundred, and, twenty, four], *X*). → *X* = 224
nombre([seven, hundred, and, ninety], *X*). → *X* = 790

nombre(*L*, 15). → *L* = [fifteen]
nombre(*L*, 243). → *L* = [two, hundred, and, forty, three]
nombre(*L*, 782). → *L* = [seven, hundred, and, eighty, two]
nombre(*L*, 999). → *L* = [nine, hundred, and, ninety, nine]

35 Grands nombres
En représentant un nombre de plusieurs dizaines de chiffres par la liste de ces chiffres à l'envers, reconstruire l'addition, la soustraction arithmétique et la multiplication, puis la relation d'ordre.

On se sert du prédicat d'inversion de listes maintes fois décrit, puis toutes les opérations sont exécutées des chiffres des unités vers les dizaines, centaines, etc., c'est-à-dire en prenant les nombres de droite à gauche. La somme est donc faite dans ce sens par le prédicat *mos* exactement comme elle est enseignée à l'école primaire, de même pour la soustraction, *uos* et la multiplication *lum*.

```
inv(L, LI) :- invbis(L, [], LI).
invbis([X | L], T, R) :- invbis(L, [X | T], R).
invbis([], T, T).
```

```
mos([X | L], [Y | M], R, [Z | N]) :-  Z is (X + Y + R) mod 10,
                                       NR is (X + Y + R) / 10, mos(L, M, NR, N).
mos([], Y, 0, Y) :- !.          % mos, uos et lum sont som, sou et mul à l'envers !
mos([], Y, 1, N) :- mos([1], Y, 0, N).
mos(L, [], 0, L) :- !.
mos(L, [], 1, N) :- mos(L, [1], 0, N).
som(X, Y, Z) :- inv(X, XI), inv(Y, YI), mos(XI, YI, 0, ZI), inv(ZI, Z).

retirzero([0, X | L], R) :- !, retirzero([X | L], R).
retirzero(L, L).
```

> som([7, 5, 2, 3, 6, 9, 8, 4, 1, 0], [1, 2, 5, 0, 6, 3, 9, 9, 8, 7], *R*).
> → *R* = [8, 7, 7, 4, 3, 3, 8, 3, 9, 7]
> som([9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9], [1], *R*).
> → *R* = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
> retirzero([0, 0, 0, 0, 2, 3], *R*). → *R* = [2, 3]

```
uos([X | L], [Y | M], R, [Z | N]) :- ZT is (X − Y − R), ZT < 0, !, Z is ZT + 10,
                                      uos(L, M, 1, N).
uos([X | L], [Y | M], R, [Z | N]) :- Z is (X − Y − R), uos(L, M, 0, N).
uos(L, [], 0, L) :- !.
uos(L, [], 1, N) :- uos(L, [1], 0, N).
sou(X, Y, Z) :- inv(X, XI), inv(Y, YI), uos(XI, YI, 0, ZI), inv(ZI, ZO),
                retirzero(ZO, Z).
```

Le prédicat *lums* réalise la multiplication scalaire (un grand nombre par un seul chiffre) à l'envers, *lum* réalise la multiplication à l'envers et *mul* la vraie multiplication de deux grands nombres, à l'endroit.

```
lums([X | L], Y, R, [Z | N]) :- ZT is (X*Y + R), Z is ZT mod 10,
                                NR is ZT/10, lums(L, Y, NR, N).
lums([], Y, R, [R]).
```

> sou([9, 4, 7, 8, 2, 3, 0, 0, 5, 6], [1, 2, 6, 8, 9, 9, 2, 7, 3], *R*).
> → *R* = [9, 3, 5, 1, 3, 3, 0, 7, 8, 3]
> sou([1, 0, 0, 0, 0, 0, 0, 0, 0, 0], [9, 9, 9, 9, 9, 9, 9, 9, 9, 9], *R*). → *R* = [1]
> lums([8, 9, 5, 3, 2, 6, 2, 4, 5, 0, 7], 8, 0, R).
> → *R* = [4, 8, 7, 8, 8, 9, 0, 4, 3, 4, 6, 5]
> lums([9, 8, 6, 5, 4], 9, 0, *R*). → *R* = [1, 0, 2, 1, 1, 4]

```
lum(L, [Y | M], R) :- lums(L, Y, 0, LY), lum(L, M, N), mos(LY, [0 | N], 0, R).
lum(L, [], [0]).
```

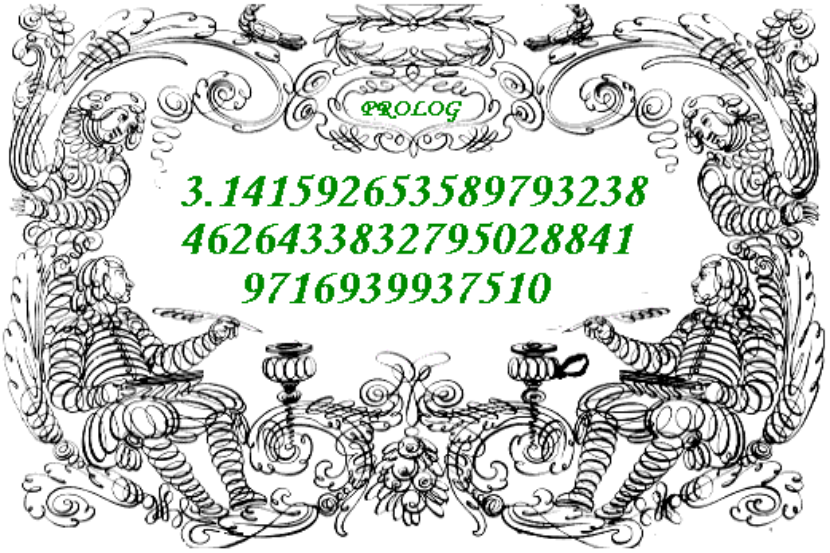
mul(X, Y, Z) :- inv(X, XI), inv(Y, YI), lum(XI, YI, ZI), inv(ZI, ZO),
retirzero(ZO, Z).

> mul([4, 2, 3, 5, 6], [7, 8, 9], *R*). → *R* = [3, 3, 4, 1, 8, 8, 8, 4]
> mul([4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4], [2], *R*).
> → *R* = [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]
> mul([2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [3, 0, 0, 0, 0, 0, 0, 0], *R*).
> → *R* = [6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Telle qu'était définie la soustraction, il suffit de l'utiliser pour le prédicat inférieur ou égal.

inferieur(L, M) :- sou(M, L, _).

> inferieur([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 7]). → yes
> inferieur([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 5]). → no
> inferieur([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6]). → yes



36 Transcription des grands nombres
Toujours pour les grands nombres, définir la relation *trans* passant d'un entier à la liste de ses chiffres.

Cette transcription entre entiers (limités par la capacité) et liste des chiffres associés, donc pour des « grands nombres pas trop grands », va se faire grâce à la relation prédéfinie *name* qui établit le passage entre une constante et la liste de ses codes Ascii, et *map48* transforme une liste de codes Ascii de nombres en ces nombres en leur retirant 48 qui est le code de 0. Le prédicat *decrypt* calcule le « petit nombre » issu de la liste unité, dizaine, centaine...

```
map48([], []).
map48([X | L], [N | R]) :- N is X − 48, map48(L, R).

trans(N, GN) :- nonvar(N), name(N, LA), map48(LA, GN).
trans(N, GN) :- var(N), inv(GN, GNI), decrypt(GNI, 1, 0, N).
decrypt([], _, N, N).
decrypt([C | L], K, SP, N) :- NS is C*K + SP, NK is 10*K,
                              decrypt(L, NK, NS, N).
```

```
name(12, X). → X = [49, 50]
name(C, [48, 50]). → C = 2
decrypt([1, 2, 3, 4, 5, 6, 7], 1, 0, N). → N = 7654321
trans(120360789, L). → L = [1, 2, 0, 3, 6, 0, 7, 8, 9]
trans(N, [7, 6, 5, 0, 3, 2, 1]). → N = 7650321
```

37 Puissances et racines *N*-ième pour les grands nombres
(Reprise du chapitre 2).

```
puiss(1, X, X) :- !.          % N est un entier, X est un grand nombre
puiss(N, X, P) :- K is N – 1, puiss(K, X, PS), !, mul(X, PS, P).
rac(N, X, Q) :- racbis(N, X, [1], Q).
racbis(N, X, Q, Q) :- puiss(N, Q, X), !.
racbis(N, X, M, Q) :- puiss(N, M, P), inferieur(X, P), !, sou(M, [1], Q).
racbis(N, X, M, Q) :- som(M, [1], MS), racbis(N, X, MS, Q).
```

```
puiss(3, [5], P). → P = [1, 2, 5] % c'est 5^3 = 125, 3^9 = 243, 7^4 = 2401
puiss(5, [3], P). → P = [2, 4, 3]
rac(4, [2, 4, 0, 1], Q). → Q = [7]
rac(5, [2, 4, 2], Q). → Q = [2]
rac(5, [2, 4, 3], Q). → Q = [3]
```

Chapitre 4

Problèmes avec contraintes

Nous présentons ici quelques problèmes classiques où il s'agit de trouver des solutions en évitant l'explosion combinatoire par des contraintes plus ou moins fortes. Il faut tester les contraintes les plus difficiles en premier de façon à diminuer au maximum l'espace de recherche. Mais un autre problème est celui de la représentation des données. Nous fournissons plusieurs variantes, notamment dans les pseudo-tableaux à construire. Le chapitre suivant reprend ce genre de problèmes avec des termes plus structurés.

1 Algorithme à retour assez simple
Construire un nombre avec une fois chacun des chiffres de 1 à 9 tel que les *k* premiers chiffres de ce nombre soit divisibles par *k*, pour *k* de 1 à 9.
Détail technique "ab + 12" est identique à [97, 98, 43, 49, 50] et *name(S, LA)* établit la relation entre la chaîne de caractères *S* et la liste de ses codes Ascii.

La première clause représente le cas où c'est fini, *D* = diviseur initialement 1, *R* = résultat, *CC* = liste des ascii des chiffres choisis, initialement [], *LC* = liste des Ascii des chiffres disponibles.

```
essaidiv([], D, CC, R) :- name(R, CC).
essaidiv(LC, D, CC, R) :-  % CC liste des Ascii des chiffres choisis
        ret(C, LC, RC),    % choisir un chiffre C dans LC, il reste RC
        append(CC, [C], NCC),    % NCC = nouvelle liste de chiffres choisis
        name(N, NCC),            % qui correspond à un nombre N
        0 is N mod D,            % il faut que N soit divisible par D
        ND is D + 1,             % ND = nouveau diviseur à essayer
        essaidiv(RC, ND, NCC, R).% faire la suite RC = reste des chiffres

append([], X, X).
append([X | Y], Z, [X | R]) :- append(Y, Z, R).

ret(X, [X | R], R).
ret(X, [Y | R], [Y | Z]) :- ret(X, R, Z).
```

> essaidiv("123456789", 1, [], *R*). → *R* = 381654729
> (3 est divisible par 1, 38 par 2, 381 par 3, etc.)

2 Des chiffres et des lettres
On va considérer uniquement des chiffres au sein d'une liste et une somme *S*. Le but est d'obtenir *S* en puisant des valeurs dans cette liste et en utilisant uniquement les addition, soustraction et multiplication.

```
compte(_, S, S, R, R) :- write(S), write(' en opérant de droite à gauche ').
compte([X | L], SP, S, E, R) :- NS is SP + X, compte(L, NS, S, [X, + | E], R).
compte([X | L], SP, S, E, R) :- NS is SP − X, compte(L, NS, S, [X, - | E], R).
compte([X | L], SP, S, E, R) :- NS is SP * X, compte(L, NS, S, [X, * | E], R).
compte([_ | L], SP, S, R) :- compte(L, SP, S, R).
                        % on peut se passer d'une donnée
compte(L, S, R) :- compte(L, 0, S, [0], R).    % départ
```

compte([1, 2, 3, 4, 5], 19, *R*). →
19 en opérant de droite à gauche *R* = [5, −, 4, *, 3, +, 2, +, 1, +, 0] ;
19 en opérant de droite à gauche *R* = [5, −, 4, *, 3, *, 2, *, 1, +, 0] ;
19 en opérant de droite à gauche *R* = [5, −, 4, *, 3, *, 2, +, 1, *, 0] ; no

compte([5, 1, 2, 3, 4, 0], 25, *R*). →
25 en opérant de droite à gauche *R* = [4, +, 3, *, 2, +, 1, *, 5, +, 0] ;
25 en opérant de droite à gauche *R* = [0, +, 4, +, 3, *, 2, +, 1, *, 5, +, 0] ;
25 en opérant de droite à gauche *R* = [0, −, 4, +, 3, *, 2, +, 1, *, 5, +, 0] ;
no

compte([4, 2, 3, 1, 8], 17, *R*). →
17 en opérant de droite à gauche *R* = [8, +, 1, *, 3, +, 2, +, 4, +, 0] ;
17 en opérant de droite à gauche *R* = [1, − 3, *, 2, +, 4, +, 0] ;
17 en opérant de droite à gauche *R* = [8, −, 1, +, 3, *, 2, *, 4, +, 0] ; no

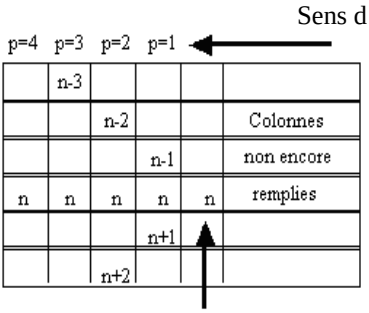
compte([7, 2, 13, 48, 23], 52, *R*). →
52 en opérant de droite à gauche *R* = [23, −, 48, +, 13, +, 2, *, 7, +, 0] ;
52 en opérant de droite à gauche *R* = [48, +, 13, +, 2, −, 7, −, 0] ; no

compte([2, 2, 2, 2, 2], 8, *R*).
8 en opérant de droite à gauche *R* = [2, +, 2, +, 2, +, 2, +, 0] ;
8 en opérant de droite à gauche *R* = [2, *, 2, −, 2, +, 2, +, 2, +, 0] ;
8 en opérant de droite à gauche *R* = [2, *, 2, +, 2, −, 2, +, 2, +, 0] ;
8 en opérant de droite à gauche *R* = [2, *, 2, *, 2, −, 2, +, 2, +, 0] ;
et 20 autres solutions

3 Reines de Gauss
Ce problème consiste à placer 8 reines sur un échiquier de 8 cases sur 8 sans qu'aucune ne puisse être en position de prise. Le généraliser à la dimension quelconque *N*. La difficulté réside dans la recherche des fils d'un état du problème et non dans l'algorithme principal, puisque Prolog fait le backtrack.

La représentation d'un état du problème, c'est-à-dire des positions respectives des reines déjà placées, peut se faire avec l'ensemble des coordonnées de ces reines, mais il est beaucoup plus simple de tenir à jour la liste des numéros de lignes occupées, la plus ancienne correspondant à la première colonne et la plus récente en début de liste correspondant au dernier choix de ligne.
Ainsi, pour 3 reines placées aux 3 premières colonnes sur les lignes 1, 3, 5, on représentera cet état provisoire par [5; 3; 1] la position suivante *i* sera donc empilée en premier. La validité indique que la rangée *i* est compatible avec l'état antérieur *e*, à savoir que *j* colonnes avant celle où on se propose de mettre une reine à la ligne *i*, si la reine se trouve à la ligne *k*, on doit avoir $i \neq k$ et sans prise possible en diagonale c'est-à-dire $k \neq i + j$ et $k \neq i - j$:

```
len([], 0).
len([_ | L], N) :- len(L, M), N is M + 1.
conc([], L, L).
conc([X | L], M, [X | R]) :- conc(L, M, R).
fils(0, _, []) :- !.
fils(N, LC, F) :- filsbis(N, 1, LC, FB), M is N - 1,
                  fils(M, LC, FM), conc(FB, FM, F).
filsbis(LG, _, [], [LG]) :- !.
% la ligne LG est compatible avec les lignes déjà choisies
filsbis(LG, P, [X | LC], []) :- (LG is X; LG is X + P; LG is X − P), !.
filsbis(LG, P, [X | LC], F) :- Q is P + 1, filsbis(LG, Q, LC, F).
```



fils(7, [3, 5, 7], *F*). → *F* = [6, 1]
fils(8, [1, 5, 3], *F*). → *F* = [8, 4]
fils(8, [3], *F*). → *F* = [8, 7, 6, 5, 1]
fils(7, [], *F*). → *F* = [7, 6, 5, 4, 3, 2, 1]

reines(N, S) :- cherche(N, [], S).
cherche(N, S, S) :- len(S, N), !.
cherche(N, LC, S) :- fils(N, LC, F), app(X, F), cherche(N, [X | LC], S).

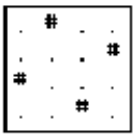
reines(4, *S*).
→ *S* = [2, 4, 1, 3] ; *S* = [3, 1, 4, 2] ; no

reines(5, *S*).
→ S = [2, 4, 1, 3, 5] ; *S* = [3, 1, 4, 2, 5] ; *S* = [1, 3, 5, 2, 4] ;
S = [2, 5, 3, 1, 4] ; *S* = [1, 4, 2, 5, 3] ; *S* = [5, 2, 4, 1, 3] ;
S = [4, 1, 3, 5, 2] ; *S* = [5, 3, 1, 4, 2] ; *S* = [3, 5, 2, 4, 1] ;
S = [4, 2, 5, 3, 1] ; no

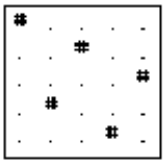
reines(6, *S*).
→ *S* = [2, 4, 6, 1, 3, 5] ; *S* = [3, 6, 2, 5, 1, 4] ; *S* = [4, 1, 5, 2, 6, 3] ;
S = [5, 3, 1, 6, 4, 2] ; no

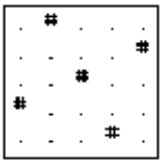
reines(7, *S*).
→ *S* = [2, 4, 6, 1, 3, 5, 7] et 39 autres solutions

reines(8, *S*).
→ *S* = [5, 7, 2, 6, 3, 1, 4, 8] et 91 solutions en tout.

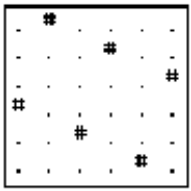


n = 4



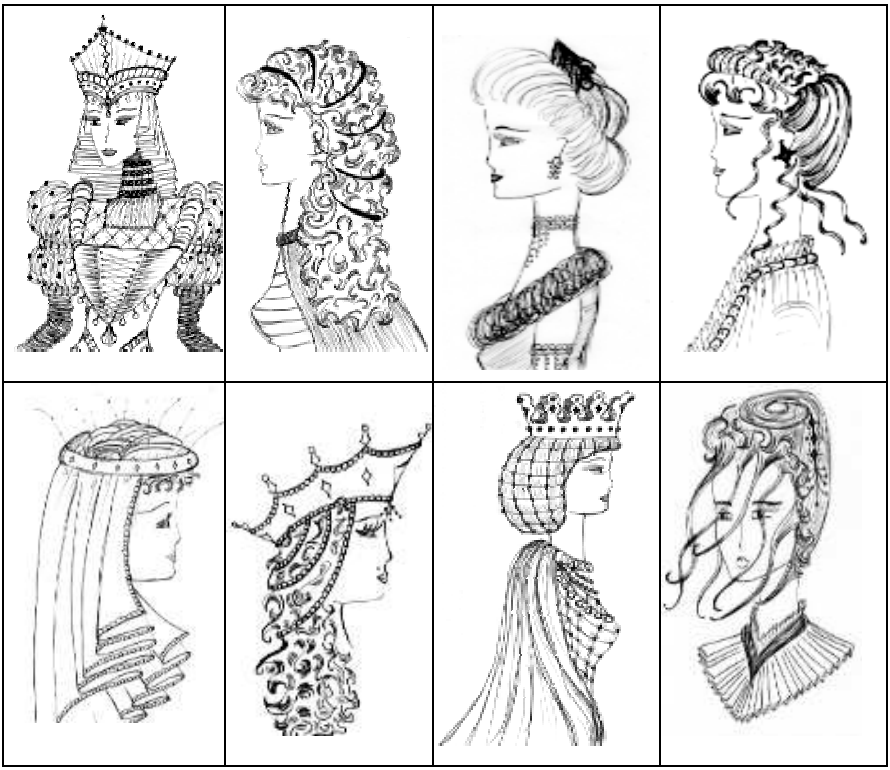


n = 5



n = 6

De gauche à droite, les reines sont aux lignes [1, 4, 2, 1], puis [1, 4, 2, 5, 3] et une autre solution [4, 1, 3, 5, 2] pour *n* = 5 et [4, 1, 5, 2, 6, 3] pour le dernier.



Les huit reines Prologa I à Prologa VIII.

4 Problème du cavalier
Un cavalier au jeu d'échec doit passer une fois et une seule fois sur chaque case. On sait qu'il ne peut se déplacer qu'en diagonale d'un rectangle de deux sur trois cases. Le départ est fixé en haut à gauche, convention qu'il est facile de modifier.

On adopte la représentation d'une case par [*X, Y, Occup*] décrite par ses coordonnées et *Occup* = le numéro du saut dans la marche du cavalier, si elle est déjà occupée. *S* est la liste de ces positions, l'ordre n'y a donc pas d'importance sauf pour la vue finale du résultat.
Si le départ est en (1, 1), alors *S* = [[1, 1, 1]] initialement, et l'état final, s'il y a au moins une solution, est obtenu dès que, pour un échiquier de taille *N* sur *N*, l'on a *app([_, _, N^2], S).* La détermination des voisins, c'est-à-dire des

coups possibles suivants est un peu longue à décrire, mais la construction du parcours est d'une très grande lisibilité.

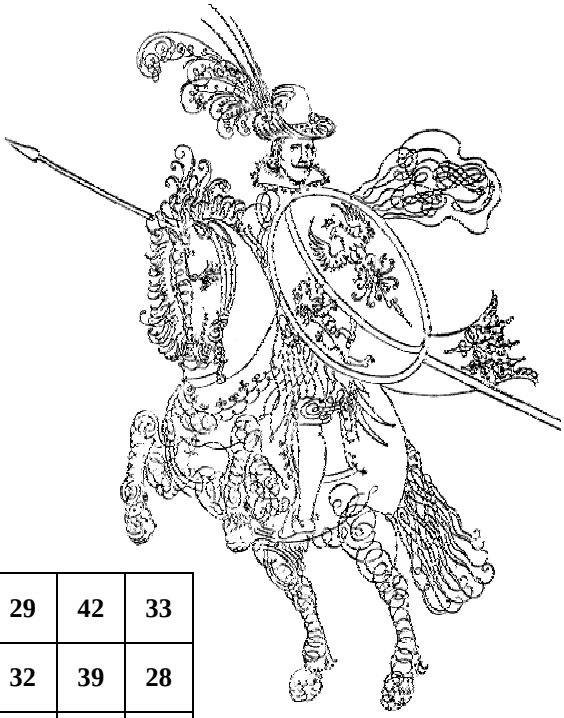
```
tronque([], _, _, []).
tronque([[X, Y, F] | L], N, S, R)
        :- (X < 1; Y < 1; N < X; N < Y; app([X, Y, _], S)),
        !, tronque(L, N, S, R).
tronque([C | L], N, S,[C | R]) :- tronque(L, N, S, R).

voisins(I, J, K, N, S, V)
                :- IA is I – 2, IB is I – 1, IC is I + 1, ID is I + 2, JA is J – 2,
                JB is J – 1, JC is J + 1, JD is J + 2, KS is K + 1,
                        tronque([[IA, JB, KS], [IB, JA, KS], [IC, JA, KS],
                            [ID, JB, KS], [ID, JC, KS], [IC, JD, KS],
                            [IB, JD, KS], [IA, JC, KS]], N, S, V).
% Les 8 voisins sauf ceux hors de l'échiquier

cavalier(N, R) :- construire(N, [[1, 1, 1]], R).

construire(N, [[X, Y, F] | S], [[X, Y, F] | S]) :- F is N*N, !.
 % cas où on a fini
construire(N, [[I, J, K] | S], R) :- voisins(I, J, K, N, S, V), app(P, V),
                                    construire(N, [P, [I, J, K] | S], R).
```

tronque([[1, 5, 2], [3, 5, 2]], 5, [], *R*). → *R* = [[1, 5, 2], [3, 5, 2]]
tronque([[1, 5, 2], [3, 5, 2]], 4, [], *R*). → *R* = []
tronque([[1, 5, 2], [3, 5, 2]], 5, [[3, 5, 1]], *R*). → *R* = [[1, 5, 2]]

voisins(1, 3, 1, 5, [], *V*). → *V* = [[2, 1, 2], [3, 2, 2], [3, 4, 2], [2, 5, 2]]
voisins(3, 3, 1, 6, [], *V*).
→ *V* = [[1, 2, 2], [2, 1, 2], [4, 1, 2], [5, 2, 2], [5, 4, 2], [4, 5, 2], [2, 5, 2], [1, 4, 2]]
voisins(2, 3, 7, 5, [[4, 4, 2]], *V*).
→ *V* = [[1, 1, 8], [3, 1, 8], [4, 2, 8], [3, 5, 8], [1, 5, 8]]

cavalier(4, *R*). → no
cavalier(5, *R*). → *R* = [[5, 5, 25], [3, 4, 24], [1, 5, 23], [2, 3, 22], [3, 5, 21], [1, 4, 20], [3, 3, 19], [5, 4, 18], [4, 2, 17], [2, 1, 16], [1, 3, 15], [2, 5, 14], [4, 4, 13], [5, 2, 12], [3, 1, 11], [1, 2, 10], [2, 4, 9], [4, 5, 8], [5, 3, 7], [4, 1, 6], [2, 2, 5], [4, 3, 4], [5, 1, 3], [3, 2, 2], [1, 1, 1]]
Et une foule d'autres solutions

1	**10**	**15**	**20**	**23**
16	**5**	**22**	**9**	**14**
11	**2**	**19**	**24**	**21**
6	**17**	**4**	**13**	**8**
3	**12**	**7**	**18**	**25**



1	**20**	**31**	**40**	**29**	**42**	**33**
18	**7**	**16**	**21**	**32**	**39**	**28**
15	**2**	**19**	**30**	**41**	**34**	**43**
8	**17**	**6**	**47**	**22**	**27**	**38**
3	**14**	**11**	**24**	**37**	**44**	**35**
12	**9**	**48**	**5**	**46**	**23**	**26**
49	**4**	**13**	**10**	**25**	**36**	**45**

cavalier(6, *R*). → *R* = [[6, 1, 36], [5, 3, 35], [6, 5, 34], [4, 6, 33], [2, 5, 32], [1, 3, 31], [3, 4, 30], [2, 6, 29], [1, 4, 28], [2, 2, 27], [4, 1, 26], [6, 2, 25], [5, 4, 24], [6, 6, 23], [4, 5, 22], [2, 4, 21], [1, 6, 20], [3, 5, 19], [5, 6, 18], [4, 4, 17], [2, 3, 16], [1, 5, 15], [3, 6, 14], [5, 5, 13], [4, 3, 12], [6, 4, 11], [5, 2, 10], [3, 1, 9], [1, 2, 8], [3, 3, 7], [2, 1, 6], [4, 2, 5], [6, 3, 4], [5, 1, 3], [3, 2, 2], [1, 1, 1]]

cavalier(7, *R*). → *R* = [[7, 1, 49], [6, 3, 48], [4, 4, 47], [6, 5, 46], [7, 7, 45], [5, 6, 44], [3, 7, 43], [1, 6, 42], [3, 5, 41], [1, 4, 40], [2, 6, 39], [4, 7, 38], [5, 5, 37], [7, 6, 36], [5, 7, 35], [3, 6, 34], [1, 7, 33], [2, 5, 32], [1, 3, 31], [3, 4, 30], [1, 5, 29], [2, 7, 28], [4, 6, 27], [6, 7, 26], [7, 5, 25], [5, 4, 24], [6, 6, 23], [4, 5, 22], [2, 4, 21], [1, 2, 20], [3, 3, 19], [2, 1, 18], [4, 2, 17], [2, 3, 16], [3, 1, 15], [5, 2, 14], [7, 3, 13], [6, 1, 12], [5, 3, 11], [7, 4, 10], [6, 2, 9], [4, 1, 8], [2, 2, 7], [4, 3, 6], [6, 4, 5], [7, 2, 4], [5, 1, 3], [3, 2, 2], [1, 1, 1]] (dessin précédent)

5 Parcours semi-magique du cavalier
Partant encore de la case gauche supérieure, le cavalier doit passer par *N*N* positions différentes d'un échiquier de côté *N*. Seulement, cette fois on cherche à ce que la somme des rangs des sauts de ce cavalier soit la même sur chaque ligne et chaque colonne.

La représentation est la même, on prend le tableau comme la liste *S* des triplets [ligne *I*, colonne *J*, numéro du saut *K*], cette liste est achevée si et seulement si on arrive à $K = N^2$.
Les prédicats *lgn* et *col* permettent de calculer, en cours de construction, la somme partielle *SL* des rangs déjà placés (et leur nombre *NL*) sur une ligne *I* ou une colonne *J* (*SC* et *NC*). *Impossible* permet alors de savoir s'il est impossible ou non de mettre le saut *K* en position (*X, Y*). Pour cela, comme on construit les sauts dans l'ordre de 1 à N^2, on sait que les D sauts suivants seront de rangs minimums *K*, *K* + 1, ... *K* + *D* – 1. Il faut donc, si *SP* est la somme partielle déjà contenue dans une ligne ou une colonne, que l'on ait une inégalité telle que :

$$SP + K + K + 1 + ... + K - D - 1 = SP + D*K + D*(D - 1) / 2 < SM$$

D'où le programme :

```
somme(N, SM) :- SM is N*(N*N + 1) / 2.
lgn(I, [[I, J, K] | S], SP, SL, NI, NL) :- !, SS is SP + K, NS is NI + 1,
                                            lgn(I, S, SS, SL, NS, NL).
```

```
lgn(I, [[_, _, _] | S], SP, SL, NI, NL) :- lgn(I, S, SP, SL, NI, NL).
lgn(I, [], SL, SL, NL, NL).
        % SL et NL sont les somme et nombre trouvés sur la ligne I
col(J, [[I, J, K] | S], SP, SC, NI, NC) :- !, SS is SP + K, NS is NI + 1,
                                        col(J, S, SS, SC, NS, NC).
col(J, [[_, _, _] | S], SP, SC, NI, NC) :- col(J, S, SP, SC, NI, NC).
col(J, [], SC, SC, NC, NC).
        % SC, NC somme et nombre trouvés sur la colonne J

nonposs(NI, N, SP, K, SM) :- N is NI + 1, !, (SM < SP + K; SM > SP + K).
        % cas du dernier placé sur une ligne ou une colonne
nonposs(NI, N, SP, K, SM) :- D is N − NI, % il reste D cases à remplir
        % on pourrait donner SM < SP + D*K.
        SM =< SP + D*K + D*(D − 1) / 2. % condition plus restrictive

app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).

voisins(I, J, K, N, S, SM, V) :- IA is I – 2, IB is I – 1, IC is I + 1, ID is I + 2,
                JA is J – 2, JB is J – 1, JC is J + 1, JD is J + 2, KS is K + 1,
                  tronque([[IA, JB, KS], [IB, JA, KS], [IC, JA, KS],
                           [ID, JB, KS], [ID, JC, KS], [IC, JD, KS],
                           [IB, JD, KS], [IA, JC, KS]], N, S, SM, V).
tronque([], _, _, _, []).
tronque([[X, Y, K] | L], N, S, SM, R) :-  (X < 1; Y < 1; N < X; N < Y;
        app([X, Y, _], S);impossible(X, Y, K, N, S, SM)), !,
        tronque(L, N, S, SM, R).
tronque([C | L], N, S, SM, [C | R]) :- tronque(L, N, S, SM, R).
impossible(X, Y, K, N, S, SM) :-      % impossible de rajouter K sur ligne X
        lgn(X, S, 0, SL, 0, NL),   % calcule SL, NL déjà placés sur la ligne X
        nonposs(NL, N, SL, K, SM).
        % impossible de placer encore K sur la ligne X

impossible(X, Y, K, N, S, SM) :-  col(Y, S, 0, SC, 0, NC),
        nonposs(NC, N, SC, K, SM).
        % impossible de placer K sur cette colonne
        % il serait possible d'ajouter des contraintes sur les diagonales
cavaliermagik(N, R) :- somme(N, SM), construire(N, [[1, 1, 1]], SM, R).

construire(N, [[X, Y, F] | S], _, [[X, Y, F] | S]) :- F is N*N, !.
        % cas où on a fini
```

```
construire(N, [[I, J, K] | S], SM, R) :-
        voisins(I, J, K, N, S, SM, V),
        % si (I, J) est la position du K-ième coup
        app([X, Y, KS], V),
        % on examine les coups suivants de l'ensemble des voisins V
        construire(N, [[X, Y, KS], [I, J, K] | S], SM, R).
```

somme(7, *SM*). → *SM* = 175
somme(8, *SM*). → *SM* = 260

lgn(2, [[2, 3, 10], [3, 1, 9], [4, 3, 8], [2, 2, 7], [3, 4, 6], [4, 2, 5], [2, 1, 4], [1, 3, 3], [3, 2, 2], [1, 1, 1]], 0, *S*, 0, *N*). → *S* = 21 *N* = 3

La première solution pour *N* = 8 a pour somme 260 et chaque demi-ligne ou demi-colonne a d'ailleurs pour somme 130.

cavaliermagik(8, *R*). → *R* = [[3, 8, 64], [1, 7, 63], [2, 5, 62], [4, 6, 61], [5, 8, 60], [7, 7, 59], [8, 5, 58], [6, 6, 57], [5, 4, 56], [7, 3, 55], [8, 1, 54], [6, 2, 53], [4, 1, 52], [2, 2, 51], [1, 4, 50], [3, 3, 49], [1, 2, 48], [3, 1, 47], [2, 3, 46], [4, 4, 45], [5, 2, 44], [7, 1, 43], [8, 3, 42], [6, 4, 41], [5, 6, 40], [7, 5, 39], [8, 7, 38], [6, 8, 37], [4, 7, 36], [2, 8, 35], [3, 6, 34], [1, 5, 33], [3, 4, 32], [1, 3, 31], [2, 1, 30], [4, 2, 29], [6, 1, 28], [8, 2, 27], [7, 4, 26], [5, 3, 25], [6, 5, 24], [8, 6, 23], [7, 8, 22], [5, 7, 21], [4, 5, 20], [2, 6, 19], [1, 8, 18], [3, 7, 17], [1, 6, 16], [3, 5, 15], [2, 7, 14], [4, 8, 13], [6, 7, 12], [8, 8, 11], [7, 6, 10], [5, 5, 9], [6, 3, 8], [8, 4, 7], [7, 2, 6], [5, 1, 5], [4, 3, 4], [2, 4, 3], [3, 2, 2], [1, 1, 1]] correspondant à :

1	**48**	**31**	**50**	**33**	**16**	**63**	**18**
30	**51**	**46**	**3**	**62**	**19**	**14**	**35**
47	**2**	**49**	**32**	**15**	**34**	**17**	**64**
52	**29**	**4**	**45**	**20**	**61**	**36**	**13**
5	**44**	**25**	**56**	**9**	**40**	**21**	**60**
28	**53**	**8**	**41**	**24**	**57**	**12**	**37**
43	**6**	**55**	**26**	**39**	**10**	**59**	**22**
54	**27**	**42**	**7**	**58**	**23**	**38**	**11**

6 Une cryptaddition
SUEDE + SUISSE = BRESIL
Chaque lettre représente un chiffre, deux lettres distinctes sont associées à deux chiffres distincts, et *S* et *B* ne peuvent être zéro. Rappel : *div* (ou / ou // est la division entière), si la relation *différent* symbolisé par \==. n'existe pas, on peut le redéfinir.

On présente une solution adaptée à cette équation, le problème général étant abordé à l'exercice suivant. Dans cette solution particulière, on décrit exactement les différentes sommes avec retenues. Le fait que l'appartenance à une liste de chiffres soit vérifiée avec tous les essais possibles permet de décrire ce qu'est la somme d'une colonne avec *X* et *Y* et une retenue *R*, deux retenues étant possibles.

```
app(X, [X|_]).
app(X, [_|L]) :- app(X, L).

chif(N) :- app(N, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
retenue(0).
retenue(1).
distincts([]).
distincts([X | L]) :-  not(app(X, L)), distincts(L).

som(R, X, Y, Z, NR) :- retenue(R), chif(X), chif(Y), T is (X + Y + R),
                       Z is (T mod 10), NR is (T / 10).

suede :- som(0, E, E, L, R1), som(R1, D, S, I, R2), som(R2, E, S, S, R3),
         som(R3, U, I, E, R4), som(R4, S, U, R, R5), som(R5, 0, S, B, 0),
         distincts([S, E, D, L, I, B, U, R]), write( 'e = '), write(E),
         write(' l = '), write(L), write(' d = '), write(D), write(' s = '),
         write(S), write(' u = '), write(U), write(' i = '), write(I),
         write(' b = '), write(B), write(' r = '), write(R).
```

distincts([a, z, e, r, t, y]). → yes
distincts([f, a, d, a]). → no
suede. → e = 9 l = 8 d = 7 s = 4 u = 6 i = 2 b = 5 r = 0 yes

```
  SUEDE      46979
+ SUISSE   + 462449
  BRESIL     509428
```

7 Cryptadditions
Il s'agit de résoudre plus généralement des sommes où chaque lettre (10 au maximum) représente de façon injective un chiffre (non nul s'il est en début de mot). Le programme pourra prendre les mots comme liste de leurs lettres, ainsi « femme » sera [f, e, m, m, e] et devra bien sûr afficher les associations *f* = 6, *e* = 1, *m* = 7... trouvées s'il y en a. Puis réaliser la transformation *pere + mere → [p, e, r, e], [m, e, r, e]* pour avoir par exemple :
jeu(plein + soleil = bruler). → n = 6, l = 1, r = 7, i = 5, e = 0, u = 2, p = 8, o = 9, s = 3, b = 4
Rappelons que "ab" est identique à [97, 98] et que *name(azerty, X).* renvoie *X* = [97, 122, 101, 114, 116, 121]. Cependant *name* ne veut pas d'un mélange de lettres et d'autres signes. (Problèmes issus du journal de Spirou.)

```
chif(0). chif(1). chif(2). chif(3). chif(4). chif(5). chif(6). chif(7).
chif(8). chif(9).
```

Ou bien *chif(X) :- app(X, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).* mais par contre une clause telle que : *chif(X) :- (−1 < X), (X < 10).* ne peut convenir car il faut forcer *X* à être instancié.

```
inv(L, LR) :- invbis(L, [], LR).        % Inverse d'une liste
invbis([], R, R).
invbis([X | L], R, LR) :- invbis(L, [X | R], LR).
toutinv([],[]).
toutinv([X|L], [IX| IL]) :- inv(X, IX), toutinv(L, IL).
```

Le prédicat *toutinv* réalise un « mapcar » de *inverse* sur une liste c'est-à-dire une application d'inverse sur chaque élément d'une liste.

```
ass(X, V, [], [[X, V]]).                % La lettre X est associée à la valeur V
ass(X, V, [[X, V] | R], [[X, V] | R]).
ass(X, V, [[X, W] | _], _) :- V \== W, fail.
ass(X, V, [[Y, V] | _], _) :- X \== Y, fail.
ass(X, V, [[Y, W] | R], [[Y, W] | NR]) :- X \== Y, V \== W, ass(X, V, R, NR).

reso(R, [], [], [Z], CH) :- chif(R), R \== 0, ass(Z, R, CH, NCH), write(NCH).
        % CH, NCH  listes des choix, RL lignes restantes à sommer
reso(S, [[X] | RL], LR, T, CH) :- chif(V), V \== 0, ass(X, V, CH, NCH),
                                  NS is (S + V), reso(NS, RL, LR, T, NCH).
                                  % cas du premier chiffre X d'une ligne
```

LR est la liste de référence constituant ce qui restera à additionner, *T* le total, *X* le chiffre courant de la colonne dont on connaît déjà la somme *S*, *NS* la nouvelle somme *S + valeur(X)* et *Z* est le chiffre du total correspondant à la colonne.

```
reso(S, [[X | L]|RL], LR, T, CH) :- chif(V), ass(X, V, CH, NCH),
        NS is (S + V), reso(NS, RL, [L | LR], T, NCH).
        % L est une ligne, RL lignes restantes à sommer
reso(S, [[] | RL], LR, T, CH) :- reso(S, RL, LR, T, CH).
        % cas d'une ligne épuisée
reso(S, [], LR, [Z, Y | T], CH) :- U is (S mod 10), ass(Z, U, CH, NCH),
        NR is (S / 10), reso(NR, LR, [],[Y | T], NCH).
        % cas de fin de colonne
solution(LT, R) :- inv(R, RR), toutinv(LT, RLT), reso(0, RLT, [], RR, []).
```

Mais on a mieux, sans *inv* ni *toutinv*, *tr(C, _, R, _, S)* réalisant le décodage d'une chaîne *C* en la liste des listes des opérandes *R* et de la somme *S*.

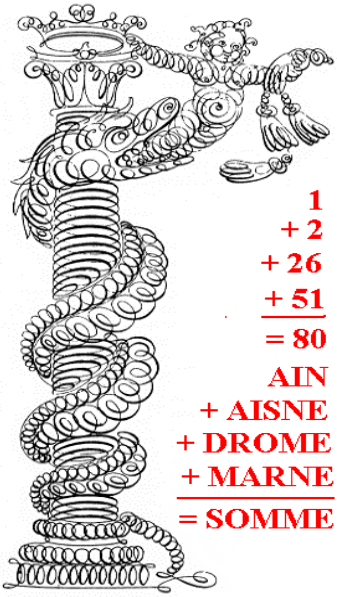
```
tr([61,Y | L], R, RO, [], S) :- tr(L, R, RO, [Y], S), !.
                        % reconnaissance du signe = (61)
tr([43 | L], R, RO, [], S) :- tr(L, [[] | R], RO, [], S), !.
                        % signe + Ascii 43
tr([X | L], [T | R], RO, [], S) :- tr(L, ,[[X | T] | R], RO, [], S), !.
tr([X | L], R, RO, SP, S) :- tr(L, R, RO, [X | SP], S), !.
                        % SP = somme partielle, S = somme
tr([], R, R, S, S).
jeu(CH) :- tr(CH, [[]], R, [], S), lettres(R, LR), lettres(S, LS),
        reso(0, LR, [], LS, []).
lettres([], []). % correspondance entre liste de code et liste de lettres
lettres([N | L], [A | LL]) :- number(N), name(A, [N]), lettres(L, LL).
lettres([LN | R], [LA | LR]) :- lettres(LN, LA), lettres(R, LR).
```

```
jeu("homme+femme=amour").
→ [[e, 1], [r, 2], [m, 7], [u, 4], [o, 5], [h, 3], [f, 6], [a, 9]]
jeu("fort+erie=ville").
→ [[t, 0], [e, 6], [i, 4], [r, 8], [l, 2], [o, 3], [f, 7], [v, 1]]
jeu("megot+megot=clopes").
→ [[t, 2], [s, 4], [o, 3], [e, 6], [g, 5], [p, 0], [m, 8], [l, 7], [c, 1]]
jeu("coup+d+epee=botte").
→ [[p, 2], [d, 8], [e, 4], [u, 0], [t, 5], [o, 3], [c, 9], [b, 1]]
jeu("lit+nuit=reve").
→ [[t, 1], [e, 2], [i, 3], [v, 6], [u, 5], [l, 7], [n, 8], [r, 9]]
```

```
jeu("un+one+uno=euro"). → [[o, 8], [e, 1], [n, 9], [u, 3], [r, 2]]
jeu("jupiter+saturne+uranus=neptune").
→ [[r, 8], [e, 6], [s, 2], [u, 3], [n, 4], [t, 0], [a, 9], [i, 7], [p, 5], [j, 1]]
jeu("roue+roue+roue+roue=auto").
→ [[e, 6], [o, 4], [u, 9], [t, 8], [r, 1], [a, 5]]
jeu("rennes+nantes=angers").
→ [[s, 0], [e, 3], [r, 6], [n, 2], [t, 1], [g, 4], [a, 9]]
jeu("jean+aime=marie").
→ [[n, 0], [e, 4], [m, 1], [a, 7], [i, 8], [r, 2], [j, 9]]
jeu("six+cinq=onze").
→ [[x, 1], [q, 2], [e, 3], [n, 4], [i, 6], [z, 0], [s, 7], [c, 8], [o, 9]]
jeu("un+un+neuf=onze"). → [[n, 1], [f, 7], [e, 9], [u, 8], [z, 4], [o, 2]]
jeu("filo+filino+patro+patrino=familio").
→ [[o, 0], [n, 1], [r, 2], [l, 5], [i, 9], [t, 8], [a, 3], [f, 6], [p, 4], [m, 7]]
jeu("rosee+froid=givre").
→ [[e, 1], [d, 0], [i, 2], [r, 3], [s, 6], [o, 8], [v, 4], [f, 5], [g, 9]]
jeu("manger+manger=grossir").
→ [[r, 0], [e, 4], [i, 8], [g, 1], [s, 2], [n, 6], [a, 3], [o, 7], [m, 5]]
jeu("un+peu+fou=fada").
→ [[n, 2], [u, 4], [a, 0], [o, 3], [e, 7], [d, 5], [p, 8], [f, 1]]
jeu("four+three=seven").
→ [[r, 2], [e, 3], [n, 5], [u, 0], [o, 9], [v, 1], [h, 4], [f, 8], [t, 6], [s, 7]]
jeu("ten+ten+twenty+twenty=twenty=eighty").
→ [[n, 4], [y, 6], [t, 1], [e, 3], [h, 5], [g, 0], [w, 2], [i, 7]]
jeu("calcul+mental=eleves").
→ [[l, 7], [s, 4], [a, 8], [u, 0], [e, 9], [c, 5], [t, 1], [v, 6], [n, 2], [m, 3]]
jeu("calcul+integral=ingenier").
→ [[l, 4], [r, 8], [a, 9], [u, 7], [e, 6], [c, 3], [i, 2], [g, 5], [n, 0], [t, 1]]
jeu("gilles+cohen=genial").
→ [[s, 1], [n, 3], [l, 4], [e, 8], [a, 6], [h, 0], [i, 5], [o, 9], [c, 2], [g, 7]]
jeu("une+demi+finale=facile").
→ [[e, 2], [i, 8], [l, 0], [m, 4], [n, 5], [u, 6], [a, 9], [d, 7], [c, 3], [f, 1]]
jeu("enigme+enigme=colles").
→ [[e, 4], [s, 8], [m, 7], [g, 5], [l, 1], [i, 0], [n, 6], [o, 2], [c, 9]]
jeu("seven+seven+six=twenty").
→ [[n, 2], [x, 0], [y, 4], [i, 5], [e, 8], [t, 1], [v, 7], [s, 6], [w, 3]]
jeu("pleine+lune=crimes").
→ [[e, 2], [s, 4], [n, 6], [u, 3], [i, 1], [m, 5], [l, 9], [r, 0], [p, 7], [c, 8]]
jeu("soleil+sable=bikini").
→ [[l, 6], [e, 7], [i, 3], [n, 0], [b, 5], [a, 1], [k, 8], [o, 9], [s, 4]]
```

jeu("bah+bah=rhum"). → [[h, 4], [m, 8], [a, 3], [u, 6], [b, 7], [r, 1]]
jeu("tenia+du+chien=cenure").
→ [[a, 2], [u, 3], [n, 5], [e, 0], [d, 8], [i, 7], [r, 6], [h, 4], [t, 9], [c, 1]
jeu("cinq+cinq+vingt=trente").
→ [[q, 3], [t, 1], [e, 7], [g, 5], [n, 8], [i, 4], [c, 6], [v, 9], [r, 0]]
jeu("cuire+en+poele=frire").
→ [[e, 9], [n, 1], [l, 0], [r, 2], [i, 7], [o, 5], [u, 6], [c, 3], [p, 4], [f, 8]]
jeu("belier+cancer+balance=taureau").
→ [[r, 1], [e, 7], [u, 9], [c, 4], [a, 8], [i, 2], [n, 0], [l, 3], [b, 5], [t, 6]]
jeu("donald+gerald=robert").
→ [[d, 5], [t, 0], [l, 8], [r, 7], [a, 4], [e, 9], [n, 6], [b, 3], [o, 2], [g, 1]]
jeu("fraise+citron+cerise=raisin").
→ [[e, 0], [n, 5], [s, 1], [o, 4], [i, 6], [r, 9], [t, 8], [a, 7], [c, 3], [f, 2]]
jeu("zwei+drei+vier=neun").
→ [[i, 4], [r, 0], [n, 8], [e, 9], [u, 7], [w, 3], [v, 1], [d, 2], [z, 5]]
jeu("one+nine+fifty+twenty=eighty").
→ [[e, 2], [y, 6], [t, 1], [n, 9], [o, 0], [i, 4], [f, 8], [h, 3], [g, 7], [w, 5]]
jeu("piano+sonata=fifteen").
→ [[o, 4], [a, 8], [n, 2], [t, 3], [e, 6], [i, 0], [p, 7], [f, 1], [s, 9]]
jeu("expense+causes=account").
→ [[e, 4], [s, 6], [t, 0], [n, 1], [u, 8], [o, 2], [p, 7], [a, 5], [c, 3], [x, 9]]
jeu("chien+chasse=gibier").
→ [[e, 1], [n, 2], [r, 3], [s, 0], [i, 4], [h, 6], [a, 9], [b, 5], [c, 7], [g, 8]]
jeu("cuatro+cuatro+cuatro+cuatro+cuatro=veinte").
→ [[o, 9], [e, 5], [r, 6], [t, 4], [n, 3], [a, 0], [i, 2], [u, 7], [c, 1], [v, 8]]
Et puis aussi, l'amusette favorite de J. Pitrat (qui admet trois solutions) :
jeu("ain+aisne+drome+marne=somme").
→ [[e, 5], [n, 0], [i, 9], [m, 3], [r, 6], [o, 8], [s, 7], [a, 1], [d, 2]]
jeu("li+li=cii"). → [[i, 0], [l, 5], [c, 1]]
jeu("pere+mere=bebe").
→ [[e, 0], [r, 2], [b, 4], [m, 1], [p, 3]]
jeu("eve+eve=adam"). → [[e, 6], [m, 2], [v, 5], [a, 1], [d, 3]]
jeu("roue+roue=velo"). → [[e, 3], [o, 6], [u, 5], [l, 0], [r, 4], [v, 9]]
jeu("suede+suisse=bresil").
→ [[e, 9], [l, 8], [s, 4], [d, 7], [i, 2], [u, 6], [r, 0], [b, 5]]
jeu("ne+au+cap=eban").
→ [[e, 1], [u, 2], [p, 3], [n, 6], [a, 4], [c, 9], [b, 0]]
jeu("nasa+has+stars+in=sight").
→ [[a, 0], [s, 2], [n, 1], [t, 5], [i, 6], [r, 9], [h, 7], [g, 8]]

```
jeu("power+for=peace").
→ [[r, 2], [e, 4], [o, 3], [c, 7], [w, 1], [f, 9], [a, 0], [p, 5]]
jeu("send+more=money").
→ [[d, 7], [e, 5], [y, 2], [r, 8], [n, 6], [o, 0], [m, 1], [s, 9]]
jeu("alaska+kansas+texas=states").
→ [[a, 4], [s, 6], [k, 1], [e, 0], [x, 9], [t, 2], [n, 8], [l, 5]]
jeu("clinton+monica=cancans").
→ [[n, 2], [a, 9], [s, 1], [c, 4], [o, 7], [t, 3], [i, 5], [m, 8], [l, 0]]
jeu("bug+bug+bug+bug+bug=win").
→ [[g, 2], [n, 0], [u, 5], [i, 6], [b, 1], [w, 7]]
jeu("gin+fizz=hips").
→ [[z, 2], [n, 1], [s, 3], [i, 8], [p, 0], [g, 9], [f, 4], [h, 5]]
jeu("pierre+peter=pareil").
→ [[r, 6], [e, 8], [l, 4], [i, 5], [t, 1], [p, 3], [a, 9]]
jeu("jean+lisa=amis").
→ [[a, 3], [n, 4], [s, 7], [i, 0], [e, 5], [m, 6], [j, 1], [l, 2]]
jeu("chien+chien=meute").
→ [[n, 3], [e, 6], [t, 2], [i, 0], [u, 1], [h, 8], [c, 4], [m, 9]]
jeu("hasta+la+vista=salut").
→ [[a, 9], [t, 7], [l, 8], [u, 4], [s, 3], [i, 0], [v, 1], [h, 2]]
```



8 Carrés magiques construits par la méthode de Bachet de Méziriac
Le carré sera une liste plate.
Soit P un entier non nul et $N = 2P + 1$ impair, $M = N^2$.
Reconstruire le prédicat *reste(K, D, R)* où R est reste de la division entière de K par D.
Construire le prédicat *init* tel que *init(5, CI).* $\rightarrow CI$ = [0, 0, 0, 0, 0] et le prédicat *placer* qui substitue X au rang R dans une liste. Puis, le carré étant une liste simple (c'est-à-dire plate) de M zéros initialement, construire le prédicat *suiv* donnant le rang *RS* de la case suivante.
On veut maintenant construire des carrés magiques de côté impair N avec les entiers K successifs en débutant par 1 situé juste dans la case sous le milieu, puis en descendant vers la droite de façon toroïdale, sauf si la case est déjà occupée auquel cas on descend de deux lignes dans la même colonne.
Construire le prédicat *construc* qui va faire ce travail pour un K allant de 1 à M. (Il va sans dire que la boucle est à écrire récursivement).

Reste d'une division entière

```
reste(X, D, R) :- X < 0, Y is X + D, reste(Y, D, R), !.
reste(X, D, X) :- X < D, !.
reste(X, D, R) :- D =< X, Y is X – D, reste(Y, D, R), !.
```

```
init(1, [0]) :- !.
init(M, [0 | L]) :- K is M – 1, init(K, L).
```

reste(23, 7, *R*). $\rightarrow R = 2$
reste(–2, 7, *R*). $\rightarrow R = 5$
reste(22, 7, 0). $\rightarrow$ no

Le prédicat *rang* liant la valeur et le rang (à partir de 1) dans une liste :

```
rang(X, 1, [X | _]) :- !.
rang(X, N, [_ | L]) :- K is N – 1, rang(X, K, L).
```

rang(a, 4, [a, b, c]). $\rightarrow$ no
rang(*X*, 2, [b, a, c]). $\rightarrow X$ = a
rang(*X*, 0, [a, b, c]). $\rightarrow$ no

```
placer(X, 1, [_ | LD], [X | LD]) :- !.
placer(X, R, [Y | LD], [Y | LR]) :- K is R − 1, placer(X, K, LD, LR).
```

```
suiv(R, N, 1) :- R is N*N, !.
suiv(R, N, S) :- M is N*(N – 1), R > M, S is R – M + 1, !.
suiv(R, N, S) :- reste(R, N, 0), S is R + 1, !.
suiv(R, N, S) :- S is R + N + 1.
```

```
placer(x, 3, [a, b, c, d, e, f], R). → R = [a, b, x, d, e, f]

suiv(25, 5, RS). → RS = 1
suiv(15, 5, RS). → RS = 16
suiv(21, 5, RS). → RS = 2
suiv(14, 5, RS). → RS = 20
```

```
construc(K, R, N, M, C1, CM) :- placer(K, R, C1, CM), K is N*N, !.
construc(K, R, N, M, C1, CM) :- placer(K, R, C1, C2), suiv(R, N, RS),
        rang(0, RS, C2), !, KS is K + 1, construc(KS, RS, N, M, C2, CM).
construc(K, R, N, M, C1, CM) :- placer(K, R, C1, C2), RS is R + 2*N,
        reste(RS, M, NR), KS is K + 1, construc(KS, NR, N, M, C2, CM).
```

Le prédicat *vue* affichera à l'écran le carré magique obtenu en utilisant *write* et *nl*, et enfin le prédicat *magik* du seul argument *P*.

```
vue(N, N, [X | L]) :- write(X), nl, vue(N, 1, L).
vue(N, K, [X | L]) :- write(X), write(' '), NK is K + 1, vue(N, NK, L).
vue(_, _, []).
```

```
magik(P) :-  N is 2*P + 1, M is N*N, init(M, CI), RD is N*(P + 1) + P + 1,
                construc(1, RD, N, M, CI, CM), vue(N, 1, CM).
```

```
magik(1).
4  9  2
3  5  7
8  1  6

magik(2).
11  24  7  20  3
4  12  25  8  16
17  5  13  21  9
10  18  1  14  22
23  6  19  2  15
```

```
magik(3).
22  47  16  41  10  35  4
5  23  48  17  42  11  29
30  6  24  49  18  36  12
13  31  7  25  43  19  37
38  14  32  1  26  44  20
21  39  8  33  2  27  45
46  15  40  9  34  3  28

magik(4).
37  78  29  70  21  62  13  54  5
6  38  79  30  71  22  63  14  46
47  7  39  80  31  72  23  55  15
16  48  8  40  81  32  64  24  56
57  17  49  9  41  73  33  65  25
26  58  18  50  1  42  74  34  66
67  27  59  10  51  2  43  75  35
36  68  19  60  11  52  3  44  76
77  28  69  20  61  12  53  4  45
```

9 Logigram I
Max a un chat, Eve n'est pas en pavillon, Luc habite un studio mais le cheval n'y est pas. Chacun habite une maison différente et possède un animal distinct. Qui habite le château et qui a le poisson ?

Une première solution d'élégance moyenne est donnée ici, son avantage est de suivre assez fidèlement l'énoncé. Une autre solution figure au chapitre suivant. Le prédicat *rel* constitue la relation entre un personnage, son animal et sa maison. Il suffit de mettre les inconnues, la difficulté est quand même d'indiquer une relation injective par le prédicat *dif*.

```
maison(chateau)
maison(studio)
maison(pavillon).
animal(chat).
animal(poisson).
animal(cheval).

rel(max, M, chat) :- maison(M).
rel(luc, studio, A) :- animal(A), A \== cheval.
rel(eve, M, A) :- maison(M), M \== pavillon, animal(A).
```

```
dif(X, X, _) :- !, fail.
dif(X, _, X) :- !, fail.
dif(_, X, X) :- !, fail.
dif(_, _, _).

reso(MM, ME, AE, AL) :- rel(max, MM, chat), rel(eve, ME, AE),
                        dif(MM, ME, studio), rel(luc, studio, AL), dif(AE, AL, chat).
```

reso(*MM*, *ME*, *AE*, *AL*).
→ *MM* = pavillon *ME* = chateau *AE* = cheval *AL* = poisson

10 Sudoku
Le jeu consiste à construire des carrés de 9 sur 9 où chaque ligne et chaque colonne contiennent exactement les neuf chiffres de 1 à 9, mais aussi chacun des 9 sous-carrés de 3 sur 3. Après cela, il serait facile de ne laisser que quelques cases de façon à créer des jeux plus ou moins faciles à compléter.

Contrairement aux carrés magiques, la dimension étant fixée, nous adoptons une représentation en une liste de listes représentant les lignes, cela a toutefois l'inconvénient de devoir écrire des arguments assez longs.

```
app(X, [X | _]).                    % prédicat d'appartenance
app(X, [_ | L]) :- app(X, L).

napp(_, []).                        % prédicat de non-appartenance
napp(X, [X | _]) :- !, fail.
napp(X, [_ | L]) :- napp(X, L).

suiv(I, J, I, JS) :- J < 9, JS is J + 1.
        % (IS, JS) est la case suivante de (I, J) en parcourant de gauche à
        % droite et de haut en bas
suiv(I, 9, IS, 1) :- IS is I + 1.

nth(1, [L | _ ], L) :- !.
nth(I, [_ | S], L) :- IS is I − 1, nth(IS, S, L).

colonne(_, [], []).
colonne(J, [L | S], [E | C]) :- nth(J, L, E), colonne(J, S, C).

compte(1, X, [X]).
compte(N, X, [X | L]) :- 1 < N, NM is N – 1, compte(NM, X, L).
```

Le prédicat *colonne(J, S, C)* est vrai dès lors que *C* est la colonne de rang *J* de la matrice *S*. Le prédicat *carre(I, J, S, M)* est vrai dès lors que *M* est la sous-matrice où se trouve (*I, J*).

```
carre(I, J, [[A, B, C | _], [D, E, F | _], [G, H, K | _] | _],
        [A, B, C, D, E, F, G, H, K]) :- I < 4, J < 4.

carre(I, J, [[_, _, _, A, B, C | _], [_, _, _, D, E, F | _], [_, _, _, G, H, K | _] | _],
        [A, B, C, D, E, F, G, H, K])
        :- I < 4, 3 < J, J < 7.
```

```
carre(I, J, [[_,_,_,_,_,_,A,B,C], [_,_,_,_,_,_,D,E,F],
        [_,_,_,_,_,_,G, H, K] | _], [A, B, C, D, E, F, G, H, K])
        :- I < 4, 6 < J.

carre(I, J, [_, _, _, [A, B, C | _], [D, E, F | _], [G, H, K | _] | _],
        [A, B, C, D, E, F, G, H, K])
        :- 3 < I, I < 7, J < 4.

carre(I, J, [_, _, _, [_, _, _, A, B, C | _], [_, _, _, D, E, F | _],
        [_, _, _, G, H, K | _] | _], [A, B, C, D, E, F, G, H, K])
        :- 3 < I, I < 7, 3 < J, J < 7.

carre(I, J, [_, _, _, [_, _, _, _, _, _, A, B, C], [_, _, _, _, _, _, D, E, F],
        [_, _, _, _, _, _, G, H, K] | _], [A, B, C, D, E, F, G, H, K])
        :- 3 < I, I < 7, 6 < J.

carre(I, J, [_, _, _, _, _, _, [A, B, C | _], [D, E, F | _], [G, H, K | _]],
        [A, B, C, D, E, F, G, H, K])
        :- 6 < I, J < 4.

carre(I, J, [_, _, _, _, _, _, [_ ,_ ,_ ,A ,B ,C | _], [_, _, _, D, E, F | _],
        [_, _, _, G, H, K | _]], [A, B, C, D, E, F, G, H, K])
        :- 6 < I, 3 < J, J < 7.

carre(I, J, [_, _, _, _, _, _, [_, _, _, _, _, _, A, B, C], [_, _, _, _, _, _, D, E, F],
        [_, _, _, _, _, _, G, H, K] ], [A, B, C, D, E, F, G, H, K])
        :- 6 < I, 6 < J.

place(X, 1, 1, [[_ | L] | S], [[X | L] | S]) :- !.
place(X, 1, J, [[Y | L] | S], [[Y | NL] | S])
        :- 1 < J, JM is J − 1, place(X, 1, JM, [L | S], [NL | S]).
place(X, I, J, [L | S], [L | NS])
        :- 1 < I, IM is I − 1, place(X, IM, J, S, NS).
```

```
place(z, 3, 5, [[a, b, c, d, e, f], [g, h, i, j, k, l], [m, n, o, p, q, r],
[s, t, u, v, w, x]], X).
→ X = [[a, b, c, d, e, f], [g, h, i, j, k, l], [m, n, o, p, z, r], [s, t, u, v, w, x]]
colonne(5, [[a, b, c, d, e, f], [g, h, i, j, k, l], [m, n, o, p, q, r],
[s, t, u, v, w, x]], X). → X = [e, k, q, w]
carre(3, 5, [[a, b, c, d, e, f], [g, h, i, j, k, l], [m, n, o, p, q, r],
[s, t, u, v, w, x]], X). → X = [d, e, f, j, k, l, p, q, r]
```

```
vue([]).
vue([L | S]) :- write(L), nl, vue(S).

construire(10, 1, S, S) :- vue(S).
construire(I, J, S, R) :- carre(I, J, S, M), nth(I, S, L), colonne(J, S, C),
        app(X, [1, 2, 3, 4, 5, 6, 7, 8, 9]), napp(X, L), napp(X, M), napp(X, C),
        place(X, I, J, S, NS), suiv(I, J, IS, JS), construire(IS, JS, NS, R).

sudoku(R) :- compte(9, 0, L), compte(9, L, S), construire(1, 1, S, R).
```

sudoku(*R*). →
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[4, 5, 6, 7, 8, 9, 1, 2, 3]
[7, 8, 9, 1, 2, 3, 4, 5, 6]
[2, 1, 4, 3, 6, 5, 8, 9, 7]
[3, 6, 5, 8, 9, 7, 2, 1, 4]
[8, 9, 7, 2, 1, 4, 3, 6, 5]
[5, 3, 1, 6, 4, 2, 9, 7, 8]
[6, 4, 2, 9, 7, 8, 5, 3, 1]
[9, 7, 8, 5, 3, 1, 6, 4, 2]

Qui n'est que le premier d'une très longue série, il suffit de changer l'ordre des chiffres dans *app* ou de le rendre aléatoire pour obtenir une multitude de départs différents.
Pour résoudre un sudoku, il faudrait appeler le programme, non pas avec un *S* formé de zéro mais avec la donnée du problème puis appeler *construire(1, 1, S, R).* et rajouter une condition $S_{i,j} > 0$ imposant le passage à la case suivante afin de ne pas détruire la donnée du problème.

11 Carrés diaboliques
Un tel carré doit avoir toutes ses lignes, toutes ses colonnes et toutes ses diagonales et codiagonales modulo *N*, de somme *S*. Un carré *N*N* formé par des entiers est représenté par un quadruplet [*LG*, *CL*, *DG*, *AG*] de listes de listes : lignes, colonnes, diagonales, codiagonales. On cherche les carrés « diaboliques » de dimension *N*, de somme *S* donnée ou non avec des entiers choisis on non dans une liste *CH*.

```
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).

ret(X, [X | L], L) :- !.
ret(X, [Y | L], [Y | R]) :- ret(X, L, R).

compte(1, [1]) :- !.
compte(N, [N | L]) :- K is N – 1, compte(K, L).

initbis(0, []) :- !.
initbis(N, [0 | L]) :- K is N – 1, initbis(K, L).
init(0, _, []) :- !.
init(NL, NC, [L | R]) :- initbis(NC, L), K is NL – 1, init(K, NC, R).

suiv(I, J, N, I, JS) :- J < N, JS is J + 1, !.
suiv(I, N, N, IS, 1) :- IS is I + 1.
```

compte(12, *X*). → *X* = [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
init(3, 4, *M*). → *M* = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
suiv(3, 4, 5, *I*, *J*). → *I* = 3 *J* = 5
suiv(3, 5, 5, *I*, *J*). → *I* = 4 *J* = 1

```
vue([]) :- nl.
vue([[X | R] | L]) :- write(X),  write(‘  ’), vue([R | L]).
vue([[] | L]) :- nl, vue(L).

rang(1, X, [X | _]) :- !.
rang(R, X, [_ | L]) :- RS is R − 1, rang(RS, X, L).
diag(I, J, N, K) :- J < I, K is N – I + J + 1, !.
diag(I, J, N, K) :- K is J – I + 1.
codiag(I, J, N, K) :- K is I + J – N – 1, 0 < K, !.
codiag(I, J, N, K) :- K is I + J – 1.
```

rang(4, a, [a, b, c]). → no
rang(2, *X*, [b, a, c]). → *X* = a
rang(0, *X*, [a, b, c]). → no
diag(3, 1, 4, *K*). → *K* = 3
diag(3, 2, 4, *K*). → *K* = 4
codiag(3, 1, 4, *K*). → *K* = 3
codiag(4, 1, 4, *K*). → *K* = 4
codiag(4, 2, 4, *K*). → *K* = 1

```
placer(X, 1, [_ | L], [X | L]) :- !.
placer(X, R, [Y | L], [Y | NL]) :- RS is R − 1, placer(X, RS, L, NL), !.
placer(X, I, J, M, NM) :- rang(I, L, M), placer(X, J, L, NL),
                          placer(NL, I, M, NM).

nth(1, [E | _], E) :- !.
nth(X, [_ | L], E) :- K is X − 1, nth(K, L, E).

poss([], 0).
possbis([], _).
poss([0 | L], S) :- 0 < S, SL is S − 1, possbis(L, SL).
poss([X | L], S) :- 0 < X, X =< S, SL is S – X, poss(L, SL).
possbis([X | L], S) :- X =< S, SL is S – X, possbis(L, SL).
```

placer(x, 3, [a, b, c, d, e, f], *R*). → *R* = [a, b, x, d, e, f]
placer(x, 3, 2, [[a, z, e, r, t, y], [u, i, o, p], [j, k, l, m], [q, s, d, f]], *R*).
→ *R* = [[a, z, e, r, t, y], [u, i, o, p], [j, x, l, m], [q, s, d, f]]
nth(2, [a, b, c, d, e, f], *E*). → *E* = b
poss([2, 3, 0, 5, 1], 12). → yes
poss([2, 3, 5], 10). → yes
poss([2, 3, 0, 5, 1], 11). → no
poss([0], 0). → no
poss([14, 2, 4, 10], 34). → no
poss([16, 0, 0, 0], 34). → yes

```
suite(N, N, N, _, _, LG, _, _, _) :- vue(LG), !.
suite(I, J, N, S, CH, LG, CL, DG, AG) :- app(X, CH), suiv(I, J, N, IS, JS),
                                         construc(X, IS, JS, N, S, CH, LG, CL, DG, AG).
```

On place *X* dans sa ligne, colonne, diagonale et antidiagonale en vérifiant au fur et à mesure que cela est possible.

```
construc(X, I, J, N, S, CH, LG, CL, DG, AG) :-        % X placé en (I, J)
        placer(X, I, J, LG, NLG), nth(I, NLG, NL), poss(NL, S),
        placer(X, J, I, CL, NCL), nth(J, NCL, NC), poss(NC, S),
        diag(I, J, N, K), placer(X, K, I, DG, NDG), nth(K, NDG, ND),
        poss(ND, S), codiag(I, J, N, G), placer(X, G, I, AG, NAG),
        nth(G, NAG, NA), poss(NA, S), ret(X, CH, NCH),
        % si X compatible dans les 4 directions, NCH = nouv choix
        suite(I, J, N, S, NCH, NLG, NCL, NDG, NAG).

diab(N) :- M is N*N, compte(M, CH), S is N*(M + 1)/2, nl, diab(N, S, CH).
diab(N, S, CH) :- app(X, CH), init(N, N, LG),
        construc(X, 1, 1, N, S, CH, LG, LG, LG, LG).
```

C'est le carré indien de Khajuraho (à gauche), avec les mêmes entiers, les 12 sommes égales à 34, celui de Dürer étant à droite :

diab(4). →

```
16  9   7   2          16   3   2  13
 5   4  14  11           5  10  11   8
10  15   1   8           9   6   7  12
 3   6  12  13           4  15  14   1
```

diab(4, 34, [14, 11, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 16]).
→

```
14  11   2   7
 1   8  13  12
15  10   3   6
 4   5  16   9
```

12 Vérification d'un carré magique

Dans le but de construire des carrés magiques par apprentissage, ou bien de vérifier si un carré est magique, si celui-ci est représenté par la liste de tous ses éléments ligne par ligne, on commence par vérifier le caractère magique de ses colonnes. Dans tout ce qui suit, le carré d'ordre *N* est donc représenté par une liste d'entiers de longueur $M = N^2$. La somme de ses lignes, colonnes et diagonales est $S = N(M + 1)/2$. Rappelons qu'au chapitre 2, on a pu compter les éléments de *K* en *K* dans une liste d'entiers. *KS* est le successeur de *K*, *KP* son prédécesseur, *SP* désigne toujours une somme partielle, *NSP* la nouvelle somme après un ajout, etc.
Pour *somk* d'une liste, on mettra les deuxième et troisième paramètres *K* en *K*, ce prédicat calcule la somme des éléments d'une liste de *K* en *K*. Le prédicat *somcol* permet donc d'avoir la somme des écarts des sommes de colonnes avec *S*, enfin *somld* fera de même pour les lignes et les deux diagonales. Dans *somcol* et dans *somld* ce paramètre *K* partira de 0.

```
somk([], _, _, R, R). % liste épuisée
somk([X | LR], K, K, SP, R) :- SS is SP + X, somk(LR, 1, K, SS, R).
                               % ajout si rang N
somk([X | LR], P, K, SP, R) :- P < K, PS is P + 1, somk(LR, PS, K, SP, R).
                               % avancée

somcol(_, N, N, _, R, R):- !.
somcol([X | L], K, N, S, SP, R) :- somk([X | L], N, N, 0, SC),
                               NSP is SP + abs(S − SC), NK is K + 1,
                               somcol(L, NK, N, S, NSP, R).
```

somk([1, 2, 3, 0, 4, 5, 3, 2, 5, 1, 1, 7, 0, 3, 8, 5], 3, 3, 0, *R*). → *R* = 10
somk([1, 2, 3, 0, 4, 5, 3, 2, 5, 1, 1, 7, 0, 3, 8, 5], 2, 2, 0, *R*). → *R* = 25
somcol([4, 8, 3, 1, 7, 9, 5, 2, 6], 0, 3, 15, 0, *R*). → *R* = 10
somcol([6, 1, 8, 7, 5, 3, 2, 9, 4], 0, 3, 15, 0, *R*). → *R* = 0

Maintenant, si on veut calculer les écarts pour les lignes et les deux diagonales, l'affaire est plus complexe.
On va utiliser le symbole de Kronecker δ_{ij} = *si i = j alors 1 sinon 0* avec la relation *kro*, puis en parcourant toute la liste, chaque fois que l'indice *K* termine une ligne, on ajoute à la somme des lignes *SL* le nouvel écart d'avec *S*, et chaque fois que l'indice passe sur une diagonale, *D1* ou *D2* sont incrémentés.

```
kro(X, X, 1) :- !.
kro(_, _, 0).
```

```
somld([], _, _, S, _, SL, D1, D2, R) :- R is SL + abs(S – D1)+abs(S – D2).
        % liste épuisee

somld([X | L], K, N, S, SPL, SL, D1, D2, R) :- I is K // N, J is K mod N,
        NP is N – 1, kro(J, NP, NL), kro(I, J, AD1), JD is N – 1 – J,
        kro(I, JD, AD2), NK is K + 1, NSPL1 is SPL + X, JS is NK mod N,
        kro(JS, 0, NL), NSL is SL + NL*abs(S – NSPL1),
        ND1 is D1 + AD1*X, ND2 is D2 + AD2*X,
        NSPL is (NSPL1)*(1 – NL),
        somld(L, NK, N, S, NSPL, NSL, ND1, ND2, R).
```

somld([4, 8, 3, 1, 7, 9, 5, 2, 6], 0, 3, 15, 0, 0, 0, 0, *R*). → *R* = 6
somld([6, 1, 8, 7, 5, 3, 2, 9, 4], 0, 3, 15, 0, 0, 0, 0, *R*). → *R* = 0

eval([4, 8, 3, 1, 7, 9, 5, 2, 6], *F*). → *F* = 16
eval([6, 1, 8, 7, 5, 3, 2, 9, 4], *F*).
→ *F* = 0 % toujours le carré chinois
et encore pour vérification :
eval([3, 0, 1, 4, 2, 1, 5, 2, 3, 3, 0, 1, 1, 7, 6, 8], *F*). → *F* = 221

13 Application à la vérification d'un carré alphamagique

Un carré alphamagique est un carré magique où les nombres, s'ils sont écrits en toutes lettres en français, fournissent un autre carré magique en comptant le nombre de lettres utilisées à chaque fois. Par exemple dans le carré ci-dessous, les sommes sont 336 et 27.

15	206	115
212	112	12
109	18	209

quinze	deux cent six	cent quinze
deux cent douze	cent douze	douze
cent neuf	dix-huit	deux cent neuf

6	11	10
13	9	5
8	7	12

Ici, on ne s'occupe que de construire le carré des mots et de leur compte. Ce problème peut servir à leur recherche par algorithme évolutionnaire (chapitre 8). On utilise les prédicats déjà vus de concaténation *conc*, de retrait *ret* et de traduction de chaîne en liste *name*.
Pour la traduction littérale, ce problème a été vu en japonais (bien plus simple qu'en français) et en anglais (chapitre 3). Nous reprenons donc toutes les difficultés (mais sans le pluriel des centaines) :

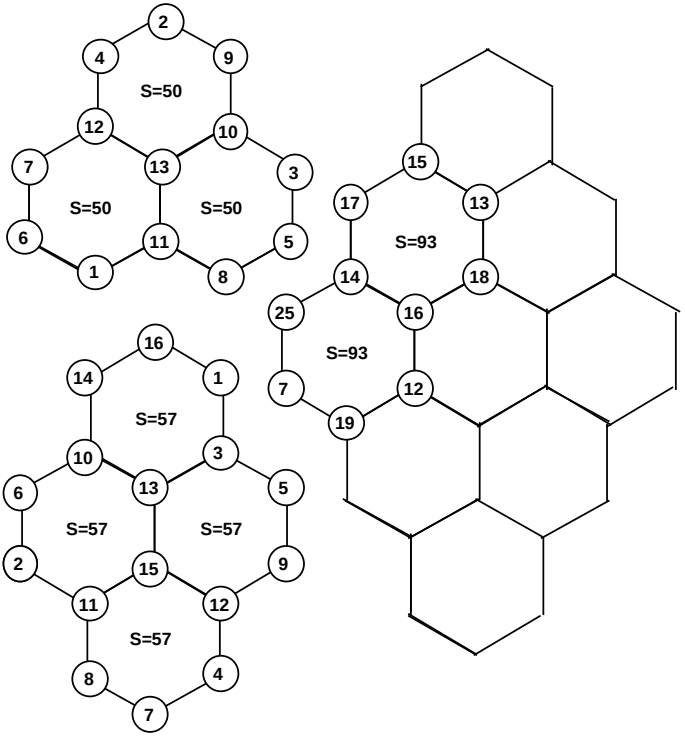
```
mot(0, []). mot(1, "un "). mot(2, "deux "). mot(3, "trois "). mot(4, "quatre ").
mot(5, "cinq "). mot(6, "six "). mot(7, "sept "). mot(8, "huit "). mot(9, "neuf ").
mot(10, "dix "). mot(11, "onze "). mot(12, "douze "). mot(13, "treize ").
mot(14, "quatorze "). mot(15, "quinze "). mot(16, "seize "). mot(20, "vingt ").
mot(30, "trente "). mot(40, "quarante "). mot(50, "cinquante ").
mot(60, "soixante "). mot(80, "quatre-vingt ").

mot(N, A) :- chiffres(N, M, C, D, U), mille(M, ML), cent(C, CL),
        diz(D, U, DL), conc(CL, DL, B), conc(ML, B, A).

chiffres(N, M, C, D, U) :- M is N // 1000, R is N mod 1000, C is R // 100,
                    S is R mod 100, D is S // 10, U is S mod 10.
mille(0, []).
mille(1, "mille ").
mille(M, R) :- mot(M, ML), conc(ML, "mille ", R).

cent(0, []).
cent(1, "cent ").
cent(C, R) :- mot(C, CL), conc(CL, "cent ", R).

diz(D, U, R) :- (D = 7 ; D = 9), NU is U + 10, ND is D − 1, diz(ND, NU, R).
diz(1, U, R) :- 6 < U, !, mot(U, UL), conc("dix ", UL, R).
diz(1, U, R) :- ND is 10 + U, mot(ND, R).
diz(D, U, R) :- (U = 1 ; U = 11), D \= 8, ND is 10*D, mot(ND, DL),
                    mot(U, UL),   conc(DL, "et ", D0), conc(D0, UL, R).
diz(D, U, R) :- ND is 10*D, mot(ND, DL), mot(U, UL), conc(DL, UL, R).

maplit([], [], []).
maplit([N | L], [M | LL], [X | LN]) :- mot(N, AS), name(M, AS),
                    ret(32, AS, ME), length(ME, X), maplit(L, LL, LN).
```

mot(300, *X*), name(*S*, *X*). → *S* = ‘trois cent ‘
mot(271, *X*), name(*S*, *X*). → *S* = ‘deux cent soixante et onze ‘
mot(3781, *X*), name(*S*, *X*). → *S* = ‘trois mille sept cent quatre vingt un ‘
mot(7050, *X*), name(*S*, *X*). → *S* = ‘sept mille cinquante ‘
maplit([15, 206, 115, 212, 112, 12, 109, 18, 209], *LL*, *LN*).
→ *LL* = [‘quinze ‘, ‘deux cent six ‘, ‘cent quinze ‘, ‘deux cent douze ‘, ‘cent douze ‘, ‘douze ‘, ‘cent neuf ‘, ‘dix huit ‘, ‘deux cent neuf ‘]
LN = [6, 11, 10, 13, 9, 5, 8, 7, 12]

14 Tortue magique

Il faut placer les entiers consécutifs dans une « tortue » dont chaque hexagone aura pour somme *S* donnée (voir dessin). Chacun de ces hexagones sera représenté par une liste débutant en haut dans le sens trigonométrique, par exemple [15; 17; 14; 16; 18; 13] et [14; 25; 7; 19; 12; 16] auront un côté commun. Une tortue sera donc une liste d'alvéoles hexagonales. Prévoir les prédicats d'appartenance, de retrait de substitution, et de vérification sur la possibilité de continuer à remplir une alvéole pour atteindre une somme *S*. La tortue étant partiellement emplie de nombres, chaque alvéole ayant une somme inférieure ou égale à *S*, il faut méthodiquement chercher le prochain sommet à substituer par un nombre parmi ceux qui restent. On s'arrête toujours à la première solution rencontrée.

Tester sur le premier exemple avec 3 hexagones, puis sur les 4 hexagones de la petite tortue dont chaque alvéole doit faire 57. Problème du coréen Suk Jung Choi (1646-1715), pour une résolution par algorithme génétique, voir [Seung Kyu Lee, Dong Il Seo, Byung Moon, *A hybrid genetic algorithm for optimal hexagonal tortoise*, Proc. of GECCO, p. 689, 2002]



Substitution de *A* par *X* dans une liste :

```
subst(A, X, [[A | AL] | H], [[X | AL] | NH]) :- !, subst(A, X, H, NH).
subst(A, X, [[B | AL] | H], [[B | NA] | NH])
        :- subst(A, X, [AL | H], [NA | NH]).
subst(A, X, [[] | H], [[] | NH]) :- subst(A, X, H, NH).
subst(_, _, [], []).
```

Le prédicat *hexagone* est vérifié par une seule structure avec des lettres à « affecter » par exemple pour 4 alvéoles :

```
hexag([[a, b, c, d, e, f], [e, d, g, h, i, j], [c, k, l, m, g, d], [g, m, r, s, n, h]]).
```

Intervalle des entiers de 1 à *n* :

```
interv(1, [1]) :- !.
interv(N, [N | I]) :- K is N − 1, interv(K, I).
```

Appartenance à une liste :

```
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).
```

Retrait d'un élément d'une liste :

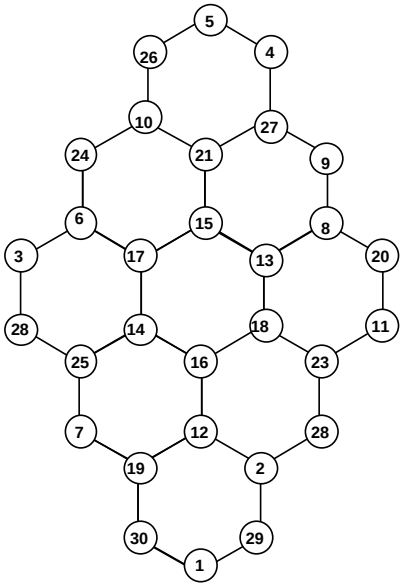
```
ret(X, [X | L], L) :- !.
ret(X, [Y | L], [Y | R]) :- ret(X, L, R).
```

Exemples

```
interv(7, I). → I = [7, 6, 5, 4, 3, 2, 1]

subst(d, 3, [[a, b, c, d, e, f], [e, d, g, h, i, j], [c, k, l, m, g, d],
  [i, h, n, o, p, q]], H).
→ H = [[a, b, c, 3, e, f], [e, 3, g, h, i, j], [c, k, l, m, g, 3], [i, h, n, o, p, q]]
```

Une solution de somme 93 pour 9 alvéoles avec la figure ci-après :

```
hexag( [[a, b, c, d, e, f], [e, d, g, h, i, j], [c, k, l, m, g, d], [i, h, n, o, p, q],
        [g, m, r, s, n, h], [l, t, u, v, m, r], [n, s, w, x, y, o], [r, v, z, aa, w, s],
        [w, aa, bb, cc, dd, x]]). %  (9 alvéoles)
```



On obtient cette figure en lançant la requête :
jeu(93, 30).
→ [[30, 29, 28, 3, 2, 1], [2, 3, 27, 26, 25, 10], [28, 23, 8, 4, 27, 3], [25, 26, 15, 16, 6, 5], [27, 4, 14, 7, 15, 26], [8, 22, 21, 24, 4, 14], [15, 7, 20, 18, 17, 16], [14, 24, 9, 19, 20, 7], [20, 19, 13, 12, 11, 18]]

Le prédicat *prochain* est utilisé pour obtenir la première lettre à affecter :

```
prochain([[] | H], X) :- prochain(H, X).
prochain([[P | A] | H], X) :- integer(P), !, prochain([A | H], X).
prochain([[P | AL] | H], P).
possible([A | AL], S1, INC) :- number(A), !, S2 is S1 – A, possible(AL, S2, INC).
possible([A | AL], S1, INC1) :- S2 is S1 – 1, INC2 is INC1 + 1,
                                possible(AL, S2, INC2).
possible([], S, INC) :- INC > 0, S >= 0.
possible([], 0, 0).
```

prochain([[1, 2, 3], [5, 1], [5, 7], [i, 5, 3, o, 2, q], [g, m, r, s, n, h], [8, u, v, m, r]], *X*). → *X* = i
possible([1, 2, 3], 6, 0). → yes
possible([a, 1, b, 2, 3], 7, 0). → no
possible([a, 1, b, 2, 3], 8, 0). → yes

```
verif([AL | H], S) :- possible(AL, S, 0), verif(H, S).
verif([], _).

remplir(H, CH, S) :- prochain(H, A), !, app(X, CH), subst(A, X, H, NH),
                     verif(NH, S), ret(X, CH, NCH), remplir(NH, NCH, S).
remplir(H, _, _) :- write(H), nl.
jeu(S, N) :- interv(N, CH), hexag(H), remplir(H, CH, S).
```

Le prédicat *hexagone* est vérifié par une seule structure avec des lettres à « affecter » par exemple pour 4 alvéoles :

```
hexag([[a, b, c, d, e, f], [e, d, g, h, i, j], [c, k, l, m, g, d], [g, m, r, s, n, h]]).
```

Exemples

```
verif([[1, 2, 3], [5, 1], [a, 3], [1, 3, b], [a, 2, b], [1, 2, a, b, c]], 6).
→ yes
verif([[1, 2, 3], [5, 1], [a, 3], [5, 3, b]], 6). → no

jeu(57, 16).
→ [[16, 14, 10, 13, 3, 1], [3, 13, 15, 12, 9, 5], [10, 6, 2, 11, 15, 13],
[15, 11, 8, 7, 4, 12]]
```

Pavage hexagonal magique sur un tore :

```
hexag([[a, b, n, k, j, i], [b, c, d, e, o, n], [e, f, p, m, l, o], [f, g, h, a, i, p],
[h, j, k, c, b, a], [n, o, l, d, c, k], [l, m, g, f, e, d], [m, p, i, j, h, g]]).

% 8 alvéoles sur un tore (le dessin n'est pas facile du tout)
```

```
jeu(51, 16).
→ [[16, 15, 8, 4, 3, 5], [15, 7, 10, 2, 9, 8], [2, 1, 12, 14, 13, 9],
[1, 11, 6, 16, 5, 12], [6, 3, 4, 7, 15, 16], [8, 9, 13, 10, 7, 4],
[13, 14, 11, 1, 2, 10], [14, 12, 5, 3, 6, 11]]
```

Soit la figure suivante sur le tore :