Exercice7

Le problème du sac à dos fait partie des problèmes classiques de la Recherche Opérationnelle.
Etant donnés :
— un sac à dos de volume total b,
— n objets tels que chaque objet i possède :
— un volume a_i,
— une utilité c_i
On souhaite remplir le sac en maximisant l'utilité des objets qu'on y met. On fait ici l'hypothèse que les coefficients c_i , a_i et b sont positifs.

Exercice8

Le problème du Sudoku

	7							6
				6	9		3	
5				7			4	8
4				9		7		
							6	
8	6		1	3				
6		8					5	3
			9			8		
9	4			1	3	6	7	2

Dans le problème du sudoku, il s'agit de remplir les cases par un chiffre allant de 1 à 9, en respectant plusieurs contraintes : un seul même chiffre par ligne, par colonne et par bloc de 3x3. A priori, il n'y a pas de maximisation ni de minimisation : si l'on respecte les contraintes, toutes les lignes, toutes les colonnes et tous les blocs de 3x3 font la même somme. Il n'y a pas non plus de données externes, si l'on indique les cases sous forme de contraintes. Donc tout le problème réside dans la formulation des contraintes.

Correction exercice7
Formulation : En choisissant comme variables : x_i qui vaut 1 si l'objet i est mis dans le sac et 0 sinon, le problème peut être modélisé par le programme linéaire en nombres entiers suivant :

$$(SAD)\begin{cases} \max \quad \sum_{i=1}^{n} c_i x_i \\ \sum_{i=1}^{n} a_i x_i \leqslant b \\ x \in \{0,1\}^n \end{cases}$$

1. Après avoir compris le codage du sac à dos ci-dessus, récupérez le fichier qui représente le modèle exo1.mod .

```
#modele de sac a dos
#donnees
param n ; #nombre d'objets
param C{i in 1..n}; #utilité de l'objet i
param A{i in 1..n}; #poids de l'objet i
param B; #capacité du sac
#variables
var x{1..n} binary;
#objectif
maximize f :sum {i in 1..n} C[i]*x[i] ;
#contraintes
subject to
capacite : sum{i in 1..n} A[i]*x[i] <= B ;
printf "------Debut de la resolution -----\n";
solve;
printf "------Fin de la resolution -----\n";
display x;
end;
```

2. Récupérez le fichier de données associé exo1.dat :

```
#un fichier de donnees pour le probleme de sac a dos

data;
param n := 5;
param C := 1 12
2 15
3 5
4 16
5 17;
param A := 1 2
2 6
3 1
4 7
5 8;
param B := 20;
end;
```

3. Pour résoudre le programme mathématique obtenu par juxtaposition du modèle et des données, il faut exécuter la commande :
glpsol -m sac_a_dos.mod -d sac_a_dos.dat

correction exercice8 :

Dans notre fichier de programmation linéaire au format GMPL, que l'on nomme **sudoku.mod**, on commence par définir une variable binaire xijk, qui prend la valeur 1 si la case de ligne *i* allant de 1 à 9 et de colonne *j* de 1 à 9 également contient le chiffre *k* qui va de 1 à 9.

```
var x{i in 1..9, j in 1..9, k in 1.. 9 } binary;
```

On définit ensuite les contraintes.

```
subject to
```

Un seul chiffre par case, pour éviter d'avoir plusieurs chiffres *k* pour une même case ij : la somme des valeurs de xijk doit être égale à 1, et ce pour toutes les cases ij :

```
valeurs {i in 1..9, j in 1.. 9}: sum {k in 1..9} x[i,j,k]
= 1;
```

Pour qu'il n'y ait qu'un même chiffre *k* sur toute la ligne *i*, on ajoute une contrainte qui postule que la somme du nombre de fois qu'un chiffre *k* donné apparaît sur la ligne *i* est égale à 1. Cela se traduit par :

```
lignes {i in 1..9, k in 1.. 9}: sum {j in 1..9} x[i,j,k] =
1;
```

On fait de même pour les colonnes *j* :

```
colonnes {j in 1..9, k in 1.. 9}: sum {i in 1..9} x[i,j,k]
= 1;
```

Pour éviter que l'on ait plusieurs fois le même chiffre dans un bloc 3x3, on ajoute une contrainte de bloc. Le nombre de fois que le chiffre *k* apparaît dans un bloc 3x3 doit être égal à 1, ce qui nécessite de parcourir les lignes *i* et les colonnes *j* par 3. Pour ce faire, on utilise deux petites variables *xx* et *yy* qui prennent successivement les valeurs 0, 3 et 6, ainsi on pourra parcourir les lignes *i* de *xx+1* à *xx+3* et les colonnes *j* de *yy+1* à *yy+3*, pour couvrir les 9 blocs 3x3 du sudoku. Ce qui nous donne :

```
blocs {xx in {0, 3, 6}, yy in {0, 3, 6}, k in 1..9}:
    sum {i in (xx+1)..(xx+3), j in (yy+1)..(yy+3)} x[i,j,k
] = 1;
```

Il nous reste à indiquer les contraintes des cases données, en forçant xijk pour un case ij. Et après calcul, on affiche le résultat sous forme de grille en s'aidant de boucles **for** et de fonctions **printf** (GMLP). On obtient au final le fichier **sudoku.mod** suivant :

```
#variable xijk=1 si case ij=k
```

```
var x{i in 1..9, j in 1..9, k in 1.. 9 } binary;

#contraintes

subject to

#un seul chiffre par case

valeurs {i in 1..9, j in 1.. 9}: sum {k in 1..9} x[i,j,k]
= 1;

#un meme chiffre par ligne

lignes {i in 1..9, k in 1.. 9}: sum {j in 1..9} x[i,j,k] =
1;

#un meme chiffre par colonne

colonnes {j in 1..9, k in 1.. 9}: sum {i in 1..9} x[i,j,k]
= 1;

#un meme chiffre par blocs de 3x3

blocs {xx in {0, 3, 6}, yy in {0, 3, 6}, k in 1..9}:

    sum {i in (xx+1)..(xx+3), j in (yy+1)..(yy+3)} x[i,j,k
] = 1;

#cases donnees

c1: x[1,2,7]=1; c2: x[1,9,6]=1;

c3: x[2,5,6]=1; c4: x[2,6,9]=1; c5: x[2,8,3]=1;

c6: x[3,1,5]=1; c7: x[3,5,7]=1; c8: x[3,8,4]=1; c9: x[3,9,
8]=1;

c10: x[4,1,4]=1; c11: x[4,5,9]=1; c12: x[4,7,7]=1;

c13: x[5,8,6]=1;

c14: x[6,1,8]=1; c15: x[6,2,6]=1; c16: x[6,4,1]=1; c17: x[
6,5,3]=1;

c18: x[7,1,6]=1; c19: x[7,3,8]=1; c20: x[7,8,5]=1; c21: x[
7,9,3]=1;

c22: x[8,4,9]=1; c23: x[8,7,8]=1;

c24: x[9,1,9]=1; c25: x[9,2,4]=1; c26: x[9,5,1]=1; c27: x[
9,6,3]=1; c28: x[9,7,6]=1; c29: x[9,8,7]=1; c30: x[9,9,2]=
1;

#calcul
```

```
solve;
#affichage
printf "\n\nSolution : \n";
printf "-----------------------------------\n";
for {i in 1.. 9} {
    for {j in 1..9} {
         for {k in 1..9 :x[i,j,k] ==1} printf " %d ", k;
         printf "|";
    }
    printf "\n";
    printf "-----------------------------------\n";
}
end;
```

Je vous laisse découvrir la solution.