TP BACKTRACKING (DÉBUT)

Syntaxe Python :

- `id` est un opérateur qui appliqué à une variable `x`, renvoie son adresse en mémoire : `id(x)`
- En python toute affectation provoque une nouvelle implantation en mémoire (vérifiable avec id):

  `x=3 → adresse de x : 137396032`

  `x=4 → adresse de x : 137396048`

  `L=[1,2] → adresse de L : 155228140`

  `L=[1,2,3] → adresse de L : 155159244`

- Pour modifier le contenu d'une liste sans la réallouer :
  `L[:]=[1,2,3,4,5] → adresse de L : 155159244 non modifiée`
  `L=[1,2,3,4,5] → l'adresse de L est modifiée, réallocation et lenteur accrue !!`

- Pour compter des événements lors de l'exécution d'une fonction, on peut utiliser une variable globale :
  dans la fonction, on écrit :
  ```
  global cpt
  if cond :
     cpt+=1
  ```
  et on initialise hors de la fonction :
  ```
  cpt=0
  ```

- Pour s'arrêter net lors de l'exécution, on peut lever une exception :
  ```
  raise Exception ('' Solution trouvée !'')
  ```

- Pour sauvegarder les solutions obtenues par backtracking dans une liste :
  une fois trouvée une solution `sol`, on l'ajoute à la liste `L_sol` avec :
  ```
  L_sol.append(sol[:])
  ```
  Il faut, dans le programme principal, initialiser :
  ```
  L_sol=[]
  sol=[]
  ```
  et penser à ajouter un paramètre à la fonction `placer(p,...,sol,L_sol)`

Exercice 1 : Quadruplets d'entiers naturels de {0,1,2,3} satisfaisant les contraintes : x0!=x1, x2!=x3 et x0+x2<x1

Les 4 entiers prennent leur valeur dans l'ensemble {0,1,2,3}.

1. Solution itérative: Écrire un programme qui compte et affiche tous les quadruplets solutions. On obtient 30 solutions . (On utilisera pour cette question de simples boucles imbriquées.)

2. Backtracking : Écrire un programme qui compte et affiche les quadruplets solutions. On écrira les 4 versions suivantes :

- **V1** : on affiche les solutions au fur et à mesure : commencer par une version basique de la fonction `ajout_possible` qui renvoie True systématiquement puis l'améliorer . On vérifiera en affichant le nombre d'appels récursifs selon la version d'ajout_possible. (constater qu'on passe de 341 à 67, 57 ou même 50 (record à battre !) selon la qualité de la fonction `ajout_possible`).
- **V2** : on compte le nombre de solutions sans les afficher (utiliser une variable globale)
- **V3** : on s'arrête dés qu'on a trouvé et affiché une première solution .
- **V4** : on place les solutions dans une liste L_sol au fur et à mesure de leur détermination.

Exercice 2 : N-uplets d'entiers naturels non nuls croissants de somme constante

Pour `N>0`, on s'intéresse aux N_uplets **croissants et sans répétition** d'entiers naturels non nuls dont la somme vaut une valeur donnée `S`.

Par exemple, pour `N=3 et S=9`, les solutions sont : `(1,2,6),(1,3,5),(2,3,4)` . Mais `(1,1,7),(2,2,5),(3,3,3),(4,4,1)` ne sont pas solutions.

3. <u>Cas particulier `N=5` : solution itérative:</u> Écrire un programme qui compte et affiche tous les quintuplets strictement croissants de somme `S`.

   Par exemple, pour `S=18`, on obtient les 3 solutions : `(1,2,3,4,8)` , `(1,2,3,5,7)`, `(1,2,4,5,6)` . (On utilisera pour cette question de simples boucles imbriquées.)

4. <u>Backtracking</u> : Écrire les fonctions suivantes
   - `est_solution(N,S,t)` : renvoie un booléen vrai si et seulement si `t` est un N_uplet solution du problème (vérification de la croissance de `t` et de l'égalité `Σt[i]=S`)
   - `ajout_possible(p,N,S,t)` : renvoie un booléen vrai si et seulement si pour `0<p<N` le remplissage de la case `t[p]` est valide : `t[p-1]<t[p]` et `t[0]+...+t[p]<=S`
   - `placer(p,N,S,t)` : fonction de backtracking permettant l'affichage de tous les N_uplets croissants sans répétition dont la somme vaut `S`.
   - `nuc(N,S)` fonction finale qui appelle simplement la fonction de backtracking `placer(p,N,S,t)` et affiche les N-uplets solutions.

Exercice 3 : Sudokus

Lors de la résolution, penser à ne pas modifier les cases dévoilées au départ !

On peut:

- pour une case de numéro r (compris entre 0 et 80), exprimer la ligne par `r//9` et la colonne par : `r%9`
- pour une case donnée `[lig][col]`, exprimer le numéro de sa région: `3*(lig//3)+(col//3)`

- pour une case donnée `[lig][col]`, exprimer le numéro (compris entre 0 et 80) de la case correspondante: `9*lig + col`

- pour une région donnée (0,1,2,3,4,5,6,7,8), exprimer les coordonnées de la case située en haut et à gauche de cette région:
  `ligne=3*(region//3)` et `col=3*(region \%3)`

Exemple de grille de sudoku :

```
0, 8, 7, 0, 0, 0, 5, 2, 0
9, 1, 0, 5, 0, 2, 0, 4, 6
2, 0, 0, 0, 0, 0, 0, 0, 7

0, 9, 0, 0, 2, 0, 0, 1, 0
0, 0, 0, 1, 0, 6, 0, 0, 0
0, 4, 0, 0, 9, 0, 0, 8, 0

6, 0, 0, 0, 0, 0, 0, 0, 3
5, 7, 0, 3, 0, 1, 0, 6, 8
0, 3, 8, 0, 0, 0, 9, 5, 0
```

1. La fonction `est_solution` est inutile ici , pourquoi ?

2. Pour l'écriture de la fonction `ajout_possible :` on peut placer dans 3 listes différentes : les éléments de la ligne de la case traitée, ceux de la colonne de la case traitée, et ceux de la région de la case traitée et vérifier que ces 3 listes ont toutes bien des éléments différents .