On travaillera depuis la machine virtuelle ClefAgreg2019. On commencera par créer deux fichiers C `TP21_ex1.c` et `TP21_ex2.c`, important les bibliothèques nécessaires (vous devriez commencer à les connaître !).

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal).

```
$ COMPILATEUR -O3 -Wall -Werror -Wextra -fsanitize=address \
  -fsanitize=undefined -pedantic -std=c11 -o TP21_exX.exe TP21_exX.c
$ ./TP21_exX.exe
```

Ex.1 Résolution par "backtracking" du problème du Sudoku

Comme en cours lundi 25/04/2022, on s'intéresse au problème du Sudoku de taille 9×9.

La recherche par retour sur trace se prête très bien à la résolution de problèmes comme le Sudoku. On va ici tout simplement tenter de remplir chaque case du haut vers le bas (et de la gauche vers la droite) tant qu'on satisfait les contraintes du Sudoku.

Commençons par rappeler le principe du Sudoku :

— On part d'une grille de $9^2 = 81$ cases réparties en une grille de 3×3 sous-grilles de 3×3 cases, et comportant des chiffres de 1 à 9 dans certaines cases. Voici un exemple :

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

— L'objectif est de remplir chaque case avec un chiffre de 1 à 9 de sorte que chaque ligne, chaque colonne et chaque sous-grille 3×3 comporte une et une seule fois chaque chiffre.

— On suppose qu'une grille de Sudoku sera toujours initialement remplie de telle sorte qu'elle admette une et une seule solution.

1. Quel type proposez vous (en C) pour représenter une grille de Sudoku ? Précisez la convention de codage que cette représentation suppose.

On choisira par la suite de représenter une grille comme une matrice (ie. un tableau de tableaux, tous de la même taille), de taille 9×9, remplis d'entiers : un `int grille[9][9]`. Une case est soit vide et contient alors un entier 0, soit remplie d'une valeur $v \in \{1, \dots, 9\}$.

2. Dans votre fonction `main`, écrire une constante `probleme1` qui représente le problème de la figure exemple ci-dessus, avec le type proposé. Indenter bien pour que cela soit lisible.

Afin de définir la fonction de résolution, on définit une première fonction `suivant` de signature `void suivant(int grille[9][9], int i, int j, int* x, int* y)` (on ne peut pas facilement renvoyer un couple (x, y) alors on passe des pointeurs et la fonction écrira le résultat dans ces pointeurs). L'appel à `suivant(g, i, j, &x, &y)` ne renvoie rien, mais modifie les valeurs des entiers `x` et `y`, de telle sorte qu'ils deviennent les coordonnées de la prochaine case libre, dans l'ordre gauche à droite puis haut vers bas, après `(i,j)`, ou `(-1,-1)` quand il n'existe pas de telle case libre. Cela signifie alors que la grille est entièrement remplie.

3. Écrire cette fonction `suivant`.
4. Dans votre `main`, faire quelques tests de `suivant`, avec `probleme1` et des `assert`.
5. À partir de `suivant`, en déduire une fonction `bool estRemplie(int grille[9][9])` qui vérifie si la grille de Sudoku est entièrement remplie. On ne cherchera pas à vérifier qu'elle soit correctement remplie, puisque dans la suite on supposera partir d'une grille correcte, et chaque modification effectuée ne sera conservée que si elle n'invalide pas la grille.

On définit également une fonction `valide` de signature `bool valide(int grille[9][9], int i, int j)` telle que `valide(g, i, j)` renvoie `true` si et seulement si la valeur `vij = g[i][j]` placée en coordonnées `(i, j)` n'invalide pas la grille (cette valeur est entre 1 et 9). Ne pas prendre cette valeur en paramètre permettant d'écrire un peu plus simplement cette fonction.

La fonction est assez directe, étant donné (i, j), on va parcourir sa ligne, sa colonne et sa sous-grille (carré 3×3) pour vérifier qu'aucun autre nombre n'est égal à cette valeur `vij`.

6. Écrire cette fonction `valide`.
7. Dans votre `main`, faire quelques tests de `valide`, avec `probleme1` et des `assert`.

On peut alors définir la fonction `resout`, de signature `void resout(int g[9][9])`, qui va résoudre cette grille `g` de Sudoku, en effectuant tous les remplissages tant qu'on a une grille valide. Dès qu'une solution est trouvée, on s'arrête. Pour cela, on a juste à faire un `return`, pas comme en OCaml où il fallait utiliser une exception pour permettre une sortie prématurée. On a fait le choix de travailler en place dans la grille, ainsi à la fin de l'exécution de la fonction, la grille correspond à la solution.

8. Écrire cette fonction `void resout(int g[9][9])`. On commencera par écrire une fonction auxiliaire récursive, `void aux(int g[9][9], int i, int j)`
9. L'utiliser dans votre `main` pour résoudre la grille `probleme1`.
10. Écrire une fonction `void afficheGrille(int g[9][9])` qui affiche dans la console une grille de Sudoku. Attention : afficher une espace `' '` pour une case vide, et pas un `0`.
11. Conclure en affichant la grille initiale `probleme1` puis sa solution. Vérifier rapidement que la solution obtenue est bien correcte.
12. (bonus) si vous êtes curieux, vous pouvez ajouter dans les fonctions `resout` et `aux` un appel à `afficheGrille` au début de la fonction, pour visualiser la résolution au fur et à mesure. Vous pouvez aussi modifier les fonctions `aux` et `resout` pour calculer le nombre d'appels récursifs à `aux`, et l'afficher à la fin.

Ex.2 Résolution de problèmes sur un échiquier : 8 reines, et d'autres problèmes similaires

Un autre exemple classique de problème que l'on peut résoudre par backtracking est celui des huit reines : étant donné un échiquier classique (8×8), peut-on placer huit reines de sorte qu'aucune reine ne puisse attaquer une autre reine ? Plus précisément : sur un plateau de 8×8 cases, peut-on placer huit pièces tels que deux pièces quelconques ne soient jamais sur la même ligne, la même colonne, ou la même diagonale ?

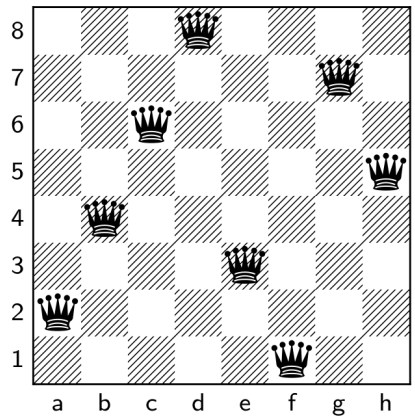

Figure 1 – Une solution maximale au problème des reines sur un échiquier classique 8×8.

Ce problème admet des solutions partielles en ne considérant que k reines à placer. Pour énumérer toutes les solutions, on peut même se contenter de solutions partielles où les k reines sont placées sur les k premières lignes.

Voici ainsi un algorithme pour énumérer les solutions :

— Supposons que k reines aient été placées et qu'on dispose d'une solution partielle.
  — Si $k = 8$ alors toutes les reines sont placées et la solution est complète, on la comptabilise (on l'ajoute à la liste des solutions déjà trouvées) ;
  — Sinon, on continue la recherche pour chaque position de la $k+1$ ème reine, sur la $k+1$ ème rangée, qui préserve le fait d'être une solution partielle.

Ici, quand on dit qu'on continue la recherche, ce qu'on signifie, c'est qu'on effectue un appel récursif.

Pour programmer cette méthode, en OCaml on avait définit une fonction récursive de signature : `val resout_reines : (int * int) list -> (int * int) list list`. Un appel à `resout_reines part` va ainsi renvoyer la liste des solutions complètes, construites à partir de la solution partielle `part`. Les solutions sont représentées par des listes de couples de coordonnées sur l'échiquier, donc dans $[\![0; 7]\!]^2$.

Cependant, en C, on ne dispose malheureusement pas d'un type primitif simple pour faire des listes, ou des listes de listes... On va donc devoir l'écrire nous-même.

Structure de liste chaînée

Couples d'entiers On définit la structure suivante qui permet de représenter un couple de coordonnées `(i, j)` entières (un `int * int` en OCaml), et un constructeur `couple(i,j)` qui

renvoie une structure représentant le couple `(i,j)`.

```
struct intint {
    int i;
    int j;
};
typedef struct intint intint;

intint couple(int i, int j) {
    intint ij;
    ij.i = i;
    ij.j = j;
    return ij;
    // return (intint) { .i = i, .j = j };
    // fonctionne aussi
}
```

Listes simplement chaînées de couples On définit ensuite la structure suivante qui permet de définir une liste simplement chaînée de `intint` (une `(int * int) list` en OCaml).

```
struct liste_intint {
    intint tete;
    struct liste_intint* queue;
};
typedef struct liste_intint liste_intint;
```

1. Écrire une fonction `liste_intint* liste_intint_vide()` qui renvoie une liste vide.
2. Écrire une fonction récursive `int longueur_liste_intint(liste_intint* liste)` qui calcule la longueur de la liste.
3. Écrire une fonction `liste_intint* empiler_intint(liste_intint* liste, int i, int j)` qui renvoie une liste contenant comme queue `liste` et comme nouvelle tête le couple `intint` obtenu par `(i, j)`. On utilisera bien-sûr un `malloc`.
4. Écrire une fonction récursive `void free_liste_intint(liste_intint* liste)` qui libère toute la mémoire allouée sur le tas pour cette liste.

Listes simplement chaînées de listes de couples On définit enfin la structure suivante qui permet de définir une liste simplement chaînée de listes de `intint` (une `((int * int) list) list` en OCaml).

```
struct liste_listes_intint {
    liste_intint* tete;
    struct liste_listes_intint* queue;
};
typedef struct liste_listes_intint liste_listes_intint;
```

6. Écrire une fonction `liste_listes_intint* liste_listes_intint_vide()` qui renvoie une liste vide.

7. Écrire une fonction `liste_listes_intint* liste_listes_intint_singleton(liste_intint* liste)` qui renvoie une liste contenant comme seule valeur la liste `liste`. On allouera un `liste_listes_intint*` sur le tas (`malloc...`) et on remplira ses deux champs `tete` et `queue`.
8. Écrire une fonction récursive `int longueur_liste_listes_intint(liste_listes_intint* listelistes)` qui calcule la longueur de la liste (sans s'intéresser aux longueurs des listes de couples qui la constitue).
9. Écrire une fonction récursive `void free_liste_listes_intint(liste_listes_intint* listelistes)` qui libère toute la mémoire allouée sur le tas pour cette liste (de listes, qu'il faut donc aussi libérer, avec des appels à `free_liste_intint`).
10. Écrire une fonction récursive `liste_listes_intint* concatener_liste_intint( liste_listes_intint* lls1, liste_listes_intint* lls2)` qui concatène deux listes. On utilisera aussi un `malloc`.

Structure générique de la résolution du problème pour un type de pièces donné (en OCaml)

Comme en cours lundi, la seule fonction préliminaire dont on a besoin est une fonction `valide_piece` (par exemple, `valide_reine` pour une reine) qui accepte comme argument `(x,y)` les coordonnées auxquelles on propose de rajouter une certaine pièce, et une liste `part` qui est une solution partielle.

Par exemple en OCaml pour les reines, on peut écrire :

```
let valide_reine (x,y) part =
  List.for_all (fun (i,j) ->
    i != x (* lignes différentes *)
    && j != y (* colonnes différentes *)
    && abs(i-x) != abs(j-y) (* diagonales différentes *)
  ) part
;;
```

En C, il faudra écrire une fonction récursive, `bool valide_reine(int x, int y, liste_intint* sol_partielle)`, qui renvoie `true` si la solution partielle (une `liste_intint* sol_partielle`) est vide, ou calcule si la reine en coordonnées `(i,j)` (en tête de la liste `sol_partielle`) ne peut pas attaquer la nouvelle reine en coordonnées `(x,y)`, et ainsi de suite avec la queue de la liste `sol_partielle`.

Une fois que l'on dispose d'une telle fonction, qui permet de vérifier si un placement d'une nouvelle pièce sur une case donnée n'est pas attaquée par aucune des pièces de la solution partielle trouvée jusque là, on peut écrire une fonction récursive `resout_pieces` (par exemple `resout_reines`), qui fonctionne comme ça :

```
let rec resout_pieces part =
  let k = List.length part in
  let resultats = ref [] in
  for x = 0 to 7 do
    let essai = (x,k) :: part in
```



```
    if valide_piece (x,k) part then
      resultats := (resout_pieces essai) @ !resultats;
  done;
  if !resultats != [] then !resultats
  else [ part ]
```

Pour écrire cela en C, il faut utiliser les fonctions sur les `liste_intint` et les `liste_listes_intint` que vous avez écrit aux parties précédentes.

Une fois que l'on obtient la liste de solutions, il faut pouvoir en faire au moins trois choses :

— en afficher (au moins) une, avec une fonction récursive `void print_liste_intint(liste_intint* liste)`,
— afficher le nombre de solutions, avec `longueur_liste_listes_intint(solutions_reines)` par exemple,
— afficher le nombre maximum de pièces dans les solutions trouvées, avec une fonction récursive `int max_longueur_liste_intint(liste_listes_intint* liste_listes)`,
— et après avoir fait cela, on n'oubliera pas d'appeler `free_liste_listes_intint(solutions_reines)` pour libérer la mémoire utilisée pour cette liste de listes de couples d'entiers.

Avec des tours (ligne et colonne)

Avant de résoudre le problème pour les reines, on va commencer par s'intéresser à des tours, qui ont moins de mouvements possibles (et donc il est peut-être possible d'en placer plus sur un échiquier). Une tour peut se déplacer sur sa ligne (gauche ou droite) ou sur sa colonne (haut ou bas).

— Écrire une fonction `bool valide_tour(int x, int y, liste_intint* sol_partielle)` qui vérifie si une tour en coordonnées `(x,y)` ne sera pas attaquée par aucune des tours présentes dans la liste de coordonnées `sol_partielle`.
— Combien trouvez-vous de solutions maximales différentes ?
On devrait en trouver $8! = 40320$, expliquer pourquoi.
— Quel est le nombre maximum de tours que l'on peut placer sur un échiquier, sans qu'aucune ne puisse attaquer aucune autre ? (8)

Avec des reines (ligne, colonne et diagonales)

Les mouvements possibles d'une reine sont ceux d'une tour ou d'un fou : sur sa ligne, sur sa colonne, et sur ses diagonales.

— Combien trouvez-vous de solutions maximales différentes ? (on devrait en trouver 92)
— Quel est le nombre maximum de reines que l'on peut placer sur un échiquier, sans qu'aucune ne puisse attaquer aucune autre ? (8)

Plus difficile : avec des fous (diagonales), des rois (cases adjacentes) ou des cavaliers

De la même manière, les mouvements possibles d'un fou sont sur ses diagonales, ceux d'un roi sont sur ses $\leqslant 8$ cases adjacentes, et ceux d'un cavalier sont de deux cases dans une direction puis un quart de tour à +90° ou -90° et une case dans la direction.

On pourrait essayer de faire de même pour énumérer les solutions du problème de placement de ces pièces, mais pour ces trois problèmes il y a beaucoup trop de solutions pour que notre approche (assez naïve) fonctionne. A titre d'exemple, j'ai trouvé un (très long) article de recherche qui montrait qu'il y a 22 522 960 placements de 8 fous sur un échiquier 8×8...

