TD 1

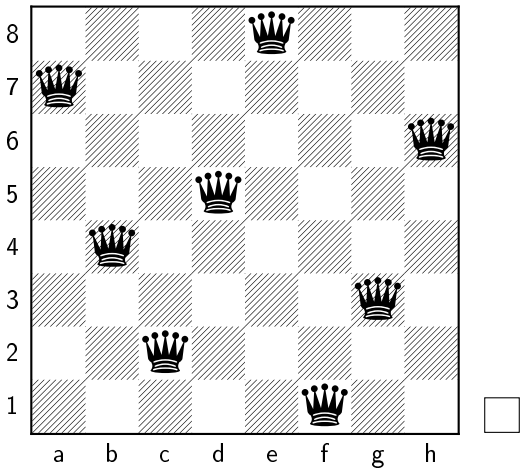
# Backtracking

Version avec preuve.

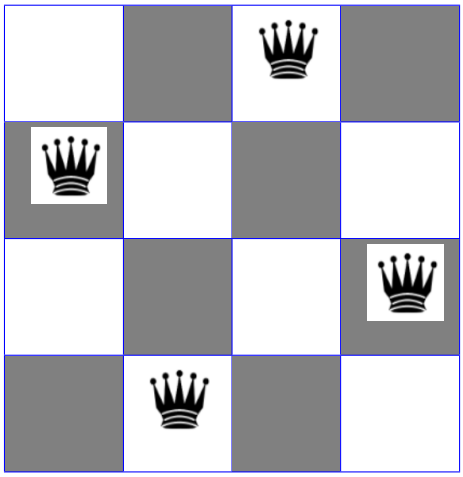
## 1 Le problème des $n$ dames

Le problème des $n$ dames consiste à placer $n$ dames sur un échiquier de taille $n \times n$ de telle sorte qu'aucune ne soit en prise : il ne faut donc pas plus d'une reine par ligne, par colonne et par diagonale.

**Exemple avec un échiquier traditionnel :**




1. Une petite découverte musical de la méthode pour $n = 4$ : `https://youtu.be/R8bM6pxlrLY`
2. Proposer une solution pour $n = 4$. Correction



3. Ecrire la fonction `possible : int -> int -> int list -> bool` qui prend en argument deux entiers représentant les coordonnées d'une case de l'échiquier, une liste d'entiers représentant la position des reines, et renvoie le booléen correspondant à la liberté de la case.

   Correction

```
let possible x y liste =
  let rec aux ligne colonne rang lis  = match lis with
    | [] -> true
    | dame :: reste ->  (colonne != dame) && ( ligne - rang != abs(
colonne - dame )) &&
        (aux ligne colonne (rang-1) reste) in
    aux x y (x-1) liste;;
```

4. En utilisant la méthode du backtracking, écrire la fonction `recherche_sol : int -> int -> int list -> int list` qui prend en argument le nombre de lignes, la ligne courante et la liste des dames déjà placées et renvoie la liste des dames, solution du problème. L'idée est de construire cette fonction de façon récursive, dont l'arrêt se fait si la ligne courante dépasse le nombre de lignes. La fonction fera appel à une fonction auxiliaire récursive qui aura pour but de remplir la liste des reines.

Correction

```
let rec recherche_sol nb_ligne x dames =
  if x > nb_ligne then dames
  else
    begin
      let rec suivant x y liste = match liste with
        | [] -> recherche_sol nb_ligne (x+1) [y]
        | d::reste ->
          begin
            if (y <= nb_ligne) &&(possible x y liste) then recherche_sol
nb_ligne (x+1) (y::liste)
            else
               if (y < nb_ligne) then suivant x (y+1) liste else
                 suivant (x-1) (d+1) reste

          end
      in suivant x 1 dames
    end;;
```

## 2 Satisfiabilité

Pour éviter de faire la table de vérité d'une proposition logique, on peut utiliser la méthode du backtracking pour résoudre le problème SAT d'une proposition.

1. On considère, sous sa forme normale conjonctive, la proposition logique suivante :

$$F(a,b,c,d) = (b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg c \vee \neg d) \wedge (\neg a \vee \neg c \vee d) \wedge (a \vee d)$$

Pour simplifier la notation, nous choisirons de noter la formule de la façon suivante :

$$F(a,b,c,d) = bc, \overline{a}\overline{b}c, \overline{c}\overline{d}, \overline{a}\overline{c}d, ad$$

(a) On suppose que $a$ prend la valeur vrai ( $\top$), déterminer la forme normale conjonctive de $F(\top, b, c, d)$.

Correction

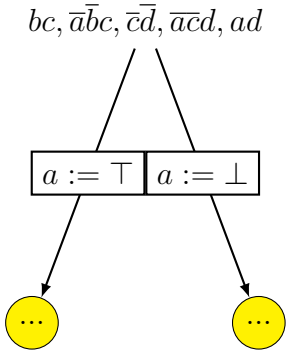
Si $a := \top$, la clause contenant $a$ est donc vraie, et donc disparait. Et celle contenant $\overline{a}$ ne dépend que du reste.
On trouve donc : $F(\top, b, c, d) = bc, \overline{b}c, \overline{c}\overline{d}, \overline{c}d$

(b) On suppose que $a$ prend la valeur faux ( $\bot$), déterminer la forme normale conjonctive de $F(\bot, b, c, d)$.

Correction

Si $a := \bot$, la clause contenant $\overline{a}$ est donc vraie, et donc disparait. Et celle contenant $a$ ne dépend que du reste.
On trouve donc : $F(\bot, b, c, d) = bc, \overline{c}\overline{d}, d$

(c) Compléter l'arbre suivant :




2. Montrer, en utilisant l'arbre, l'insatisfiabilité de la formule suivante :

$$F_2(a,b,c) = abc, a\overline{b}, b\overline{c}, \overline{a}\overline{b}\overline{c}, \overline{a}c$$

3. Pour résoudre le problème en Ocaml, on définira une formule à l'aide de listes de listes.Une clause est donc représentée par une liste avec autant d'éléments que de variable. Pour chaque littéral positif, on associera la valeur 1, et -1 pour un littéral négatif. Si la variable n'apparait pas dans la clause, on associera la valeur 0.
Exemple : Pour la formule $F$, on aura alors :

```
let exemple_F
=[[0;1;1;0];[-1;-1;1;0];[0;0;-1;-1];[-1;0;-1;1];[1;0;0;1]];;
```

(a) Écrire la fonction `positive : int list list -> int list list` qui prend en argument une proposition et renvoie la proposition pour laquelle la première variable prend la valeur vraie. Exemple :

```
positive exemple_F;;
- : int list list = [[1; 1; 0]; [-1; 1; 0]; [0; -1; -1]; [0; -1; 1]]
```

Correction

```
let rec positive exp = match exp with
  | [] -> []
  | liste::reste ->
    begin
      match liste with
        | [] -> positive reste
        | 0::reste_liste -> reste_liste ::(positive reste)
        | 1::reste_liste -> positive reste
        | (-1)::reste_liste -> reste_liste ::(positive reste)
    end;;
```

(b)

(c) Écrire la fonction `negative : int list list -> int list list` qui prend en argument une proposition et renvoie la proposition pour laquelle la première variable prend la valeur faux. Exemple :

```
negative exemple_F;;
- : int list list = [[1; 1; 0]; [0; -1; -1]; [0; 0; 1]]
```

Correction

```
let rec positive exp = match exp with
  | [] -> []
  | liste::reste ->
    begin
      match liste with
        | [] -> positive reste
        | 0::reste_liste -> reste_liste ::(positive reste)
        | 1::reste_liste -> positive reste
        | (-1)::reste_liste -> reste_liste ::(positive reste)
    end;;
```

(d) Écrire la fonction `solution : int list list -> bool` qui prend une liste de listes, représentant un état terminal dans l'arbre, et renvoie le booléen.

- Soit la dernière listes est vide, donc toutes les clauses ont été éliminées. La proposition est donc vraie.
- Soit la liste contient les clauses constituées d'un seul littéral. Il faut alors qu'il soit toujours du même signe. ( si les clauses contiennent que des zero, alors la proposition est fausse...)

Correction

```
(*  on suppose ici que l'expression ne possede qu'une variable   *)
let  solution exp =
  let rec aux exp =
    match exp with
      | [] -> (true , 0)
      |[]::[] -> (false,0)
```

```
      |[0]::[] -> (false , 0)  (* tout a disparu, mais s'il reste la
variable, c'est que c'est faux *)
      | [0]::reste -> aux reste
      | [1] :: reste -> let (bool,valeur ) = aux reste in
          if (valeur = 1 || valeur = 0  )&& bool then (true , 1)
          else (false , 0 )
      | [-1] :: reste -> let (bool,valeur ) = aux reste in
          if (valeur = -1 || valeur = 0  )&& bool then (true , -1)
          else (false , 0 )
      in
      let (b,i) =aux exp in b;;
```

(e) En déduire la fonction `recherche_sat : int list list -> bool` qui utilise la méthode du backtracking pour résoudre le problème SAT.

Correction

```
let rec recherche_sat exp =
  if exp = []  then true else
  if List.length (List.hd exp ) = 1 then  solution exp
    else
      begin
        recherche_sat (positive exp) || recherche_sat (negative exp)
      end;;
```

# 3 Résolution d'un sudoku

On considère la grille de sudoku suivante :

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

On utilisera les matrices, la valeur 0 remplaçant le case vide :

```
let grille_exemple =
[|[| 2;5;0;0;3;0;9;0;1|];
[| 0;1;0;0;0;4;0;0;0|];
[| 4;0;7;0;0;0;2;0;8|];
[| 0;0;5;2;0;0;0;0;0|];
[| 0;0;0;0;9;8;1;0;0|];
[| 0;4;0;0;0;3;0;0;0|];
[| 0;0;0;3;6;0;0;7;2|];
[| 0;7;0;0;0;0;0;0;3|];
[| 9;0;3;0;0;0;6;0;4|]|];;
```

1. Ecrire les fonctions suivantes, utiles à la résolution :

   - Une fonction d'affichage : `affiche : int array array -> unit` Correction

```
let affiche grille =
  for i = 0 to 8 do
    for j = 0 to 8 do
      print_int (grille.(i).(j) );
      print_string(";") ;
    done;
    print_newline () ;
  done;;
```

- Une fonction vérifiant qu'une valeur est absente d'une ligne donnée :
  `absent_ligne : 'a -> int -> 'a array array -> bool`
  Correction

```
let absent_ligne k ligne grille =
  (* la fonction renvoie vraie si k est absent sur la ligne *)
  let b = ref true in
  for colonne = 0 to 8 do
    if grille.(ligne).(colonne) = k then b := false
  done;
  !b;;
```

- Une fonction vérifiant qu'une valeur est absente d'une colonne donnée :
  `absent_colonne : 'a -> int -> 'a array array -> bool`
  Correction

```
let absent_colonne k colonne grille =
(* la fonction renvoie vraie si k est absent sur la colonne *)
  let b = ref true in
  for ligne = 0 to 8 do
    if grille.(ligne).(colonne) = k then b := false
  done;
  !b;;
```

- Une fonction vérifiant qu'une valeur est absente d'un bloc donné par la case étudiée :
  `absent_block : 'a -> int * int -> 'a array array -> bool` Correction

```
let absent_block k (i,j) grille =
(* la fonction renvoie vraie si k est absent sur le block contenant
le point (i,j) *)
  let b = ref true in
  let l = i - (i mod 3) in
  let c = j - (j mod 3 ) in
  for ligne = l to l + 2 do
    for colonne = c to c+2 do
    if grille.(ligne).(colonne) = k then b := false
    done ;
  done;
  !b;;
```

2. En utilisant la méthode du backtracking, écrire la fonction
   `est_valide : int -> int array array -> bool` qui renvoie un booléen indiquant si la grille peut être complétée. L'idée est de travailler récursivement sur la position courante ( entre 0 et 80 ).Lors de l'inspection de la case à la position $p$ deux cas peuvent se produire :

   - la case est occupée, dans ce cas, l'algorithme passe à la case $p+1$.
   - La case est libre. Dans ce cas, il cherche une valeur $v$ qui n'est pas sur la ligne, la colonne ni le bloc de la case $p$. L'algorithme tente de résoudre le sudoku en mettant $v$ dans la case $p$ puis passe à la case $p+1$ (il fait donc un pas en avant). Si la valeur $v$ ne permet pas de remplir la grille, l'algorithme libère la case $p$ (il fait donc un pas en arrière) et tente une autre valeur si il en existe une. Si aucune des valeurs proposée pour remplir la case $p$ ne conduit à une solution, l'algorithme renvoie faux.
     Correction

```
let rec est_valide  position grille =

  if position = 81 then true
  else
      let i = position / 9 in
      let j = position mod 9 in
      if grille.(i).(j) > 0 then( est_valide  (position+1) grille)
      else

        let rec suivant k grille_2 =
          if k = 10 then (grille.(i).(j) <- 0 ;false)
          else
            if (absent_ligne k i grille_2) && ( absent_colonne k j
grille_2) && (absent_block k (i,j) grille_2) then
              begin
                grille_2.(i).(j) <- k ;
                if  est_valide  (position+1) grille_2 then true
                else
                   (suivant  (k+1) grille_2)
              end else ( suivant (k+1) grille_2 )
        in suivant 1 grille

    ;;
```