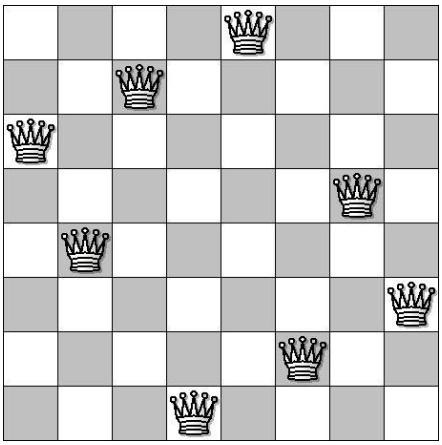
TP : Backtracking

1 Le problème des dames : résolution par backtracking

On rappelle qu'au jeu d'échecs, on travaille sur un échiquier de taille 8×8. Deux dames posées sur l'échiquier sont « en prise » si elles se trouvent sur la même colonne, la même ligne, ou la même diagonale (au sens large). Le problème des 8 dames consiste à placer 8 dames sur un échiquier de sorte que deux d'entre elles ne soient jamais en prise. La figure qui suit est un exemple de solution.



Ce problème se généralise naturellement au placement de n dames sur un échiquier $n \times n$. Les questions qui suivent donnent une stratégie pour calculer (et afficher le nombre de solutions) au problème, pour des n raisonnables (la complexité est exponentielle, mais c'est un peu la faute de ce qu'on cherche à calculer !).

On représente une grille avec des dames posées dessus comme une matrice (tableau de tableaux) `t` de booléens : lignes et colonnes sont numérotées de haut en bas et de gauche à droite, et un `true` en `t.(i).(j)` indique qu'une dame est posée dans la case (i, j). On va essayer de calculer des solutions en suivant le principe du backtracking.

Question 1. *Fonctions utiles*. Écrire les fonctions suivantes :

- `case_suivante n i j` prenant en entrée l'entier n et les indices d'une case (i, j), et renvoyant la case suivante (celle située immédiatement à sa droite si elle existe, ou $(i + 1, 0)$ sinon. On ne traitera pas à part le cas $i = j = n - 1$: arriver en $(n, 0)$ signifiera que l'exploration est terminée.

```
# case_suivante 4 0 0 ;;
- : int * int = (0, 1)
# case_suivante 4 1 3 ;;
- : int * int = (2, 0)
# case_suivante 4 3 3 ;;
- : int * int = (4, 0)
```

- `imprime t` permettant d'imprimer une matrice de booléens à l'écran : on écrira `x` à la place de `true` et `o` à la place de `false`. Par exemple :

```
# imprime [|[|false; false; true; false|]; [|true; false; false; false|];
[|false; false; false; true|]; [|false; true; false; false |]|] ;;

ooxo
xooo
ooox
oxoo

- : unit = ()
```

  On fera usage de `print_string : string -> unit` affichant à l'écran une chaîne de caractères et `print_newline : unit -> unit` revenant à la ligne.

La technique de résolution va consister à parcourir la matrice de haut en bas puis de gauche à droite : en chaque case, on teste si on peut poser une dame en cette case, sans violer les règles vis à vis des dames déja placées (elles sont situées au dessus ou à gauche).

Question 2. *Test d'une case*. Écrire une fonction `en_prise t i j` prenant en entrée une matrice de booléens représentant un échiquier $n \times n$ (n est accessible par `Array.length t`) et testant si la case (i, j) est en prise : ceci signifie qu'il existe un `true` sur une case située à gauche, au dessus, au dessus à gauche en diagonale, ou au dessus à droite en diagonale (on ne considérera pas les cases situées à droite ou en dessous). Attention à ne pas faire de dépassement d'indice sur les cases !

```
# test_en_prise () ;;
- : bool * bool * bool * bool = (true, true, true, false)
```

On a maintenant tout le matériel pour écrire une fonction calculant toutes les solutions au problème des n dames. L'algorithme fera usage d'une fonction récursive interne `aux i j m` où :

— `(i,j)` est soit une case de la grille, soit $(n, 0)$;

— m est le nombre de dames déja placées dans les cases précédentes (situées à gauche ou au dessus).

Cette fonction, renvoyant un entier, effectue deux choses :

— afficher à l'écran toutes les solutions au problème des n dames, qui complètent les choix déja effectués sur la grille (avant la case (i, j) ;

— renvoyer le nombre de telles solutions : elle renvoie donc un entier.

Les traitements à effectuer sont :

— Cas de base : si $m < i$, cela signifie qu'on a placé moins de une dame par ligne située au dessus de la case (i, j), on ne peut donc compléter les choix précédents en une solution. Que faire dans ce cas ?

— Cas de base : si $i = n$, cela signifie qu'on a réussi à placer une dame par ligne de la grille. Que faire dans ce cas ?

— Sinon, on teste si la case (i, j) peut contenir une dame. Si ce n'est pas le cas, on fait directement un appel récursif sur la suivante. Si c'est le cas, il faut faire deux appels récursifs suivant si on met ou non une dame en case (i, j). Attention, lorsque le deuxième appel termine, il faudra que la case ne contienne plus de dame, puisqu'on reviendra en arrière via les appels récursifs, et que l'on utilise une seule matrice pour faire le travail.

L'appel initial sera `aux 0 0 0` : on fait en effet un appel sur la première case, et pour le moment on a placé aucune dame.

Question 3. *Solution par Backtracking*. Compléter la fonction suivante (disponible en annexe) en suivant les indications.

```
def calcul_sol n =
  let t=Array.make_matrix n n false in
  let rec aux i j m =
    if m<i then ...
    else if i=n then ...
    else (
      let i2,j2=case_suivante n i j in
      if en_prise t i j then ...
      else ...
    )
  in aux 0 0 0;;
```

Question 4. Tester !

```
# calcul_sol 4 ;;

oxoo
ooox
xooo
ooxo


ooxo
xooo
ooox
oxoo

- : int = 2
```

Vérifier que vous obtenez 92 solutions pour $n = 8$.

2 Le problème de la somme de sous-ensembles

Cette section est moins guidée ! On se donne un tableau d'entiers strictement positifs et une somme s à atteindre, et on se demande si s est la somme de certains éléments du tableau. On veut écrire une fonction qui affiche à l'écran une telle solution si elle existe, et renvoie dans tous les cas un booléen indiquant si une solution à été trouvée, par exemple :

```
# pb_somme [|54; 14; 2; 8; 21; 6; 17; 25; 144; 12; 20|] 123 ;;
54 14 2 8 25 20
- : bool = true
# pb_somme [|54; 14; 2; 8; 21; 6; 17; 25; 144; 12; 20|] 308 ;;
- : bool = false
# pb_somme [|54; 14; 2; 8; 21; 6; 17; 25; 144; 12; 20|] 15 ;;
- : bool = false
```

Question 5. Écrire une telle fonction `pb_somme t s` faisant usage de backtracking, qui aura a priori une complexité $O(2^n)$, où n est la taille du tableau `t`. Essentiellement, pour chaque indice i du tableau, on choisira ou non de considérer l'élément `t.(i)`, en procédant de gauche à droite au fur et à mesure que les appels récursifs s'imbriquent.

Question 6. On peut essayer d'améliorer un peu la vitesse d'exécution en prenant en compte les deux faits suivants :

- si la somme des éléments déja choisis dépasse la somme s à atteindre, on peut s'arrêter ;
- si la somme des éléments déja choisis, plus la somme de tous les éléments à partir de l'indice i courant ne suffit pas à atteindre s, on peut s'arrêter également.

Pour le deuxième test, on pourra précalculer les sommes cumulées de chaque indice jusqu'à la fin du tableau. Écrire la version `pb_somme_2` correspondante.

```
# pb_somme [|548; 147; 251; 848; 201; 651; 177; 256; 512; 121; 209; 247; 114; 875; 142; 851; 124; 311;
1130; 514; 419; 857; 120; 315; 74; 125|] 1000 ;; (* quelques secondes *)
548 251 201
- : bool = true
# pb_somme_2 [|548; 147; 251; 848; 201; 651; 177; 256; 512; 121; 209; 247; 114; 875; 142; 851; 124; 311;
1130; 514; 419; 857; 120; 315; 74; 125|] 1000 ;; (* instantané *)
548 251 201
- : bool = true
```