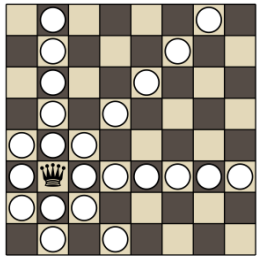
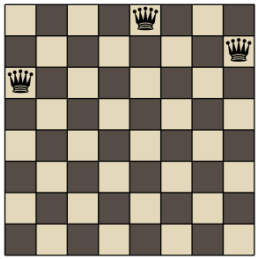
TP11. Le problème des 8 dames en C

Sur un échiquier la dame peut prendre une pièce se trouvant sur la même diagonale, la même ligne ou la même colonne qu'elle. Sur l'échiquier ci-dessous, la dame peut prendre toute pièce se trouvant sur une case marquée d'un rond blanc. On essaie dans cet exercice de positionner des dames sur l'échiquier de sorte qu'aucune ne puisse prendre une autre. En particulier on ne peut pas mettre plus d'une dame par ligne.


Prises de la reine


Solution partielle

On s'intéresse au problème de savoir s'il est possible de placer n dames sur un échiquier de dimension $n \times n$ (et comment). Les positions sur l'échiquier seront repérées comme les coefficients d'une matrice $n \times n$ (ligne puis colonne, de haut en bas et de gauche à droite).
On représente une solution (totale ou partielle) par un tableau de dimension n indiquant, dans sa i-ème case, la position choisie pour la dame se trouvant sur la i-ème ligne du plateau.
Sur l'exemple ci dessous, le tableau aurait pour taille `8` et pour 3 premières valeurs `4, 7` et `0`.

1. Écrire une fonction `void affichage(int* sol, int n)` qui affiche un échiquier de taille $n \times n$, en indiquant les cases vides par un `X` et les dames par un `#`.

On construit une solution ligne par ligne. Si on suppose construite une solution partielle qui définit où sont placées les dames sur $l < n$ premières lignes, il s'agit de de choisir une colonne c pour la dame à placer en ligne l.

2. Implémenter une fonction `bool est_extension_valide(int* sol, int nbl, int col)` qui teste, étant donné une solution partielle sol définie sur les `nbl` premières lignes, et une colonne `col`, s'il est possible de placer une dame en ligne `nbl` et colonne `col` sans invalider la solution partielle.

I. Implémentation par retour sur trace

On propose une méthode de résolution par retour sur trace. Étant donné une solution partielle valide construite sur les i premières lignes, deux cas peuvent se présenter :

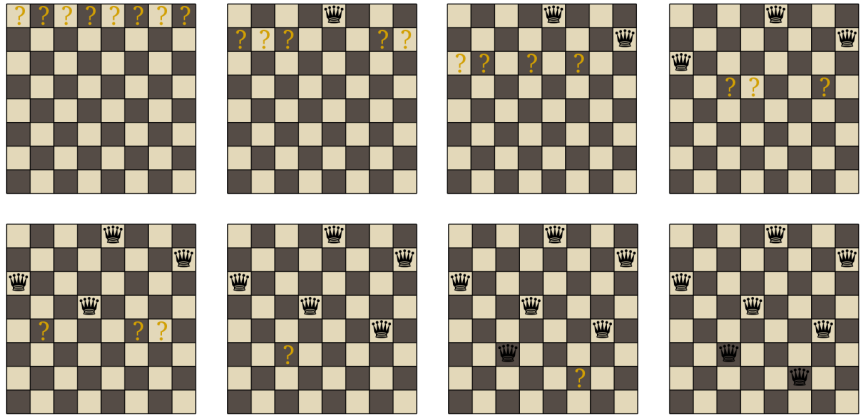
- Si $i = n$, alors il est possible de placer n dames.
- Sinon, il faut compléter la solution. On envisage successivement chaque colonne j valide. On essaie récursivement de compléter la solution obtenue avec cette nouvelle dame. Si un de ces appels récursifs répond positivement, on répond positivement, sinon on répond négativement.

3. Définir une fonction `bool peut_etre_completee(int* sol, int nbl, int n)` qui prend en argument une solution sol définie de manière valide pour les `nbl` premières lignes, et qui teste si celle-ci peut être complétée en une solution de taille `n` par l'algorithme récursif décrit ci-avant. De plus, dans le cas où la solution peut être complétée, le tableau `sol` est complété de manière à stocker, à l'issue de l'appel, une solution complète qui étend la solution partielle de départ.

4. Définir finalement une fonction `void resolution_deterministe(int n)` qui teste s'il existe une solution au problème des `n` dames et affiche le résultat. *On veillera à libérer l'espace mémoire alloué au cours de cette fonction.*

II. Version probabiliste

Au lieu d'explorer l'espace de tous les placements possibles de dames sur un échiquier, on définit un algorithme qui "tente" une génération d'un placement valide des dames. Comme précédemment, on construit la solution ligne à ligne. Au lieu d'explorer toutes les possibilités, on n'explore qu'une seule branche choisie aléatoirement. En effet, à chaque étape l'algorithme choisit où placer la dame sur la prochaine ligne de manière uniforme parmi les positions valides (i.e. compatibles avec les dames déjà placées). La figure ci-dessous donne un exemple d'exécution de cet algorithme de génération.



Les cases marquées d'un **?** représentent les cases parmi lesquelles l'algorithme choisit uniformément, à chaque étape le placement d'une nouvelle dame. À la dernière ligne, il n'est pas possible de placer une dame. compatible avec celles placées au préalable, la fonction de génération a échoué, et renvoie alors `false`. S'il avait réussi à fabriquer une solution complète, il aurait renvoyé `true`.

L'algorithme de résolution du problème consiste alors simplement à relancer l'algorithme de génération tant que celui-ci répond `false`. On remarque que si le problème des n-dames n'admet pas de solution pour la valeur de n avec laquelle l'algorithme est appelé, celui-ci ne s'arrête pas.

5. Définir une fonction `int positions_valides(int* sol, int nbl, int n, int* tab_pos)` qui prend en argument

- `n` la taille de l'échiquier considéré ;
- `nbl` un nombre de ligne inférieur à n ;
- une solution `sol` définissant des placements valides pour les `nbl` premières lignes
- `tab_pos` un tableau de `n` entiers, déjà alloué (peu importe comment il est rempli) ;

et qui enregistre dans le tableau `tab_pos` les indices de placement d'une dame sur la ligne `nbl` valides, et qui renvoie le nombre de telles positions.

6. Définir une fonction `bool generation_aleatoire(int n, int* sol)` qui prend en argument `n` la taille de l'échiquier considéré et `tab_sol` un tableau déjà alloué de taille `n` et qui tente de construire une solution valide de taille `n` par l'algorithme probabiliste précédemment décrit, et qui renvoie si cette tentative a réussi. De plus, dans l'affirmative, la solution construite est enregistrée dans le tableau `sol`.
On pourra se référer à l'encadré en fin de TP pour la génération de nombres aléatoires en C.

7. Définir finalement une fonction `void resolution_probabiliste(int n)` qui teste, avec l'algorithme probabiliste présenté, s'il existe une solution au problème des `n` dames et affiche le résultat.

Génération de nombres aléatoires en C

Dans la libraire `<stdlib.h>` est définie la fonction `int rand()` qui renvoie un nombre pseudo-aléatoire entre `0` et `RAND_MAX`.
Cette fonction est un générateur, à chaque appel au cours d'une exécution elle donne un nouvel entier. Cependant, utilisée telle quelle, cette fonction a le même comportement à chaque exécution.

Pour pallier ce problème, il faut initialiser cette suite à l'aide de la fonction `srand`, avec une nouvelle valeur à chaque exécution. On utilise généralement la fonction `time` de la librairie `time.h` qui renvoie le nombre de secondes écoulées depuis le 01/01/1970.

Ainsi, le programme suivant génère des nombres aléatoires :

```
int main(){
    srand(time(NULL));
    for (int i = 0; i < 10; i++){
        int n = rand() % 100; //nbr aléatoire entre 0 et 99
        printf("%d\n", n);
    }
}
```