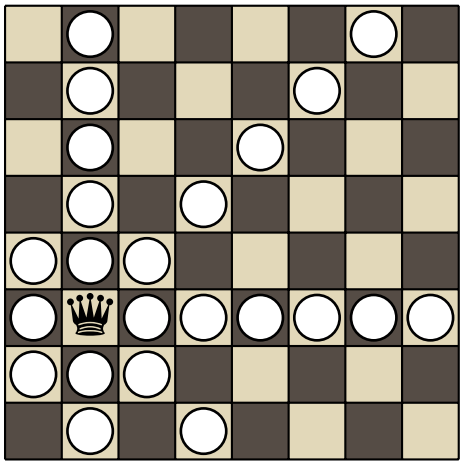
TP n°7 - Comparer un algorithme déterministe et un algorithme probabiliste de type Las Vegas

Notions abordées

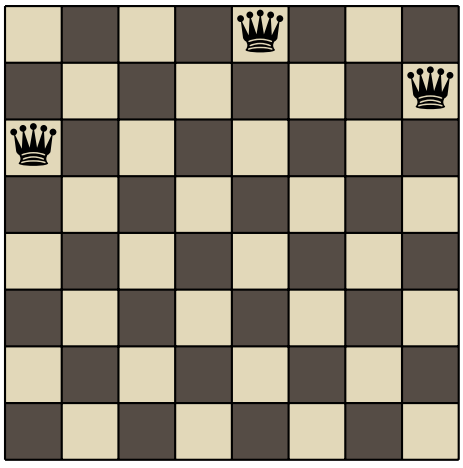
- recherche exhaustive
- retour sur trace
- algorithme probabiliste de type Las Vegas
- générer des nombres aléatoires en C

Exercice 1 : Problème des huits dames

Sur un échiquier la dame peut prendre une pièce se trouvant sur la même diagonale, la même ligne ou la même colonne qu'elle. Aussi, sur l'échiquier ci-dessous, la dame (représentée par une couronne) peut prendre toute pièce se trouvant dans une case marquée d'un rond blanc cerclé de noir. 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, donc sur un échiquier classique, c'est-à-dire de dimensions 8×8, il n'est pas possible de placer plus de 8 dames♣. Se pose donc la question de savoir s'il est possible de placer exactement 8 dames sur un échiquier, et si oui de comment les placer.



Prises de la reine



Solution partielle

Dans la suite nous allons en fait généraliser au problème de savoir s'il est possible de placer n dames sur un échiquier de dimension $n \times n$. 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). Dans cet exercice les implémentations sont à faire en C. On se propose de représenter une solution au problème des n dames au moyen d'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. Afin de compléter l'information fournie par

♣. par application du principe **des tiroirs/chaussettes**

le tableau, l'algorithme maintiendra un entier indiquant la ligne jusqu'à laquelle la solution partielle a été construite. Ainsi le contenu des cases du tableau après cette ligne n'est pas spécifié et ne sera jamais lu. Par exemple la solution partielle ci-dessus a été construite jusqu'à la 3-ième ligne et est représentée au moyen d'un tableau d'entiers de la forme : `{4, 7, 0, … }`, notons `sol_ex` cette solution partielle dans la suite.
À la fois dans l'algorithme déterministe et dans l'algorithme probabiliste que nous allons mettre en place ci-après, on construit une solution ligne par ligne. Si on suppose construite une solution partielle qui définit où sont placées les dames sur $i < n$ premières lignes♣, il s'agit de de choisir une colonne j pour la dame à placer en ligne i. Afin que la solution partielle ainsi obtenue puisse être étendue en une solution complète, on doit s'assurer qu'en plaçant une dame sur la case (i, j), on n'a pas invalidé la solution.
Afin de représenter les solutions (même partielles), on définit le type suivant.

```
typedef int* solution;
```

Ainsi, lorsqu'on s'intéresse au problème pour un échiquier de taille n, une solution est toujours un tableau d'entiers de n cases, et une solution partielle est un tel tableau dont seules les premières cases sont significatives. Les autres cases, celles correspondant aux lignes où l'on n'a pas encore placé de dame, peuvent être remplies par exemple♡ avec la valeur `-1`.

Q. 1 Bonus Définir une fonction qui affiche l'échiquier représenter par un objet de type `solution`. On pourra utiliser par exemple un `X` pour les cases vides, un # pour celles où il y a une dame. L'affichage de la solution `{1, 3, 0, 3}` pourrait être celui donné ci-contre.

```
0  X # X X
1  X X X #
2  # X X X
3  X X X #
```

Q. 2 Implémenter une fonction `bool est_extension_valide(solution 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.

Exemples :
`est_extension_valide(sol_ex, 3, 0)` vaut `false`
`est_extension_valide(sol_ex, 3, 1)` vaut `false`
`est_extension_valide(sol_ex, 3, 2)` vaut `true`

1. Version déterministe, par retour sur trace

On propose d'abord un algorithme déterministe permettant de répondre au problème par **retour sur trace**♠. L'idée est donc la suivante. Étant donné une solution partielle valide construite sur les i premières lignes, deux cas peuvent se présenter.

- Si $i = n$, alors oui il est possible de placer n dames,
- Sinon il faut compléter la solution. On envisage successivement chaque colonne j pour placer une nouvelle dame, si celle-ci définit une position de placement valide (eu égard aux dames déjà placées), alors 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.

Q. 3 Donner un majorant du nombre d'appels récursifs réalisés par cet algorithme pour résoudre le problème sur un échiquier de taille $n \times n$.

♣. c'est-à-dire de la ligne 0 à la ligne $i - 1$ comprise
♡. En pratique, si l'algorithme est bien codé, on ne consulte jamais la valeur contenue dans ces cases du tableau, mais mettre -1 permet de détecter une erreur.
♠. aussi appelé backtracking en anglais

Q. 4 Dessiner l'arbre des appels récursifs effectués par cet algorithme sur une **petite** instance, où petite s'entend connaissant la réponse à la question précédente.

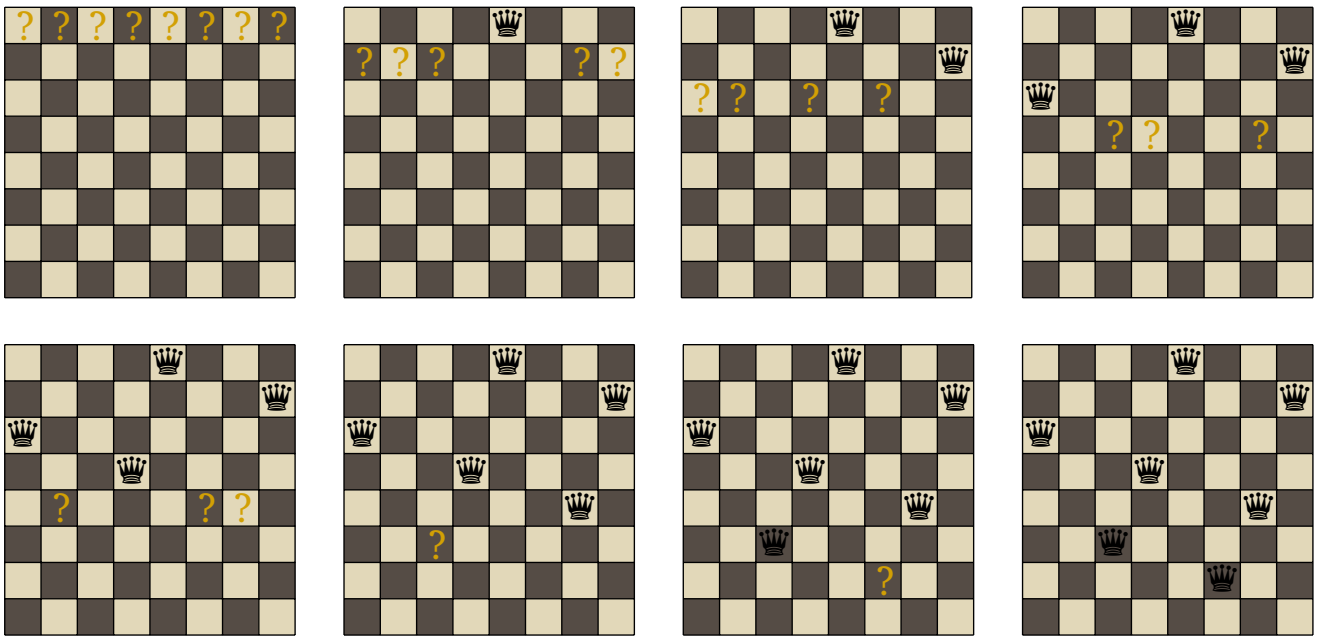
Q. 5 Définir une fonction `bool peut_etre_completee(solution sol, int nbl, int n)` qui prend en argument une solution `sol` définie de manière valide pour les `nbl` premières lignes, et qui teste si celle-ci peut être complétée en une solution de taille `n` par l'algorithme récursif décrit ci-avant. 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 étand la solution partielle de départ.

Q. 6 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. Si la question 1 a été traitée, cette fonction pourra afficher la solution dans le cas où la réponse est vrai, sinon on se contentera d'afficher d'une phrase disant s'il existe ou non un placement valide de n dames. *On veillera à libérer l'espace mémoire alloué au cours de cette fonction.*

On observe évidemment que cet algorithme, au comportement exponentiel, ne permet que difficilement de répondre au problème du placement possible des dames sur un échiquier 20×20 et pas du tout sur un échiquier de taille 50×50.

2. Version probabiliste

On implémente maintenant un algorithme de type Las Vegas permettant de trouver des solutions au problème du placement des dames. 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 l'algorithme déterministe, l'algorithme de génération construit la solution ligne à ligne, mais au lieu d'explorer toutes les possibilités, il 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 ? 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.

Q. 7 Définir une fonction `int positions_valides(solution 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 au regard des reines déjà positionnées sur les `nbl` premières lignes selon `sol`, et qui renvoie le nombre de telles positions.

Q. 8 Définir une fonction `bool generation_aleatoire(int n, solution tab_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 `tab_sol`. *Si besoin, on pourra se référer à l'encadré en fin de TP pour la génération de nombres aléatoires en C. On veillera à libérer l'espace mémoire alloué au cours de cette fonction.*

Q. 9 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. Si la question 1 a été traitée, cette fonction pourra afficher la solution trouvée, sinon on se contentera de l'affichage d'une phrase disant qu'il existe un placement valide de n dames. *On veillera à libérer l'espace mémoire alloué au cours de cette fonction.*

Q. 10 Comparer les temps d'exécution des deux algorithmes (probabiliste et déterministe). *On pourra d'abord comparer "à l'œil nu" la différence des temps d'exécution. Pour plus de précision, on utilisera la commande* `time` *. Pour faciliter les tests, le programme peut prendre en paramètre la taille de l'échiquier. Deux encadrés sur ces sujets sont donnés en fin de TP.*

Générer un nombre aléatoire en C

Pour générer un nombre pseudo-aléatoire en C, on utilise la fonction `rand`. Celle-ci est codée dans la librairie standard, qu'il faut donc inclure. Cette fonction est un générateur, à chaque appel au cours d'une exécution elle donne un nouvel entier, la suite des entiers ainsi obtenus étant une suite de nombres pseudo-aléatoires. Cependant, utilisée telle quelle, cette fonction a le même comportement à chaque exécution. En commentant la ligne 6 du programme ci-contre avant compilation, on peut observer qu'il s'exécute toujours de la même manière.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  int main(){
6    srand(time(NULL));
7    for(int i=0;i<=20;i++){
8      printf("%d\n", rand());
9    }
10   return 0;
11 }
```

Pour pallier ce problème, il faut initialiser cette suite avec une nouvelle valeur à chaque exécution. Pour cela, on utilise généralement la fonction `time` de la librairie `time.h` qui renvoie le nombre de secondes écoulées depuis le 01/01/1970 à 00h00mm00s.

Les entiers générés simulent une loi uniforme sur $[\![0, N]\!]$ avec $N =$ `RAND_MAX`, où `RAND_MAX` est une constante supérieure à $2^{15} - 1$ qu'on peut utiliser (et afficher par curiosité). Pour obtenir des entiers de $[\![0, p-1]\!]$, on pourra prendre les valeurs obtenues modulo p, et pour obtenir des entiers de $[\![a, b]\!]$, on pourra prendre les valeurs obtenues modulo $p = b - a + 1$ puis leur ajouter a, même si ces opérations ne préservent pas tout à fait l'uniformité du tirage.

Exercice Écrire un programme qui affiche la proportion d'entiers plus petits que `RAND_MAX`$/2$ pour 1000 tirages effectués avec la fonction `rand`.

Mesurer un temps d'exécution à partir du terminal

En Bash (la langage du terminal sous Debian notamment), la commande `time` permet de mesurer le temps d'execution d'une commande. Pour essayer, on peut par exemple l'utiliser avec la commande `sleep` qui prend en paramètre un entier `N` et qui a pour effet de laisser s'écouler `N` secondes avant d'être terminée.

```
user@machine:~/dossier$ time sleep 2

real  0m2,004s
user  0m0,001s
sys  0m0,003s
```

On peut simplifier un peu la sortie pour ne récupérer que le temps "real". Il suffit d'affecter la valeur `%R` à la variable bash `TIMEFORMAT`, comme dans l'exemple ci-dessous. Attention cette affectation [a] ne vaut que pour le terminal courant. Dans un autre terminal, ou après fermeture, il faut refaire l'affectation. En revanche si on garde le même terminal, il n'est pas utile de la refiare avant chaque appel à `time`.

```
user@machine:~/dossier$ TIMEFORMAT=%R
user@machine:~/dossier$ time sleep 2
2,004
user@machine:~/dossier$ time echo "bonjour"
bonjour
0,000
```

La commande `time` n'affiche le temps qu'à l'issue de l'exécution, de sorte qu'au cours d'une exécution qui paraît longue, cette commande ne permet pas de savoir depuis combien de temps elle dure. On pourra utiliser la commande `date` pour afficher l'heure au début et/ou à la fin d'une exécution, comme dans l'exemple suivant.

```
user@machine:~/dossier$ TIMEFORMAT=%R
user@machine:~/dossier$ echo "début à `date +"%Hh%Mm%S"`"; time sleep 3; echo
↪  "fin à `date +"%Hh%Mm%S"`"
début à 17h25m17
3,004
fin à 17h25m20
```

a. affectation à faire comme dans l'exemple, sans espace autour du =... Bash est un peu sensible

Créer un exécutable qui prend des arguments en ligne de commande en C

Il est possible de passer des paramètres à un programme C compilé `prgm` au moment où on lance son exécution en ligne de commande, à condition d'avoir déclaré dans le code source du programme la fonction `main` avec la signature suivante `int main(int argc, char* argv[]);`
Attention : contrairement à ce qu'on peut faire avec les arguments d'une fonction, les paramètres d'un programme sont tous des chaînes de caractères.
Au moment de l'exécution du programme, la fonction `main` est alors appelée avec des valeurs d'arguments qui dépendent de la ligne de commande

- `argc` a pour valeur le nombre de mots constituant la ligne de commande, y compris la commande elle-même, c'est donc le nombre de paramètres réellement souhaités plus 1
- `argv` est un tableau de `argc` chaînes de caractères initialisées selon les mots de la ligne de commande, en particulier `argv[0]` est le nom de la commande (donc rarement utile).

Exemples

• La ligne de commande `./prog a 12`, `argc` vaut 3 (la commande + les 2 paramètres), et `argv[0]` pointe vers la chaîne `"./prog"`, `argv[1]` vers `"a"` et `argv[2]` vers `"12"`. Pour récupérer la valeur d'un entier à partir d'une chaîne de caractères contenant son écriture décimale, on peut utiliser la fonction `atoi` de la librairie standard.

• Si `add` est le fichier obtenu en compilant le code ci-contre, alors `./add 12 24` affiche `12 + 24 = 36`, mais `./add 12 24 36` affiche `on attend 2 entiers` avant l'erreur d'assertion.

Exercice Créer en C un programme `echo` qui affiche la ligne de commande qui l'a lancé, hormis le premier mot `./echo`.

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  int nb_arg_attendu = 2;
  if(argc != nb_arg_attendu +1){
    printf("on attend 2 entiers\n");
    assert(false);
  }
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);

  printf("%d + %d = %d\n", a, b,
  ↪  a+b);

  return 0;
}
```