

# Problème des reines sur l'échiquier [hs04] - Exercices

Karine Zampieri, Stéphane Rivière

Unisciel  algoprogram  UNIVERSITÉ HAUTE-ALSACE Version 21 mai 2018

## Table des matières

<b>1</b>	<b>Le problème des reines</b>	<b>2</b>
<b>2</b>	<b>Algorithmique, Programmation</b>	<b>3</b>
2.1	Représentation du problème . . . . .	3
2.2	Résolution récursive . . . . .	4
2.3	Résolution à partir d'une position . . . . .	7
2.4	Blocage de l'échiquier des reines . . . . .	11
2.5	Heuristique des n reines . . . . .	11
2.6	Force brute des reines . . . . .	17
<b>3</b>	<b>Références générales</b>	<b>18</b>

## C++ - Problème des reines sur l'échiquier (Solution)



Mots-Clés Backtraking, Heuristique ■

Requis Axiomatique objet, Récursivité des actions, Heuristique ■

Difficulté ●●○ (4 h) ■



### Objectif

Cet exercice résout le problème des reines sur l'échiquier : résolution récursive, recherche particulière et heuristique des reines.



### Remarque

Dans le même ordre d'idées, l'exercice @[Problèmes du parcours du cavalier] résout celui du cavalier.

...(énoncé page suivante)...

# 1 Le problème des reines

Voici un autre casse-tête pour les mordus des échecs : il consiste à placer  $n$  reines sur un échiquier  $n \times n$  sans qu'elles ne se prennent l'une l'autre. Une reine peut prendre toutes les pièces se trouvant sur la même ligne, sur la même colonne ou sur les mêmes diagonales qu'elle-même.

## Exemple

Sur la figure de gauche, cinq reines sont placées mais la sixième est bloquée. Sur la partie droite, sept reines sont placées mais la huitième est bloquée.

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

## Historique

Ce problème fut posé pour la première fois en 1848 par MAX BESSEL dans un journal d'échecs. Cette publication donna lieu à un engouement extraordinaire. Le 21 septembre 1850, le Dr. NAUCK donna pour  $n = 8$  toutes les 92 solutions alors que le mathématicien GAUSS n'en trouva que 72. D'après KRAITCHICK, le nombre  $s$  de solutions en fonction de l'ordre  $n$  de l'échiquier est :

$n$	1	2	3	4	5	6	7	8	9	10	11	12
$s$	1	0	0	2	10	4	40	92	352	720	2680	14200

...(suite page suivante)...

## 2 Algorithmique, Programmation

### 2.1 Représentation du problème

#### Structures de données

Deux reines ne pouvant être sur la même ligne et puisqu'il faut placer  $n$  reines et qu'il y a  $n$  lignes, il y aura une reine par ligne. De même, il y a une reine par colonne. Ainsi si les reines sont placées sur les lignes 1 à  $n$ , il suffit d'un tableau unidimensionnel pour mémoriser l'état de l'échiquier d'où :

- « La reine  $k$  » est une abréviation de « la reine qui est sur la ligne  $k$  »
- On peut modéliser l'échiquier par un vecteur.



Définissez :

- Le type `Plateau` comme étant un `Vecteur` d'entiers.
- Le type `Solutions` comme une `Liste` de `Plateau`.

#### Outil C++

Le modèle de classes `vector<T>` est défini dans la bibliothèque `<vector>` et celui de `list<T>` dans la bibliothèque `<list>`.

#### Test de case libre

Pour savoir si une case est libre il faut regarder si la colonne et les diagonales qui y passent sont libres. Si  $R = r[k]_{1..n}$  est le vecteur qui indique les numéros des colonnes des reines :

- Alors  $r[k] = 0$  si la  $k$ -ème reine est libre.
- Sinon  $r[k]$  donne l'indice de la colonne où elle est posée.

Les diagonales sont caractérisées par le fait que la somme ou la différence des numéros de ligne et de colonne sont constantes. Ainsi si  $i$  désigne la colonne de la reine à placer et  $j = 1, 2, \dots, i - 1$  une des colonnes précédentes alors :

- Deux reines sont sur une même ligne si  $r[i] = r[j]$ .
- Deux reines sont sur une même diagonale si le coefficient angulaire de la droite qui les relie est  $\pm 1$  (angle d'une diagonale d'un carré de  $45^\circ$ ).

Ces deux conditions s'écrivent (les reines des colonnes  $i$  et  $j$  se menacent) :

$$\frac{r[i] - r[j]}{i - j} = \pm 1$$

ce qui s'écrit aussi :

$$|r[i] - r[j]| = |i - j|$$



Écrivez une fonction `issafe(cb, k)` qui, sur un `Plateau cb`, teste et renvoie `\linline Vrai@` s'il n'y a pas de conflit de mettre la reine en colonne  $k$ , ligne `cb[k]` avec toutes les  $(k-1)$ -ème reines déjà placées.



Validez vos définitions et votre fonction avec la solution.

**Solution C++**    @[pgreines.cpp]

```
/**
/**
  Test de position valide (pas de conflit)
  @param[in] cb - un Plateau
  @param[in] k - numero de reine
  @return Vrai s'il n'y a pas de conflit de mettre la reine
  en colonne k, ligne CB[k] avec toutes les (k-1)-ieme reines
  deja placees
*/
bool issafe(const Plateau& cb, unsigned k)
{
  // La reine 0 ne peut être en prise
  // avec ses precedentes
  if (0 == k)
  {
    return true;
  }
  // Pour chaque reine deja placee
  for (unsigned j = 0; j < k; ++j)
  {
    // Delta sur les lignes
    int deltaY = cb[k] - cb[j];

    // Delta sur les colonnes: NOTE k > j donc k-j est positif
    int deltaX = static_cast<int>(k - j);

    // Si même ligne, diagonales montante ou descendante
    // alors même ligne: conflit
    if (deltaY * (deltaX - deltaY) * (deltaX + deltaY) == 0)
    {
      return false;
    }
  }
  // Ici aucun conflit
  return true;
}
```

## 2.2 Résolution récursive

Dans ce problème la complexité n'est pas à craindre car le plus mauvais des programmes récursifs est encore raisonnablement rapide.

### Stratégie récursive

Elle s'énonce : « Sachant qu'il y a des reines bien placées dans les lignes 1 à  $k - 1$  et connaissant leur place, trouver toutes les solutions possibles en plaçant convenablement les reines de  $k$  à  $n$ . »

**Algorithme de placement**

L'algorithme `placerRN(k)` qui tente de placer la  $k$ -ème reine est alors :

- Si  $k=n$  alors écrire la solution obtenue.
- Sinon Parcourir toutes les cases de la ligne  $k$  :
  - Si une case est libre (c.-à-d. non condamnée par les reines déjà en place) alors :
    - Y mettre la reine  $k$  puis
    - Résoudre le problème pour  $k+1$ .
  - Libérer la reine  $k$ .



Écrivez la procédure `placerRN(cb, sols, k)` qui réalise l'algorithme de placement de la reine  $k$  sur un `Plateau cb` comme décrit ci-dessus. Dans le cas d'une solution, elle l'enregistre dans `Solutions sols`.



Déduisez une procédure `dfsRN(cb, sols)` qui réalise la résolution récursive sur un `Plateau cb` et mémorise toutes les solutions dans `Solutions sols`.



Écrivez une procédure `afficherSolutions(sols)` qui affiche les `Solutions sols`.



Validez vos procédures avec la solution.

**Solution C++**    @[pgreines.cpp]

```
/**
 * Version recursive: Tente le placement de la reine
 * @param[in,out] cb - un Plateau
 * @param[in,out] sols - les Solutions
 * @param[in] k - numero de reine
 */
void placerRN(Plateau& cb, Solutions& sols, unsigned k)
{
    // Y a-t-il n reines placees?
    // si oui, enregistre la solution
    if (k == cb.size())
    {
        sols.push_back(cb);
    }
    else
    {
        // Sinon on tente de placer la reine k
        for (unsigned j = 0; j < cb.size(); ++j)
        {
            // Place la reine colonne k, ligne j
            cb[k] = j;

            // Si pas conflit de reine avec les (k-1)-emes
            // precedentes: en placer une de plus
            if (issafe(cb, k))
            {
                placerRN(cb, sols, k+1);
            }
        }
    }
}
```

```

    }

    // Libere la reine colonne k
    cb[k] = -1;
  }
}
}

/**
 Lance la recherche
 @param[out] cb - un Plateau
 @param[out] sols - les Solutions
 */
void dfsRN(Plateau& cb, Solutions& sols)
{
  // Cas particulier!
  if (cb.size() == 0)
  {
    return;
  }
  // Reinitialise les donnees
  fill(cb.begin(), cb.end(), -1);
  sols.clear();

  // Lance le solveur a partir de la
  // premiere reine numero 0
  placerRN(cb, sols, 0);
}

/**
 Affiche les solutions
 @param[in] sols - les Solutions
 */
void afficherSolutions(const Solutions& sols)
{
  if (sols.empty())
  {
    cout<<"Aucune solution"<<endl;
  }
  else
  {
    cout<<"Il y a "<<sols.size()<<" solutions. On en affiche? ";
    unsigned ns;
    cin >> ns;
    Solutions::const_iterator it = sols.begin();
    for (int k = 1; k <= ns && it != sols.end(); ++k, ++it)
    {
      cout<<"Sol #"<<setw(3)<<k<<": [";
      copy(it->begin(), it->end(), ostream_iterator<int>(cout, " "));
      cout<<"]"<<endl;
    }
  }
}
}

```



Écrivez une procédure `test_dfsRN` qui saisit un entier représentant l'ordre de l'échiquier, instancie un échiquier de problème des reines, puis cherche et affiche l'ensemble des solutions au problème du placement des reines.



Testez. Exemple d'exécution :

```
Dimension n de l'échiquier? 8
Il y a 92 solutions. On en affiche? 5
Sol # 1: [0 4 7 5 2 6 1 3 ]
Sol # 2: [0 5 7 2 6 3 1 4 ]
Sol # 3: [0 6 3 5 7 1 4 2 ]
Sol # 4: [0 6 4 7 1 3 5 2 ]
Sol # 5: [1 3 5 7 2 0 6 4 ]
```



Validez votre procédure avec la solution.

**Solution C++** @[pgreines.cpp]

```
void test_dfsRN()
{
    // Saisit la taille de l'échiquier
    cout<<"Dimension n de l'échiquier? ";
    unsigned n;
    cin >> n;
    // Definit le plateau: ligne i, colonne CB[i]
    Plateau cb(n);
    // Les solutions
    Solutions sols;
    // Lance la resolution
    dfsRN(cb, sols);
    // Affiche le resultat
    afficherSolutions(sols);
}
```

## 2.3 Résolution à partir d'une position

Les solutions ne sont pas toutes distinctes car l'échiquier possède des symétries. Ainsi pour optimiser le programme récursif, on ne cherchera que les solutions principales au problème des reines c.-à-d. celles à partir desquelles elles peuvent toutes être trouvées en utilisant ces symétries (Sont considérées comme équivalentes des solutions déduites l'une de l'autre, par symétrie par rapport aux droites joignant les milieux de l'échiquier ou par rotation de l'échiquier).

La procédure actuelle fait parcourir à la première reine toute la première ligne. Or il suffit d'en parcourir la moitié et de compléter par symétrie. Cependant il n'est pas facile d'arrêter le parcours lorsque  $r[1]$  a atteint la valeur  $\frac{n}{2}$  mais il est facile de commencer le parcours à  $\frac{n}{2} + 1$ .



**Copiez/collez** la procédure `placerRN(cb, sols, k)` en la procédure `placerPosRN(cb, sols, k)`. Modifiez-la de sorte à lancer la recherche à partir de la ligne 0.



De même **copiez/collez** la procédure `dfsRN(cb, sols)` en la procédure `dfsPosRN(cb, sols, pos)`. Modifiez-la de sorte à placer la première reine en la ligne numéro `pos`.



Écrivez une procédure `dessinerEchiquier(cb)` qui « dessine » un `Plateau cb` des reines posées en utilisant les symboles 'Q' ou '-'.



Déduisez une procédure `dessinerSolutions(sols)` qui « dessine » les `Solutions sols`.



Validez vos procédures avec la solution.

### Solution C++ @[pgreines.cpp]

```
/**
 * Version recursive: Tente le placement de la reine
 * @param[in,out] cb - un Plateau
 * @param[in,out] sols - les Solutions
 * @param[in] k - numero de reine
 */
void placerPosRN(Plateau& cb, Solutions& sols, unsigned k)
{
    // Teste la condition d'arrêt
    if (k == cb.size())
    {
        sols.push_back(cb);
    }
    else
    {
        // Sinon lance la recherche a partir de la ligne 0
        bool placement = false;
        for (unsigned j = 0; j < cb.size() && !placement; ++j)
        {
            // Verifie si la position (j, k) n'est pas en prise;
            // si oui, invoque placer() pour poser la reine suivante
            // sur la prochaine colonne. Sinon tente le placement
            // de la reine sur la ligne suivante

            // Place la reine colonne k, ligne j
            cb[k] = j;

            // Tente le placement suivant
            if (issafe(cb, k))
            {
                placerPosRN(cb, sols, k+1);
            }

            // Libere la reine colonne k
            cb[k] = -1;
        }
    }
}
```



```

/**
 Lance la recherche, ligne pos
 @param[out] cb - un Plateau
 @param[out] sols - les Solutions
 @param[in] pos - ligne de la premiere reine
 */
void dfsPosRN(Plateau& cb, Solutions& sols, unsigned pos)
{
 // Reinitialise les donnees
 fill(cb.begin(), cb.end(), -1);
 sols.clear();

 // Place la premiere reine en (0, pos)
 cb[0] = pos;

 // Positionne les autres reines (colonnes 1 a size()-1)
 placerPosRN(cb, sols, 1);
}

```

```

/**
 "Dessine" un echiquier des reines posees
 en utilisant les symboles 'Q' ou '-'
 @param[in] cb - un Plateau
 */
void dessinerEchiquier(const Plateau& cb)
{
 // Affiche le repere des colonnes
 cout<<" ";
 for (unsigned x = 0; x < cb.size(); ++x)
 {
 cout<<x<<" ";
 }
 cout<<endl;

 // Affiche l'echiquier
 for (unsigned y = 0; y < cb.size(); ++y)
 {
 cout<<y<<" ";
 for (unsigned x = 0; x < cb.size(); ++x)
 {
 cout<<(static_cast<unsigned>(cb[x]) == y ? " Q" : "-");
 }
 cout<<endl;
 }
}

```

```

/**
 Dessine les solutions
 @param[in] sols - les Solutions
 */
void dessinerSolutions(const Solutions& sols)
{
 if (sols.empty())
 {
 cout<<"Aucune solution"<<endl;
 }
 else
 {
 cout<<"Il y a "<<sols.size()<<" solutions"<<endl;
 }
}

```

```

    for (Solutions::const_iterator it = sols.begin(); it != sols.end(); ++it)
    {
        dessinerEchiquier(*it);
        cout<<endl;
    }
}
}

```



Copiez/collez la procédure `test_dfsRN` en la procédure `test_dfsPosRN` puis complétez-la et modifiez-la de sorte à utiliser les procédures de ce problème.



Testez. Exemple d'exécution :

```

Dimension n de l'échiquier? 5
# de ligne de la reine en colonne 0? 3
Il y a 2 solutions
  0 1 2 3 4
0 - Q - - -
1 - - - - Q
2 - - Q - -
3 Q - - - -
4 - - - Q -

  0 1 2 3 4
0 - - - - Q
1 - Q - - -
2 - - - Q -
3 Q - - - -
4 - - Q - -

```



Validez votre procédure avec la solution.

### Solution C++ @[pgreines.cpp]

```

void test_dfsPosRN()
{
    // Saisit la taille de l'échiquier
    cout<<"Dimension n de l'échiquier? ";
    unsigned n;
    cin >> n;
    // Definit le plateau: ligne i, colonne CB[i]
    Plateau cb(n);
    // Les solutions
    Solutions sols;
    // Demande les informations complémentaires:
    cout<<"# de ligne de la reine en colonne 0? ";
    unsigned pos;
    cin >> pos;
    // Lance la résolution
    dfsPosRN(cb, sols, pos);
}

```

```
// Affiche le resultat  
dessinerSolutions(sols);  
}
```

## 2.4 Blocage de l'échiquier des reines

Ce problème cherche quel est le plus petit nombre de reines nécessaire pour bloquer l'échiquier de sorte qu'il n'est plus possible de placer une pièce sur une case sans qu'elle ne soit prise par une reine, c.-à-d. « Comment bloquer l'échiquier avec le minimum de reines ? ». **Attention** Avec  $n$  reines il y avait une reine par ligne et une par colonne. Ici ceci n'est plus vrai.



Donnez les bornes inférieure et supérieure pour le nombre de reines nécessaire pour bloquer l'échiquier standard (taille  $8 \times 8$ ).

### Aide simple

Le nombre  $x$  de reines n'étant pas donné il faut essayer des  $x$  croissants ou décroissants. Il faut trouver une borne inférieure pour le nombre de reines. Sur l'échiquier standard vide une reine peut bloquer 28 cases : il faut donc au moins trois reines pour bloquer l'échiquier. Il en faut au plus sept car l'échiquier est souvent bloqué avant d'avoir pu mettre la 8-ème reine ; il est même probable que six suffissent. Il faut explorer la tranche de trois à six reines (cas de l'échiquier d'ordre 8).



Proposez une stratégie du blocage de l'échiquier.

### Aide simple

Voici une idée. Placez  $k$  reines puis parcourez l'échiquier à la recherche d'une case libre. S'il n'y en a pas, affichez la solution. S'il y en a une, déplacez une reine et recommencez.

## 2.5 Heuristique des $n$ reines

Ce problème tente l'heuristique d'accessibilité pour résoudre le problème des reines.



En vous inspirant de l'heuristique du problème du cavalier @[Heuristique du cavalier d'Euler], formulez une heuristique pour ce problème.



Pourquoi la stratégie suivante est-elle intéressante ? « Pour chaque case, on inscrit le nombre de cases éliminées dans le restant de l'échiquier lorsqu'une reine y est placée (par ex. sur l'échiquier standard, un coin possède la valeur 22). On place alors une reine dans la case qui **supprime le moins** de possibilités pour le placement des autres reines. »



Écrivez les définitions du delta de déplacements ainsi que l'opération + | [sur les \linlinePosition](#)|.



Validez vos définitions et opérations avec la solution.

### Solution C++ @ [pgreines.cpp]

```

/**
 * Definit les delta du voisinage d'une case de l'échiquier
 */
Position arrDelta[] = {
    Position(-1, 0), Position(-1,-1),
    Position( 0,-1), Position( 1,-1),
    Position( 1, 0), Position( 1, 1),
    Position( 0, 1), Position(-1, 1)
};

/**
 * Vecteur base sur les tableaux
 */
typedef vector<Position> CVDeplacements;
const CVDeplacements delta(&arrDelta[0], &arrDelta[8]);

/**
 * Operation + sur les Position
 * @param[in] pos1 - Une Position
 * @param[in] pos2 - Une Position
 * @return la Position pos1 + pos2
 */
Position plusPos(const Position& pos1, const Position& pos2)
{
    return Position(pos1.getX()+pos2.getX(), pos1.getY()+pos2.getY());
}

/**
 * Surdefinit l'operation + sur les Position
 * @param[in] pos1 - Une Position
 * @param[in] pos2 - Une Position
 * @return la Position pos1 + pos2
 */
Position operator+(const Position& pos1, const Position& pos2)
{
    return Position(pos1.getX()+pos2.getX(), pos1.getY()+pos2.getY());
}

```



Écrivez une procédure `calculerHB(cb,hb,delta)` qui calcule le tableau des accessibilités Echiquier `hb` selon les `CVDeplacements delta` d'un Echiquier `cb`.



Écrivez une procédure `placerNaif(cb,hb,sols,pos0)` qui lance la pose des reines à partir d'une `Position pos0`. La recherche de la prochaine position de pose s'effectue en parcourant systématiquement tout l'Echiquier `hb` des accessibilités. En cas de solution, celle-ci est enregistrée dans les `Solutions sols`.



Validez vos procédures avec la solution.

### Solution C++ @[pgreines.cpp]

```

/*
  Calcule l'accessibilité de chacune des positions
  @param[in] cb - un Echiquier
  @param[in,out] hb - Echiquier des accessibilités
  @param[in] delta - delta de déplacements
*/
void calculerHB(const Echiquier& cb, Echiquier& hb, const CVDeplacements& delta)
{
  for (unsigned y = 0; y < cb.sizeN(); ++y)
  {
    for (unsigned x = 0; x < cb.sizeN(); ++x)
    {
      Position pos(x, y);

      unsigned naces = 0;
      for (unsigned nv = 0; nv < delta.size(); ++nv)
      {
        Position nvpos(pos);
        while (nvpos = nvpos + delta[nv], cb.contient(nvpos))
        {
          ++naces;
        }
      }
      hb.set(pos, naces);
    }
  }
}

/*
  Lance la pose des reines à partir d'une position
  Version naïve: recherche la prochaine position de pose en
  parcourant systématiquement tout l'échiquier
  @param[out] cb - un Echiquier
  @param[out] hb - Echiquier des accessibilités
  @param[out] sols - les Solutions
  @param[in] pos0 - Position de départ
*/
void placerNaif(Echiquier& cb, Echiquier& hb, vector<Position>& sols, const Position&
  pos0)
{
  // Reinitialise les données
  cb.reset(0);
  sols.clear();

  // Demarre la pose à partir de pos
  if ( not cb.contient(pos0) )
  {
    return;
  }

  // Effectue une pose tant que possible
  const unsigned NULL_ACCESS = 4 * cb.sizeN() + 1;

```

```

Position pos = pos0;
while(true)
{
    // Enfile la position et la marque sur l'échiquier
    sols.push_back(pos);
    cb.set(pos, 1);

    // Marque toutes les cellules résultant de la pose en pos
    for (unsigned nv = 0; nv < delta.size(); ++nv)
    {
        // Position suivante: On part de la position courante et
        // tant que la position (suivante) est dans l'échiquier,
        // on la marque
        Position nvpos = pos;
        while (nvpos = nvpos + delta[nv], cb.contient(nvpos))
        {
            cb.set(nvpos, 1);
        }
    }

    // Recherche la prochaine position de pose de reine (position
    // non condamnée et d'accessibilité la plus petite)
    Position nextpos; // prochaine position
    unsigned naces = NULL_ACCESS; // sa valeur d'accessibilité
    for (unsigned y = 0; y < cb.sizeN(); ++y)
    { for (unsigned x = 0; x < cb.sizeN(); ++x)
        {
            Position pos(x, y);
            if (cb.get(pos) == 0 and hb.get(pos) < naces)
            {
                naces = hb.get(pos) ;
                nextpos = pos;
            }
        }
    }

    // En cas d'échec: fin de la traversée
    if (NULL_ACCESS == naces)
    {
        break;
    }
    // Sinon, on se place sur la prochaine position
    pos = nextpos;
}
}

```



Écrivez une procédure `test_heuristiq` qui lance la recherche sur chacune des positions.



Testez. Exemple d'exécution :

```

Dimension n de l'échiquier? 12 12 12 12 12
12 14 14 14 12
12 14 16 14 12
12 14 14 14 12
12 12 12 12 12

```

```

Solution issue de (1,1)
[(1,1) (3,0) (4,2) (0,4) (2,3) ]
Solution issue de (2,2)
[(2,2) (1,0) (4,1) (0,3) (3,4) ]
Solution issue de (4,4)
[(4,4) (1,0) (0,2) (3,1) (2,3) ]

```



De même, écrivez une procédure `placerOptm(cb, hb, sols, pos0)` qui réalise une version améliorée de la recherche de pose en utilisant une liste de toutes les positions encore disponibles.



Validez votre procédure avec la solution.

### Solution C++ @ [pgreines.cpp]

```

/**
 * Version ameliee: recherche la prochaine position de pose
 * en utilisant une liste de toutes les positions encore disponibles
 * @param[out] cb - un Echiquier
 * @param[out] hb - Echiquier des accessibilites
 * @param[out] sols - les Solutions
 * @param[in] pos0 - Position de depart
 */
void placerOptm(Echiquier& cb, Echiquier& hb, vector<Position>& sols, const Position&
    pos0)
{
    // Reinitialise les donnees
    cb.reset(0);
    sols.clear();

    // Demarre la pose a partir de pos
    if ( not cb.contient(pos0) )
    {
        return;
    }

    // Liste des positions encore disponibles.
    // au depart: toutes les positions sont non marquées
    list<Position> dispo;
    for (unsigned y = 0; y < cb.sizeN(); ++y)
    {
        for (unsigned x = 0; x < cb.sizeN(); ++x)
        {
            dispo.push_back(Position(x, y));
        }
    }

    // Effectue une pose tantque possible
    const unsigned NULL_ACCESS = 4 * cb.sizeN() + 1;

    Position pos = pos0;
    while(true)

```

```

{
  // Enfile la position et la marque sur l'échiquier
  sols.push_back(pos);
  cb.set(pos, 1);

  // Marque toutes les cellules résultant de la pose en pos
  for (unsigned nv = 0; nv < delta.size(); ++nv)
  {
    // Position suivante: On part de la position courante et
    // tant que la position (suivante) est dans l'échiquier,
    // on la marque:
    Position nvpos = pos;
    while (nvpos = nvpos + delta[nv], cb.contient(nvpos))
    {
      cb.set(nvpos, 1);
    }
  }

  // Recherche la prochaine position de pose de reine de plus
  // petite valeur d'accessibilité: traverse la liste des
  // positions disponibles et retire celles qui ont été
  // marquées lors de la dernière pose
  Position nextpos; // prochaine position
  unsigned naces = NULL_ACCESS; // sa valeur d'accessibilité

  list<Position>::iterator it = dispo.begin();
  while( it != dispo.end() )
  {
    // Si la position est marquée: on la supprime des positions
    // encore disponibles, puis on passe à la position suivante.
    // Sinon: si elle est d'accessibilité plus petite, on la prend;
    // puis on passe à la position suivante
    if ( cb.get(*it) != 0 )
    {
      it = dispo.erase(it);
    }
    else
    {
      unsigned a = hb.get(*it);
      if (a < naces)
      {
        naces = a;
        nextpos = *it;
      }
      ++it;
    }
  }

  // En cas d'échec: fin de la traversée
  if (NULL_ACCESS == naces)
  {
    break;
  }

  // Sinon, on se place sur la prochaine position
  pos = nextpos;
}
}

```





Testez en appelant cette procédure dans `test_heuristiq`.



Validez votre procédure avec la solution.

### Solution C++ @[pgreines.cpp]

```
void test_heuristiq()
{
    // Saisit la taille de l'échiquier
    cout<<"Dimension n de l'échiquier? ";
    unsigned n;
    cin >> n;
    // Definit le plateau: ligne i, colonne CB[i]
    Echiquier cb(n);
    // Les solutions
    vector<Position> sols;
    // Tableau des accessibilités
    Echiquier hb(n);
    calculerHB(cb, hb, delta);
    // afficherEchiquier(hb, 3);
    // Lance la recherche
    for (unsigned y = 0; y < cb.sizeN(); ++y)
    {
        for (unsigned x = 0; x < cb.sizeN(); ++x)
        {
            Position pos(x, y);
            //placerNaif(cb, hb, sols, pos);
            placerOptm(cb, hb, sols, pos);

            if (sols.size() == cb.sizeN())
            {
                cout<<"Solution issue de "<<pos<<endl;
                afficherVector(sols);
            }
        }
    }
}
```

## 2.6 Force brute des reines

Ce problème tente différentes approches par force brute pour résoudre le problème des reines.



Trouvez des solutions au problème des  $n$  reines à l'aide de l'approche par force brute aléatoire (voir l'approche par force brute dans le cas du cavalier).



Développez une approche exhaustive en traitant toutes les combinaisons possibles des  $n$  reines sur l'échiquier.



Comparez et faites ressortir les différences entre les approches par force brute aléatoire et exhaustive.

### 3 Références générales

**Comprend** [Moliner-M1 :c3 :ex2] ■