

# Tris, complexité, backtracking et assertions

Algorithmique et structures de données, 2024-2025

---

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA  
2024-11-25

En partie inspirés des supports de cours de P. Albuquerque

# Le tri à bulle

## Algorithme

- Parcours du tableau et comparaison des éléments consécutifs:
  - Si deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés.
- On recommence depuis le début du tableau jusqu'à avoir plus d'échanges à faire.

**Que peut-on dire sur le dernier élément du tableau après un parcours?**

## Algorithme

- Parcours du tableau et comparaison des éléments consécutifs:
  - Si deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés.
- On recommence depuis le début du tableau jusqu'à avoir plus d'échanges à faire.

## Que peut-on dire sur le dernier élément du tableau après un parcours?

- Le plus grand élément est **à la fin** du tableau.
  - Plus besoin de le traiter.
- A chaque parcours on s'arrête un élément plus tôt.

# Tri à bulle (2/4)

## Exemple

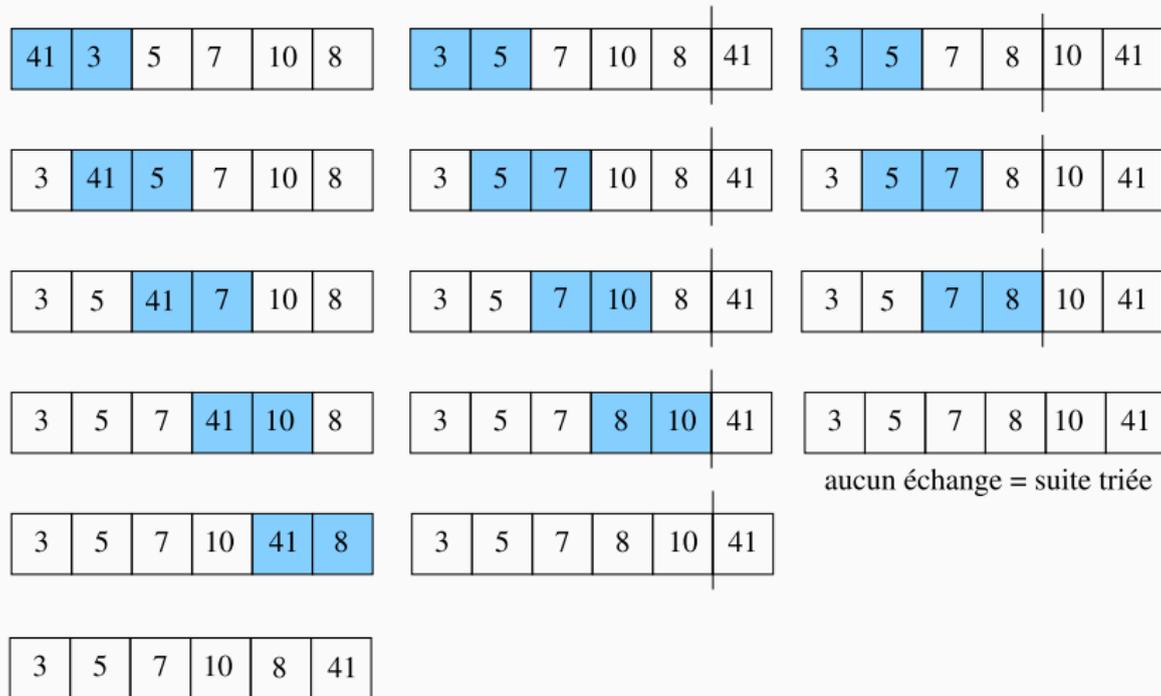


Figure 1: Tri à bulles d'un tableau d'entiers

**Exercice: écrire l'algorithme (poster le résultat sur matrix)**

## Tri à bulle (3/4)

**Exercice: écrire l'algorithme (poster le résultat sur matrix)**

```
rien tri_a_bulles(entier tableau[])
  pour i de longueur(tableau)-1 à 1:
    trié = vrai
    pour j de 0 à i-1:
      si (tableau[j] > tableau[j+1])
        échanger(tableau[j], tableau[j+1])
        trié = faux

    si trié
      retourner
```

Quelle est la complexité du tri à bulles?

### Quelle est la complexité du tri à bulles?

- Dans le meilleurs des cas:
  - Le tableau est déjà trié:  $\mathcal{O}(N)$  comparaisons.
- Dans le pire des cas,  $N \cdot (N - 1)/2 \sim \mathcal{O}(N^2)$ :

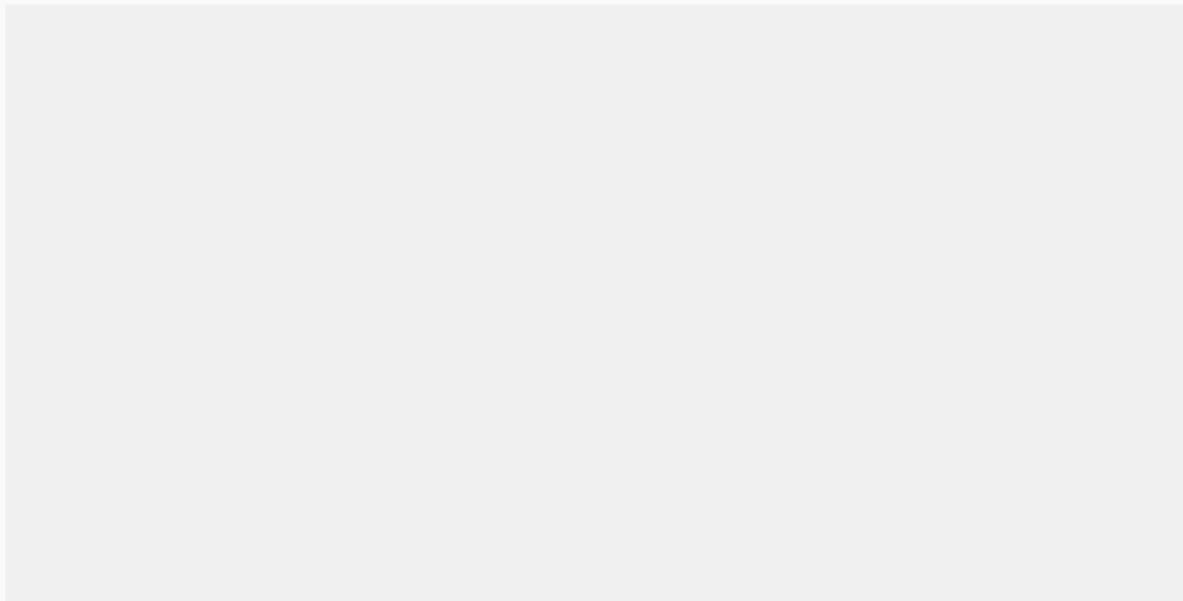
$$\sum_{i=1}^{N-1} i \text{ comparaison et } 3 \sum_{i=1}^{N-1} i \text{ affectations (swap)} \Rightarrow \mathcal{O}(N^2).$$

- En moyenne,  $\mathcal{O}(N^2)$  ( $N^2/2$  comparaisons).

# L'algorithme à la main

## Exercice *sur papier*

- Trier par tri à bulles le tableau [5, -2, 1, 3, 10, 15, 7, 4]



# Tri par insertion (1/3)

## But

- trier un tableau par ordre croissant

## Algorithme

Prendre un élément du tableau et le mettre à sa place parmi les éléments déjà triés du tableau.

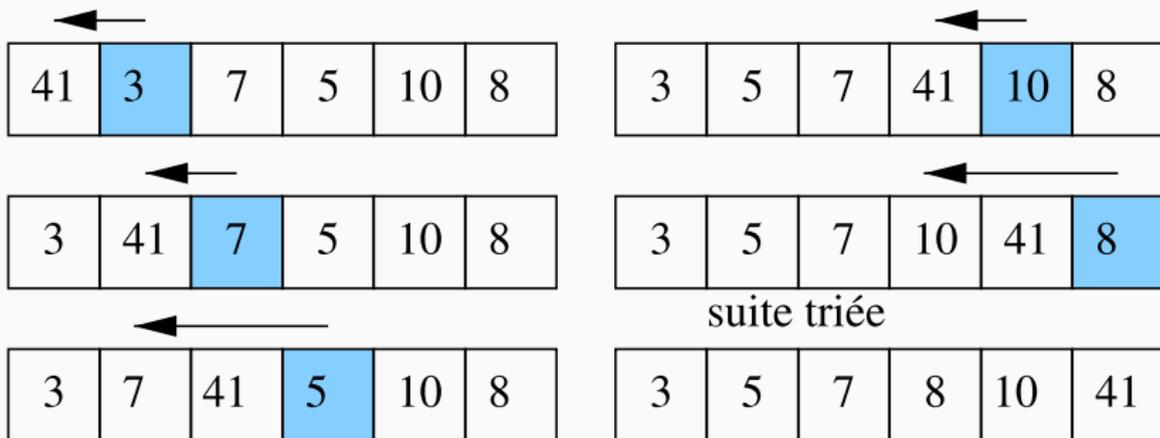


Figure 2: Tri par insertion d'un tableau d'entiers

## Tri par insertion (2/3)

**Exercice: Proposer un algorithme (en C)**

## Tri par insertion (2/3)

**Exercice: Proposer un algorithme (en C)**

```
void tri_insertion(int N, int tab[N]) {
    for (int i = 1; i < N; i++) {
        int tmp = tab[i];
        int pos = i;
        while (pos > 0 && tab[pos - 1] > tmp) {
            tab[pos] = tab[pos - 1];
            pos      = pos - 1;
        }
        tab[pos] = tmp;
    }
}
```

**Question: Quelle est la complexité?**

### Question: Quelle est la complexité?

- Parcours de tous les éléments ( $N - 1$  passages dans la boucle)
  - Placer: en moyenne  $i$  comparaisons et affectations à l'étape  $i$
- Moyenne:  $\mathcal{O}(N^2)$

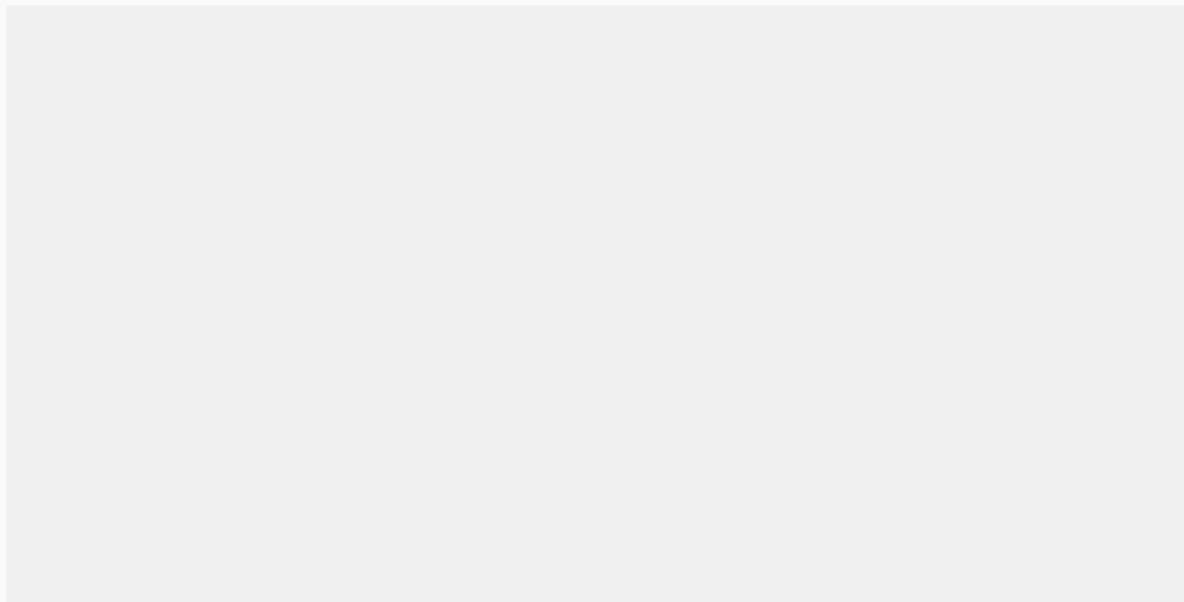
### Question: Quelle est la complexité?

- Parcours de tous les éléments ( $N - 1$  passages dans la boucle)
  - Placer: en moyenne  $i$  comparaisons et affectations à l'étape  $i$
- Moyenne:  $\mathcal{O}(N^2)$
- Pire des cas, liste triée à l'envers:  $\mathcal{O}(N^2)$
- Meilleurs des cas, liste déjà triée:  $\mathcal{O}(N)$

# L'algorithme à la main

## Exercice *sur papier*

- Trier par insertion le tableau [5, -2, 1, 3, 10]



# Complexité algorithmique du radix-sort (1/2)

## Pseudo-code

```
rien radix_sort(entier taille, entier tab[taille]):  
  # initialisation  
  entier val_min = valeur_min(taille, tab)  
  entier val_max = valeur_max(taille, tab)  
  decaler(taille, tab, val_min)  
  entier nb_bits = nombre_de_bits(val_max - val_min)  
  # algo  
  entier tab_tmp[taille]  
  pour pos de 0 à nb_bits:  
    alveole_0(taille, tab, tab_tmp, pos) # 0 -> taille  
    alveole_1(taille, tab, tab_tmp, pos) # taille -> 0  
    echanger(tab, tab_tmp)  
  # post-traitement  
  decaler(taille, tab, -val_min)
```

# Complexité algorithmique du radix-sort (2/2)

## Pseudo-code

```
rien radix_sort(entier taille, entier tab[taille]):  
# initialisation  
    entier val_min = valeur_min(taille, tab) #  $O(\text{taille})$   
    entier val_max = valeur_max(taille, tab) #  $O(\text{taille})$   
    decaler(taille, tab, val_min)           #  $O(\text{taille})$   
    entier nb_bits =  
        nombre_de_bits(val_max - val_min)  #  $O(\text{nb\_bits})$   
# algo  
    entier tab_tmp[taille]  
    pour pos de 0 à nb_bits:                #  $O(\text{nb\_bits})$   
        alveole_0(taille, tab, tab_tmp, pos) #  $O(\text{taille})$   
        alveole_1(taille, tab, tab_tmp, pos) #  $O(\text{taille})$   
        echanger(tab, tab_tmp)             #  $O(1)$   
# post-traitement  
    decaler(taille, tab, -val_min)         #  $O(N)$ 
```

# Complexité algorithmique du radix-sort (2/2)

## Pseudo-code

```
rien radix_sort(entier taille, entier tab[taille]):  
# initialisation  
    entier val_min = valeur_min(taille, tab) #  $O(\text{taille})$   
    entier val_max = valeur_max(taille, tab) #  $O(\text{taille})$   
    decaler(taille, tab, val_min)           #  $O(\text{taille})$   
    entier nb_bits =  
        nombre_de_bits(val_max - val_min)  #  $O(\text{nb\_bits})$   
# algo  
    entier tab_tmp[taille]  
    pour pos de 0 à nb_bits:                #  $O(\text{nb\_bits})$   
        alveole_0(taille, tab, tab_tmp, pos) #  $O(\text{taille})$   
        alveole_1(taille, tab, tab_tmp, pos) #  $O(\text{taille})$   
        echanger(tab, tab_tmp)             #  $O(1)$   
# post-traitement  
    decaler(taille, tab, -val_min)         #  $O(N)$ 
```

- Au final:  $\mathcal{O}(N \cdot (b + 4))$ .

# Complexité algorithmique du merge-sort (1/2)

## Pseudo-code

```
rien tri_fusion(entier taille, entier tab[taille])
    entier tab_tmp[taille];
    entier nb_etapes = log_2(taille) + 1;
    pour etape de 0 a nb_etapes - 1:
        entier gauche = 0;
        entier t_tranche = 2**etape;
        tant que (gauche < taille):
            fusion(
                tab[gauche..gauche+t_tranche-1],
                tab[gauche+t_tranche..gauche+2*t_tranche-1],
                tab_tmp[gauche..gauche+2*t_tranche-1]);
            gauche += 2*t_tranche;
    echanger(tab, tab_tmp);
```

## Complexité algorithmique du merge-sort (2/2)

### Pseudo-code

```
rien tri_fusion(entier taille, entier tab[taille])
  entier tab_tmp[taille]
  entier nb_etapes = log_2(taille) + 1
  pour etape de 0 a nb_etapes - 1: #  $O(\log_2(taille))$ 
    entier gauche = 0;
    entier t_tranche = 2**etape
    tant que (gauche < taille): #  $O(taille)$ 
      fusion(
        tab[gauche..gauche+t_tranche-1],
        tab[gauche+t_tranche..gauche+2*t_tranche-1],
        tab_tmp[gauche..gauche+2*t_tranche-1])
      gauche += 2*t_tranche
  echanger(tab, tab_tmp)
```

## Complexité algorithmique du merge-sort (2/2)

### Pseudo-code

```
rien tri_fusion(entier taille, entier tab[taille])
  entier tab_tmp[taille]
  entier nb_etapes = log_2(taille) + 1
  pour etape de 0 a nb_etapes - 1: #  $O(\log_2(taille))$ 
    entier gauche = 0;
    entier t_tranche = 2**etape
    tant que (gauche < taille): #  $O(taille)$ 
      fusion(
        tab[gauche..gauche+t_tranche-1],
        tab[gauche+t_tranche..gauche+2*t_tranche-1],
        tab_tmp[gauche..gauche+2*t_tranche-1])
      gauche += 2*t_tranche
    echanger(tab, tab_tmp)
```

- Au final:  $\mathcal{O}(N \log_2(N))$ .

## Complexité algorithmique du quick-sort (1/2)

### Pseudocode: quicksort

```
rien quicksort(entier tableau[], entier ind_min, entier ind_max)
  si (longueur(tab) > 1)
    ind_pivot = partition(tableau, ind_min, ind_max)
    si (longueur(tableau[ind_min:ind_pivot-1]) != 0)
      quicksort(tableau, ind_min, pivot_ind - 1)
    si (longueur(tableau[ind_pivot+1:ind_max-1]) != 0)
      quicksort(tableau, ind_pivot + 1, ind_max)
```

## Complexité algorithmique du quick-sort (2/2)

Quelle est la complexité du tri rapide?

### Quelle est la complexité du tri rapide?

- Pire des cas:  $\mathcal{O}(N^2)$ 
  - Quand le pivot sépare toujours le tableau de façon déséquilibrée ( $N - 1$  éléments d'un côté 1 de l'autre).
  - $N$  boucles et  $N$  comparaisons  $\Rightarrow N^2$ .
- Meilleur des cas (toujours le meilleur pivot):  $\mathcal{O}(N \cdot \log_2(N))$ .
  - Chaque fois le tableau est séparé en 2 parties égales.
  - On a  $\log_2(N)$  partitions, et  $N$  boucles  $\Rightarrow N \cdot \log_2(N)$ .
- En moyenne:  $\mathcal{O}(N \cdot \log_2(N))$ .

# Le problème des 8-reines

# Problème des 8-reines

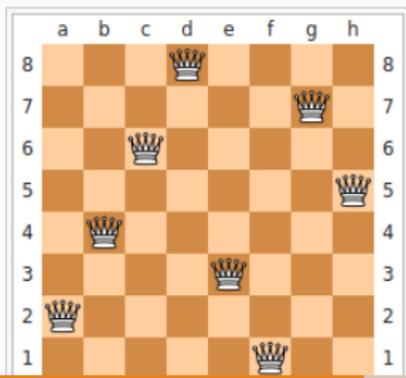
- Placer 8 reines sur un échiquier de  $8 \times 8$ .
- Sans que les reines ne puissent se menacer mutuellement (92 solutions).

## Conséquence

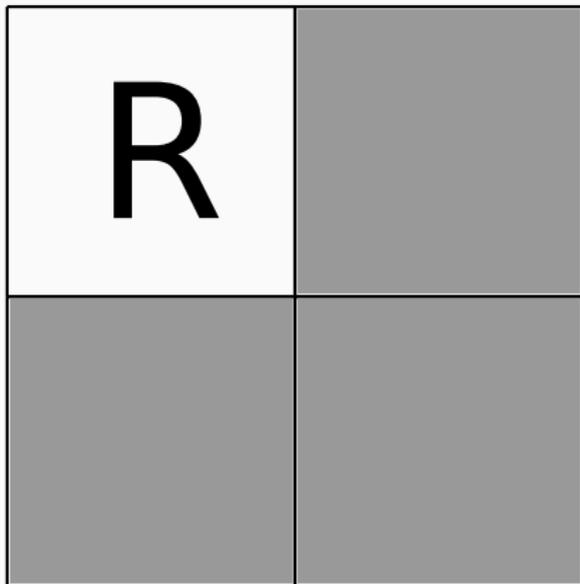
- Deux reines ne partagent pas la même rangée, colonne, ou diagonale.
- Donc chaque solution a **une** reine **par colonne** ou **ligne**.

## Généralisation

- Placer  $N$  reines sur un échiquier de  $N \times N$ .
- Exemple de **backtracking** (retour en arrière)  $\Rightarrow$  récursivité.



## Problème des 2-reines



**Figure 4:** Le problème des 2 reines n'a pas de solution.

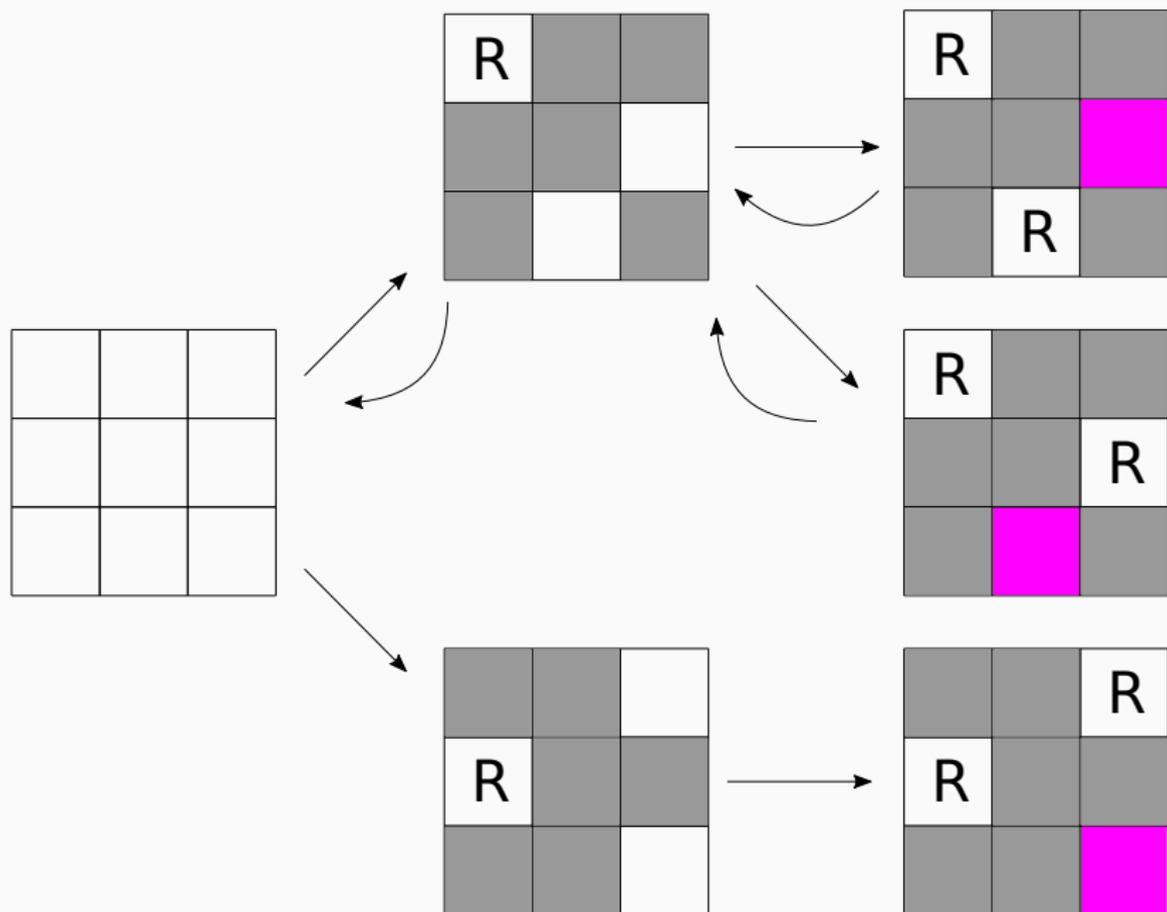
## Comment trouver les solutions?

- On pose la première reine sur la première case disponible.
- On rend inaccessibles toutes les cases menacées.
- On pose la reine suivante sur la prochaine case non-menacée.
- Jusqu'à ce qu'on puisse plus poser de reine.
- On revient alors en arrière jusqu'au dernier coup où il y avait plus qu'une possibilité de poser une reine.
- On recommence depuis là.

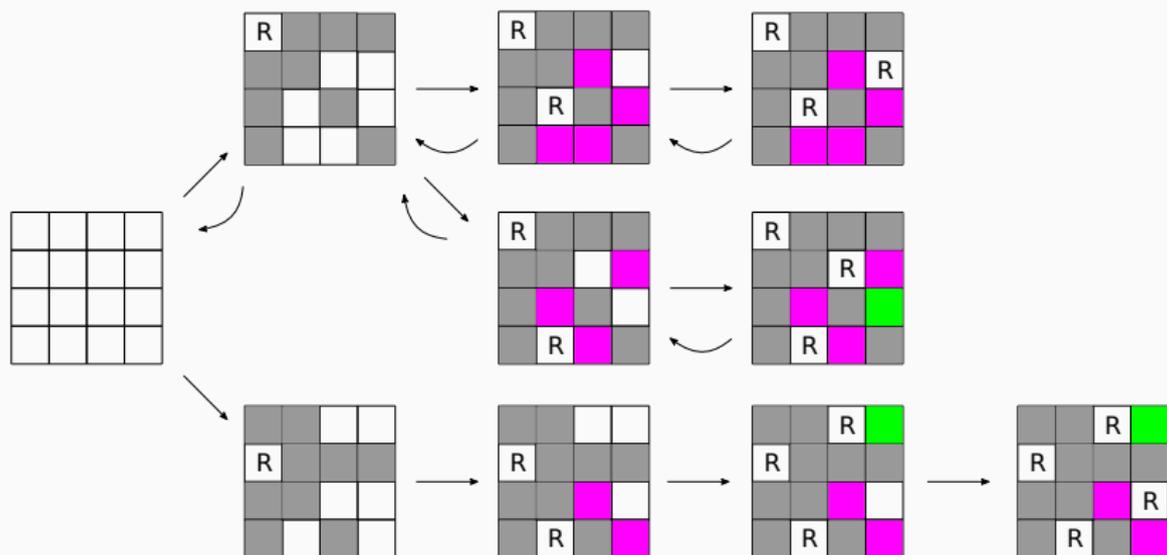
## Comment trouver les solutions?

- On pose la première reine sur la première case disponible.
- On rend inaccessibles toutes les cases menacées.
- On pose la reine suivante sur la prochaine case non-menacée.
- Jusqu'à ce qu'on puisse plus poser de reine.
- On revient alors en arrière jusqu'au dernier coup où il y avait plus qu'une possibilité de poser une reine.
- On recommence depuis là.
- Le jeu prend fin quand on a énuméré *toutes* les possibilités de poser les reines.

# Problème des 3-reines

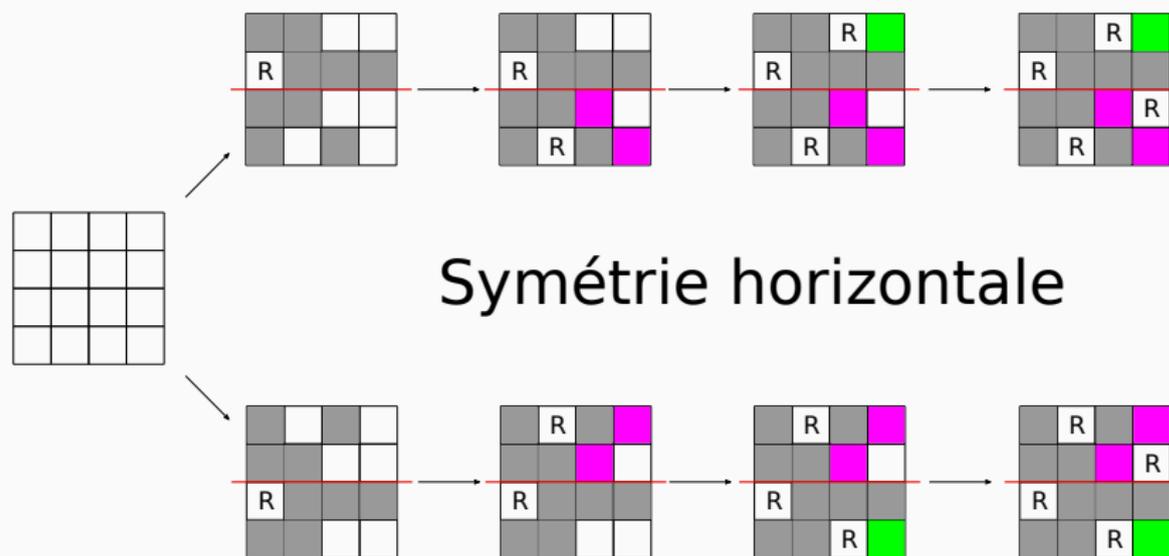


# Problème des 4-reines



**Figure 6:** Le problème des 4 reines a une solution.

# Problème des 4-reines, symétrie



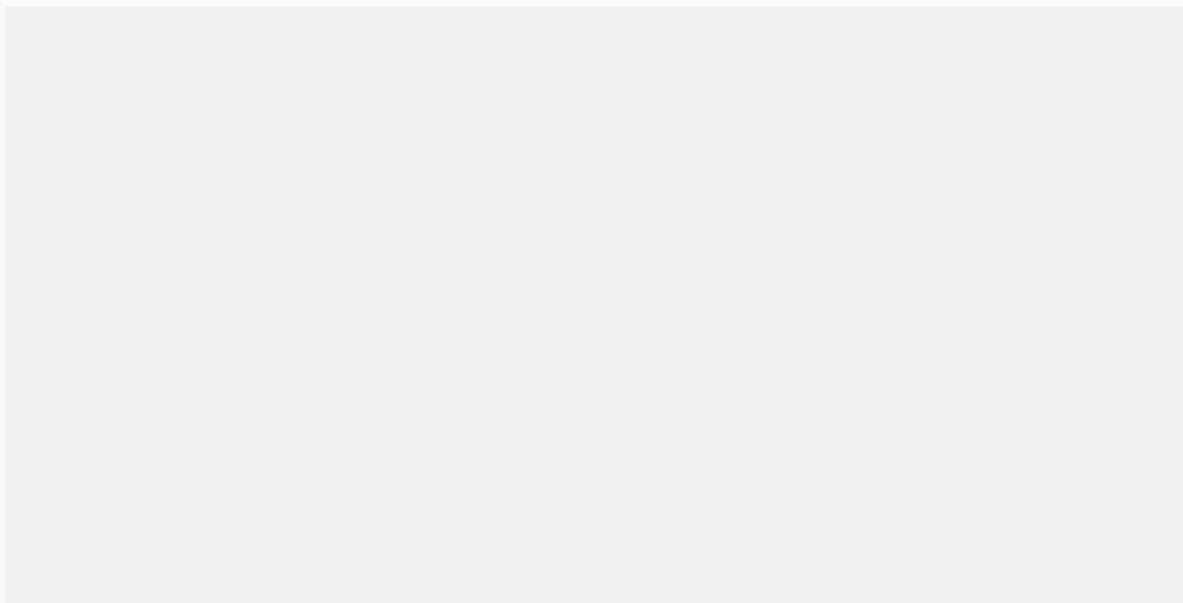
Symétrie horizontale

Figure 7: Le problème des 4 reines a une autre solution (symétrie horizontale).

# Problème des 5 reines

## Exercice: Trouver une solution au problème des 5 reines

- Faire une capture d'écran / une photo de votre solution et la poster sur matrix.



## Quelques observations sur le problème

- Une reine par colonne au plus.
- On place les reines sur des colonnes successives.
- On a pas besoin de “regarder en arrière” (on place “devant” uniquement).
- Trois étapes:
  - On place une reine dans une case libre.
  - On met à jour le tableau.
  - Quand on a plus de cases libres on “revient dans le temps” ou c’est qu’on a réussi.

## Le code du problème des 8 reines (1/5)

Quelle structure de données?

# Le code du problème des 8 reines (1/5)

## Quelle structure de données?

Une matrice de booléens fera l'affaire:

```
bool board[n][n];
```

## Quelles fonctionnalités?

# Le code du problème des 8 reines (1/5)

## Quelle structure de données?

Une matrice de booléens fera l'affaire:

```
bool board[n][n];
```

## Quelles fonctionnalités?

```
// Pour chaque ligne placer la reine sur toutes les colonnes  
// et compter les solutions  
void nbr_solutions(board, column, counter);  
// Copier un tableau dans un autre  
void copy(board_in, board_out);  
// Placer la reine à li, co et rendre inaccessible devant  
void placer_devant(board, li, co);
```

## Le code du problème des 8 reines (2/5)

### Le calcul du nombre de solutions

*// Calcule le nombre de solutions au problème des <n> reines*

```
rien nbr_solutions(board, column, count)
```

```
  pour chaque ligne
```

```
    si la case libre
```

```
      si column < n - 1
```

```
        copier board dans un "new" board,
```

```
        y poser une reine
```

```
        et mettre à jour ce "new" board
```

```
        nbr_solutions(new_board, column+1, count)
```

```
      sinon
```

```
        on a posé la n-ème et on a gagné
```

```
        count += 1
```

## Le code du problème des 8 reines (3/5)

### Le calcul du nombre de solutions

```
// Placer une reine et mettre à jour  
rien placer_devant(board, ligne, colonne)  
    board est occupé à ligne/colonne  
        toutes les cases des colonnes  
            suivantes sont mises à jour
```

## Le code du problème des 8 reines (4/5)

Compris? Alors écrivez le code et postez le!

# Le code du problème des 8 reines (4/5)

Compris? Alors écrivez le code et postez le!

## Le nombre de solutions

```
// Calcule le nombre de solutions au problème des <n> reines
void nb_sol(int n, bool board[n][n], int co, int *ptr_cpt) {
    for (int li = 0; li < n; li++) {
        if (board[li][co]) {
            if (co < n-1) {
                bool new_board[n][n]; // alloué à chaque nouvelle tentative
                copy(n, board, new_board);
                prises_devant(n, new_board, li, co);
                nb_sol(n, new_board, co+1, ptr_cpt);
            } else {
                *ptr_cpt = (*ptr_cpt)+1;
            }
        }
    }
}
```

# Le code du problème des 8 reines (5/5)

## Placer devant

```
// Retourne une copie du tableau <board> complété avec les positions  
// prises sur la droite droite par une reine placée en <board(li,co)>  
void placer_devant(int n, bool board[n][n], int li, int co) {  
    board[li][co] = false; // position de la reine  
    for (int j = 1; j < n-co; j++) {  
        // horizontale et diagonales à droite de la reine  
        if (j <= li) {  
            board[li-j][co+j] = false;  
        }  
        board[li][co+j] = false;  
        if (li+j < n) {  
            board[li+j][co+j] = false;  
        }  
    }  
}
```