

Backtracking et piles

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA
2025-12-02

En partie inspiré des supports de cours de P. Albuquerque

Le problème des 8-reines

Problème des 8-reines

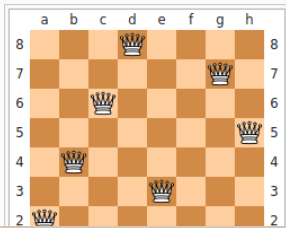
- Placer 8 reines sur un échiquier de 8×8 .
- Sans que les reines ne puissent se menacer mutuellement (92 solutions).

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

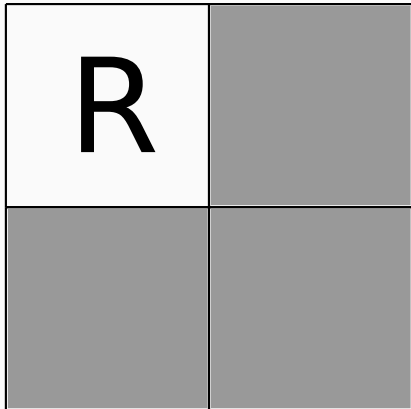


Figure 2 : Le problème des 2 reines n'a pas de solution.

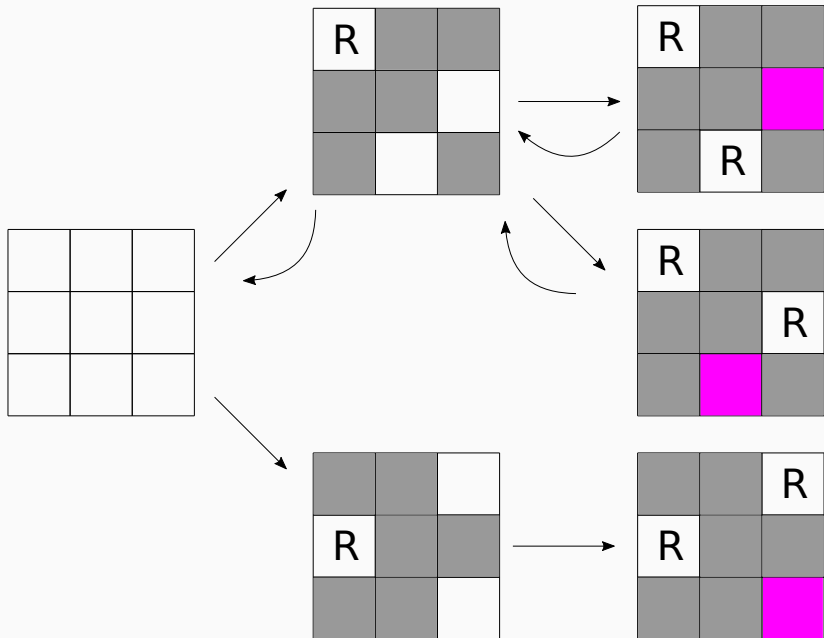
Comment trouver les solutions ?

- 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 ne 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 ne 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

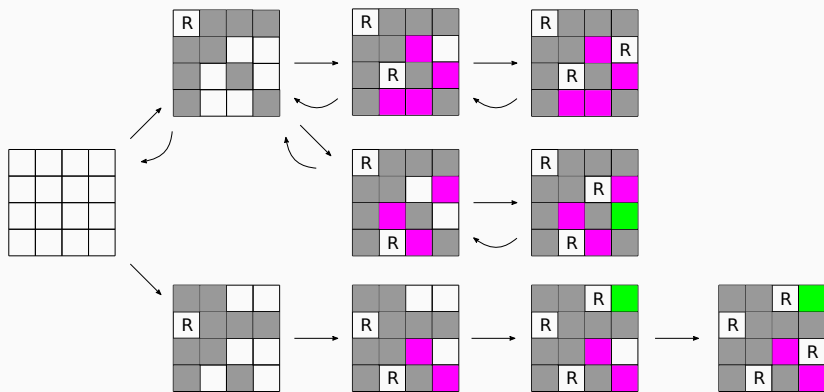


Figure 4 : Le problème des 4 reines a une solution.

Problème des 4-reines, symétrie

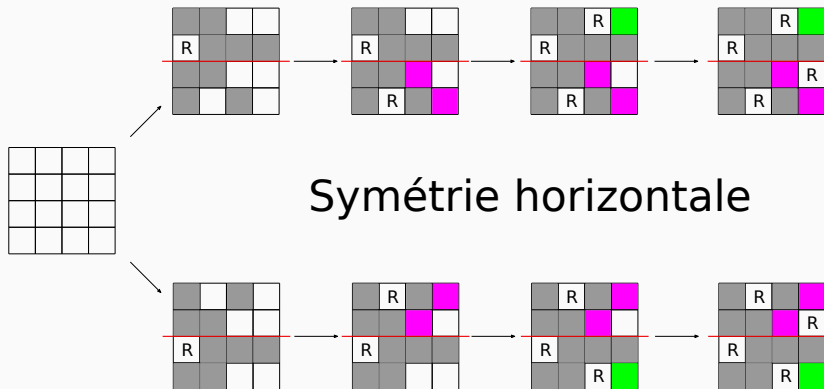
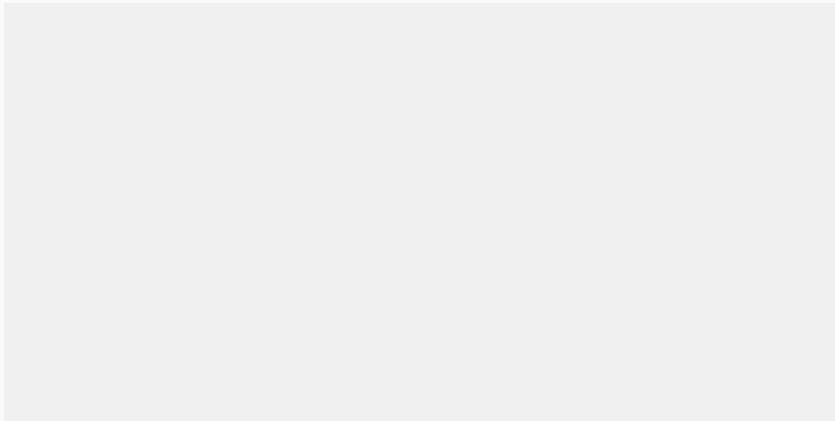


Figure 5 : Le problème des 4 reines a une autre solution (symétrie horizontale).

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 n'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 n'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  
rien nbr_solutions(board, column, counter);  
// Copier un tableau dans un autre  
rien copier(board_in, board_out);  
// Placer la reine à row, column et rendre inaccessible devant  
rien placer_devant(board, row, column);
```

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, counter)  
  pour chaque ligne  
    si la board[ligne][column] 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, counter)  
      sinon  
        on a posé la n-ième et on a gagné  
        counter += 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, row, column)  
    board est occupé à row/column  
        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 nbr_solutions(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
                copier(n, board, new_board);
                prises_devant(n, new_board, li, co);
                nbr_solutions(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 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;  
        }  
    }  
}
```

Les piles

Les piles (1/5)

Qu'est-ce donc ?

- Structure de données abstraite...

Les piles (1/5)

Qu'est-ce donc ?

- Structure de données abstraite...
- de type LIFO (*Last in first out*).

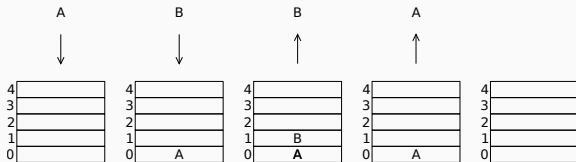


Figure 6 : Une pile où on ajoute A, puis B avant de les retirer. Source : [Wikipedia](#)

Des exemples de la vraie vie

Les piles (1/5)

Qu'est-ce donc ?

- Structure de données abstraite...
- de type LIFO (*Last in first out*).

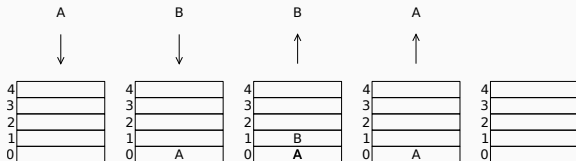


Figure 6 : Une pile où on ajoute A, puis B avant de les retirer. Source : [Wikipedia](#)

Des exemples de la vraie vie

- Pile d'assiettes, de livres, ...
- Adresses visitées par un navigateur web.
- Les calculatrices du passé (en polonaise inverse).
- Les boutons *undo* de vos éditeurs de texte (aka *u* dans vim).

Les piles (2/5)

Fonctionnalités

Les piles (2/5)

Fonctionnalités

1. Empiler (push) : ajouter un élément sur la pile.
2. Dépiler (pop) : retirer l'élément du sommet de la pile et le retourner.
3. Pile vide ? (is_empty?).

Les piles (2/5)

Fonctionnalités

1. Empiler (push) : ajouter un élément sur la pile.
2. Dépiler (pop) : retirer l'élément du sommet de la pile et le retourner.
3. Pile vide ? (is_empty?).
4. Jeter un œil (peek) : retourner l'élément du sommet de la pile (sans le dépiler).
5. Nombre d'éléments (length).

Comment faire les 4, 5 à partir de 1 à 3 ?

Les piles (2/5)

Fonctionnalités

1. Empiler (push) : ajouter un élément sur la pile.
2. Dépiler (pop) : retirer l'élément du sommet de la pile et le retourner.
3. Pile vide ? (is_empty?).
4. Jeter un œil (peek) : retourner l'élément du sommet de la pile (sans le dépiler).
5. Nombre d'éléments (length).

Comment faire les 4, 5 à partir de 1 à 3 ?

4. Dépiler l'élément, le copier, puis l'empiler à nouveau.
5. Dépiler jusqu'à ce que la pile soit vide, puis empiler à nouveau.

Les piles (2/5)

Fonctionnalités

1. Empiler (push) : ajouter un élément sur la pile.
2. Dépiler (pop) : retirer l'élément du sommet de la pile et le retourner.
3. Pile vide ? (is_empty?).
4. Jeter un œil (peek) : retourner l'élément du sommet de la pile (sans le dépiler).
5. Nombre d'éléments (length).

Comment faire les 4, 5 à partir de 1 à 3 ?

4. Dépiler l'élément, le copier, puis l'empiler à nouveau.
5. Dépiler jusqu'à ce que la pile soit vide, puis empiler à nouveau.

Existe en deux goûts

- Pile avec ou sans limite de capacité (à concurrence de la taille de la mémoire).

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser ?

Les piles (3/5)

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser ?

Et oui vous avez deviné : un tableau !

La structure : de quoi avons-nous besoin (pile de taille fixe) ?

Les piles (3/5)

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser ?

Et oui vous avez deviné : un tableau !

La structure : de quoi avons-nous besoin (pile de taille fixe) ?

```
#define MAX_CAPACITY 500
typedef struct _stack {
    int data[MAX_CAPACITY]; // les données
    int top;                 // indice du sommet
} stack;
```

Initialisation

Les piles (4/5)

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide ?

Les piles (4/5)

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide ?

```
bool stack_is_empty(stack s) {  
    return s.top == -1;  
}
```

Empiler (ajouter un élément au sommet)

Les piles (4/5)

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide ?

```
bool stack_is_empty(stack s) {  
    return s.top == -1;  
}
```

Empiler (ajouter un élément au sommet)

```
void stack_push(stack *s, int val) {  
    s->top += 1;  
    s->data[s->top] = val;  
}
```

Les piles (5/5)

Dépiler (enlever l'élément du sommet)

Les piles (5/5)

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

Les piles (5/5)

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack s) {  
    return s.data[s.top];  
}
```

Quelle est la complexité de ces opérations ?

Les piles (5/5)

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack s) {  
    return s.data[s.top];  
}
```

Quelle est la complexité de ces opérations ?

Voyez-vous des problèmes potentiels avec cette implémentation ?

Les piles (5/5)

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack s) {  
    return s.data[s.top];  
}
```

Quelle est la complexité de ces opérations ?

Voyez-vous des problèmes potentiels avec cette implémentation ?

- Empiler avec une pile pleine.
- Dépiler avec une pile vide.
- Jeter un oeil au sommet d'une pile vide.

- Il y a plusieurs façon de traiter les erreurs :
 - Ne rien faire (laisser la responsabilité à l'utilisateur).
 - Faire paniquer le programme (il plante plus ou moins violemment).
 - Utiliser des codes d'erreurs.

La panique

- En C, on a les `assert()` pour faire paniquer un programme.

Les assertions

Assertions (1/3)

```
#include <assert.h>
void assert(int expression);
```

Qu'est-ce donc ?

- Macro permettant de tester une condition lors de l'exécution d'un programme :
 - Si `expression == 0` (condition fausse), `assert()` affiche un message d'erreur sur `stderr` et termine l'exécution du programme.
 - Sinon l'exécution se poursuit normalement.
 - Peuvent être désactivés à la compilation avec `-DNDEBUG` (équivalent à `#define NDEBUG`)

À quoi ça sert ?

- Permet de réaliser des tests unitaires.
- Permet de tester des conditions catastrophiques d'un programme.
- **Ne permet pas** de gérer les erreurs.

Assertions (2/3)

Exemple

```
#include <assert.h>

void stack_push(stack *s, int val) {
    assert(s->top < MAX_CAPACITY-1);
    s->top += 1;
    s->data[s->top] = val;
}

int stack_pop(stack *s) {
    assert(s->top >= 0);
    s->top -= 1;
    return s->data[s->top+1];
}

int stack_peek(stack *s) {
    assert(s->top >= 0);
    return s->data[s->top];
}
```

Assertions (3/3)

Cas typiques d'utilisation

- Vérification de la validité des pointeurs (typiquement `!= NULL`).
- Vérification du domaine des indices (dépassement de tableau).

Bug vs. erreur de *runtime*

- Les assertions sont là pour détecter les bugs (erreurs d'implémentation).
- Les assertions ne sont pas là pour gérer les problèmes externes au programme (allocation mémoire qui échoue, mauvais paramètre d'entrée passé par l'utilisateur, ...).

Cas typiques d'utilisation

- Vérification de la validité des pointeurs (typiquement `!= NULL`).
- Vérification du domaine des indices (dépassement de tableau).

Bug vs. erreur de *runtime*

- Les assertions sont là pour détecter les bugs (erreurs d'implémentation).
- Les assertions ne sont pas là pour gérer les problèmes externes au programme (allocation mémoire qui échoue, mauvais paramètre d'entrée passé par l'utilisateur, ...).
- Mais peuvent être pratiques quand même pour ça...
- Typiquement désactivées dans le code de production.

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique ?

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique ?

```
// alloue une zone mémoire de size octets  
void *malloc(size_t size);  
// change la taille allouée à size octets (contiguïté garantie)  
void *realloc(void *ptr, size_t size);
```


La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique ?

```
// alloue une zone mémoire de size octets  
void *malloc(size_t size);  
// change la taille allouée à size octets (contiguïté garantie)  
void *realloc(void *ptr, size_t size);
```

Attention : malloc sert à allouer un espace mémoire (**pas** de notion de tableau).

Et maintenant ?

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique ?

```
// alloue une zone mémoire de size octets  
void *malloc(size_t size);  
// change la taille allouée à size octets (contiguïté garantie)  
void *realloc(void *ptr, size_t size);
```

Attention : malloc sert à allouer un espace mémoire (**pas** de notion de tableau).

Et maintenant ?

```
void stack_create(stack *s); // crée une pile avec une taille pa  
// vérifie si la pile est pleine et réalloue si besoin  
void stack_push(stack *s, int val);  
// vérifie si la pile est vide/trop grande  
// et réalloue si besoin  
void stack_pop(stack *s, int *ret);
```

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique ?

```
// alloue une zone mémoire de size octets  
void *malloc(size_t size);  
// change la taille allouée à size octets (contiguïté garantie)  
void *realloc(void *ptr, size_t size);
```

Attention : malloc sert à allouer un espace mémoire (**pas** de notion de tableau).

Et maintenant ?

```
void stack_create(stack *s); // crée une pile avec une taille pa  
// vérifie si la pile est pleine et réalloue si besoin  
void stack_push(stack *s, int val);  
// vérifie si la pile est vide/trop grande  
// et réalloue si besoin  
void stack_pop(stack *s, int *ret);
```