Chapter 4: Backtracking

Advanced Algorithms

P, NP and NP-hard | Go back and retry | State-space tree | Pruning | n-queens problem | Min-conflicts | Sudoku | Graph coloring and Hamiltonian paths | Word search | Complexity

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_04_Backtracking.html

Learning Outcomes

Concepts

Explain backtracking in terms of partial solutions, candidates, constraints and pruning; draw the state-space tree; distinguish P, NP and NP-hard.

Template

Write the recursive "choose → explore → undo" template in Java and adapt it to a new problem.

Applications

Solve n-queens, Sudoku, mazes, graph coloring, Hamiltonian path and word search step by step.

Analysis

Show why the worst case is exponential; assess the effect of pruning and heuristics (MRV, min-conflicts).

Hard Problems: P, NP and NP-Hard

P

Decision problems **solvable** in polynomial time (e.g. "a path of length ≤ k?", "a spanning tree of weight ≤ k?").

NP

Decision problems whose proposed solutions can be **verified** in polynomial time.

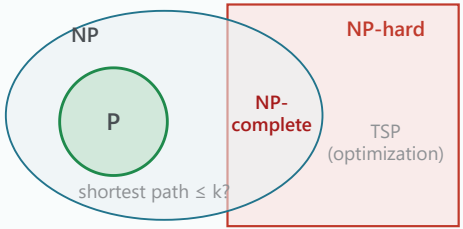
NP-hard

Every problem in NP reduces to it in polynomial time; it is at least as hard as the hardest problem in NP.

NP-complete

Both in NP and NP-hard. Examples: SAT, 3-SAT, Hamiltonian path and cycle, graph 3-coloring.

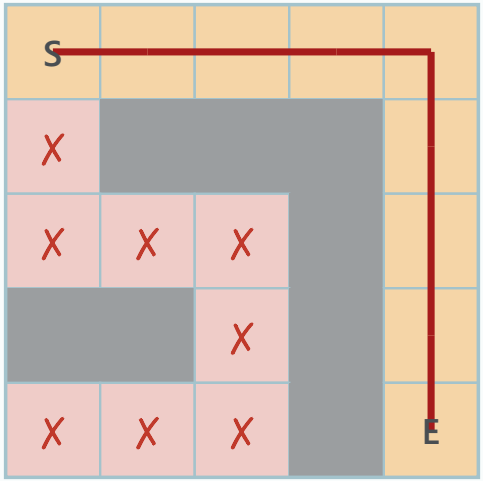
- Hamiltonian path, graph coloring, SAT and subset sum (as decision problems) are NP-complete; **optimization** TSP and knapsack are NP-hard.
- If P ≠ NP, none has a polynomial-time **exact** algorithm. Exact methods are exponential in the worst case, so we shrink the search by **pruning** (backtracking, branch and bound) or settle for an **approximate** solution.



CORE IDEA

Go Back and Try Again

- Some problems can only be solved by **trying different paths** and **backing out of wrong ones**.
- **Backtracking** applies this strategy systematically to hard problems.
- The solution is built step by step as a **partial solution**; if a step violates a constraint, that branch is **abandoned**.
- Unlike **brute force**, which blindly generates every possibility, it never extends an invalid partial solution.

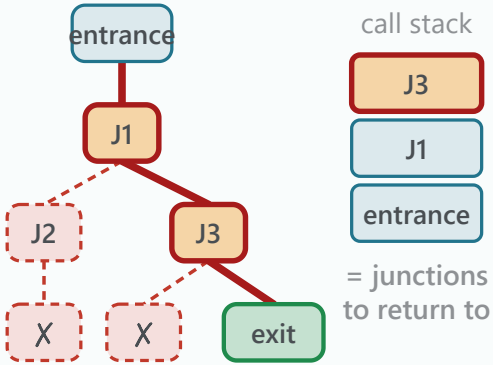


Path found (orange) and dead ends tried and abandoned (X)

CORE IDEA

A Traveler in a Maze

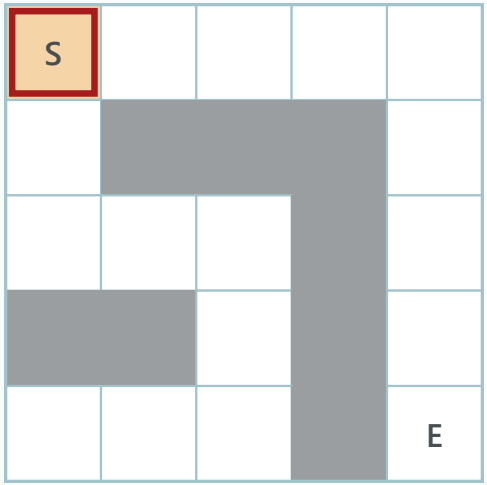
- Backtracking is like a person **walking through a maze**.
- At every **junction** the person makes a decision and moves on.
- If the chosen path does not lead to the exit, they **go back** and try another.
- Each step is a decision, and we check **whether it was right**.
- This **trial and error** continues until the problem is solved (or every option is exhausted).
- Uses: puzzles and AI games such as mazes, Sudoku and chess.



Key point: To go back, you must remember where you came from. Recursion gives this for free: the **call stack** is the list of junctions.

CORE IDEA

Backtracking in a Maze, Step by Step

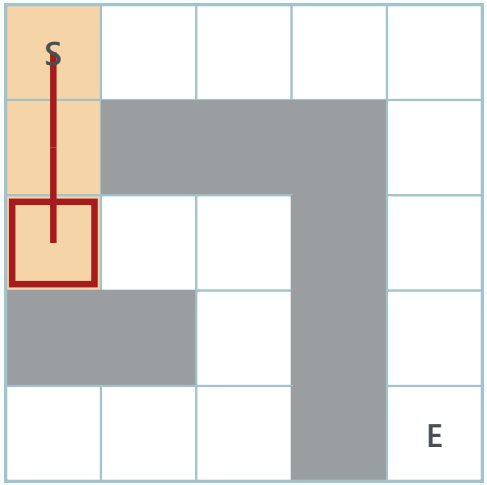


Start at cell S. Direction order: **down, right, up, left**. Each visited cell is marked, so no cell is entered twice.

Step 1 / 8

CORE IDEA

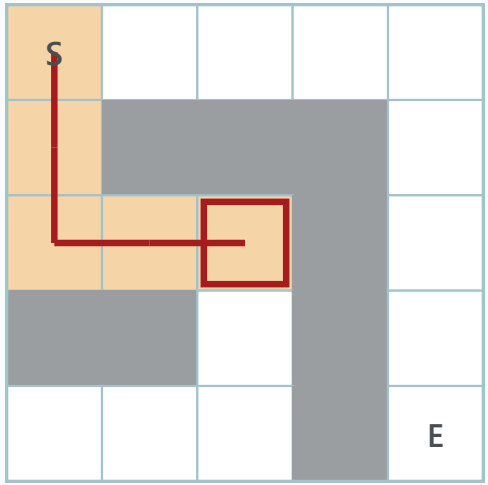
Backtracking in a Maze, Step by Step



Two steps down to (2,0). The next cell down is a wall, so the next direction (right) is tried.

CORE IDEA

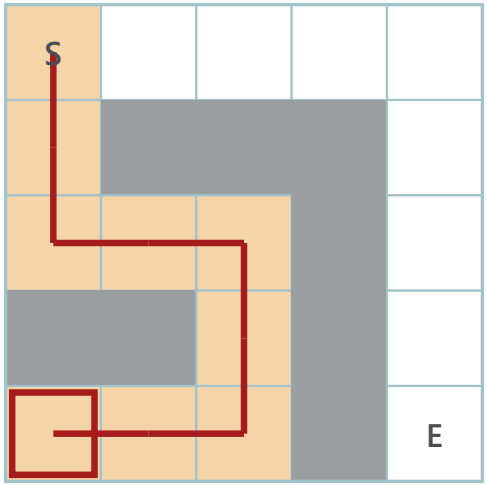
Backtracking in a Maze, Step by Step



Two steps right; (2,2) is a **junction**. Down is tried first; right and up are walls.

CORE IDEA

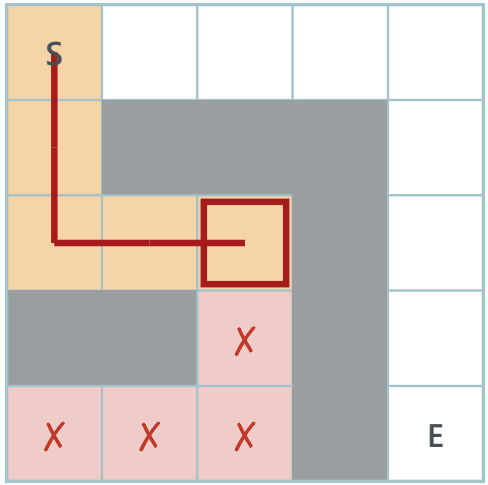
Backtracking in a Maze, Step by Step



The corridor going down and left hits a **dead end** at (4,0): all four directions are walls, borders or visited.

CORE IDEA

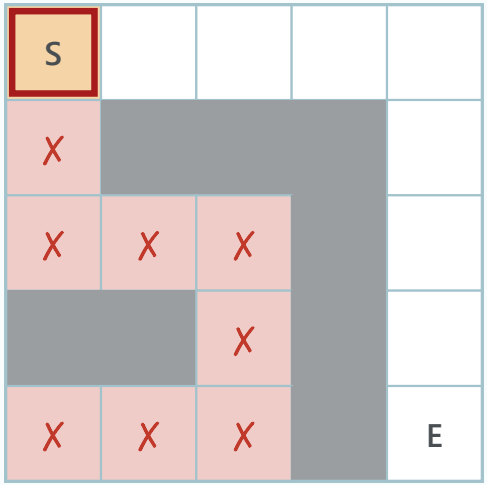
Backtracking in a Maze, Step by Step



Backtrack: (4,0), (4,1), (4,2), (3,2) are removed from the path (*X*). Junction (2,2) is retried, but its remaining directions are closed too.

CORE IDEA

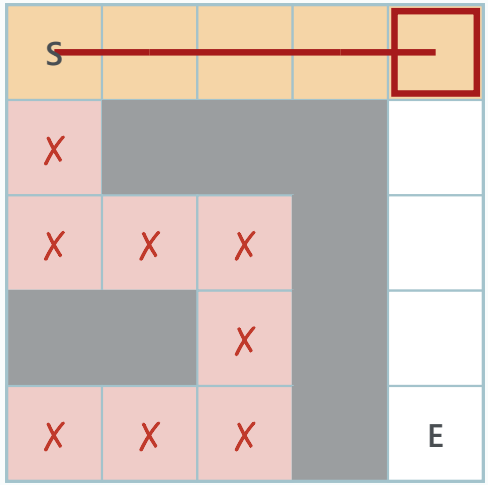
Backtracking in a Maze, Step by Step



Backtracking continues to the entrance: (2,2), (2,1), (2,0), (1,0) are dead ends too. At the entrance the next direction, **right**, is tried.

CORE IDEA

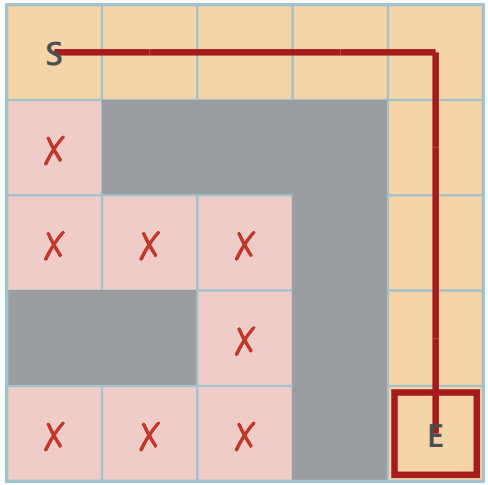
Backtracking in a Maze, Step by Step



Right along the top corridor to (0,4). Down is open.

CORE IDEA

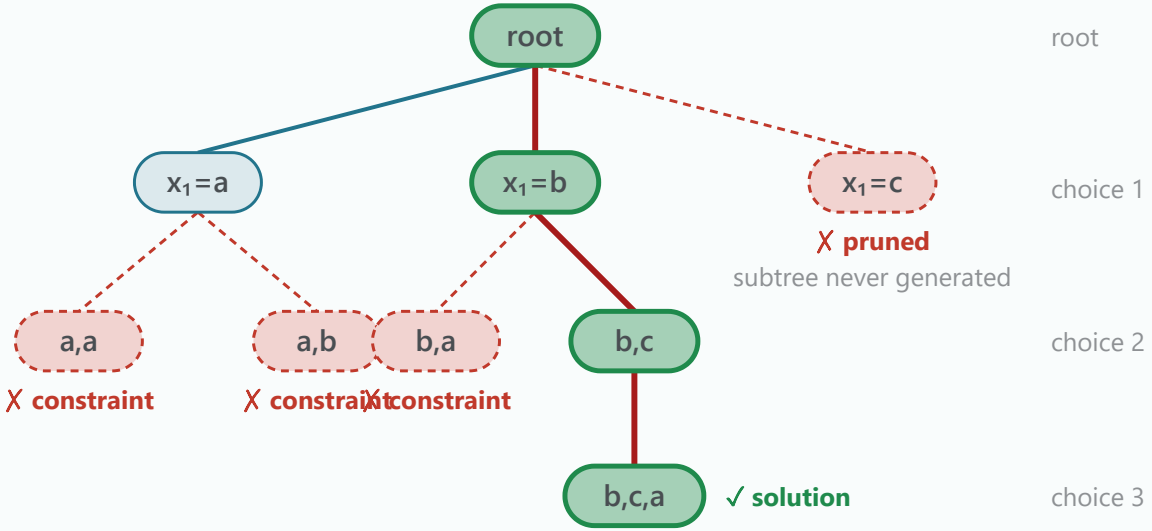
Backtracking in a Maze, Step by Step



The **exit** (E) is reached via (1,4), (2,4), (3,4). The path has 9 cells; 8 cells were tried as dead ends and abandoned.

CORE IDEA

State-Space Tree



Partial solution
The path from the root to a node: the choices made so far.

Candidate
The choices available at the next level: the node's children.

Constraint
If the partial solution is invalid, the node's **subtree is pruned**.

Goal test
Have we reached a complete, valid solution? (a leaf)

CORE IDEA

General Template: Choose → Explore → Undo

```
void backtrack(State partial) {
  if (isSolution(partial)) {
    record(partial);
    return;
  }
  for (Candidate c : candidates(partial)) {
    if (isValid(partial, c)) {
      add(partial, c);      // choose
      backtrack(partial);   // explore
      remove(partial, c);   // undo
    }
  }
}
```

1. **Goal test:** if the partial solution is complete, record it (or stop).
2. **Choose:** add the next candidate to the partial solution.
3. **Explore:** recurse one level deeper.
4. **Undo:** remove the candidate; the state returns to what it was before the call, and the next candidate is tried.

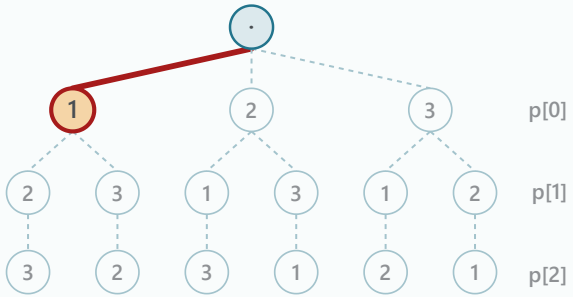
Pruning happens inside `isValid`: for an invalid candidate the recursive call is **never made**, so the whole subtree below it is skipped. If one solution is enough, `backtrack` returns a `boolean` and stops at the first `true`.

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```

CALL STACK

- **permute(k=1)**
- permute(k=0)
- main()



p[0..2] is empty and no number is used (k = 0). Candidates 1, 2, 3 are tried in order: **a = 1** is chosen, used[1] = true, and we go one level deeper.

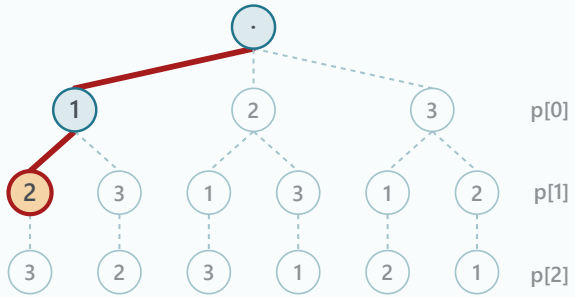
Step 1 / 11

VARIABLES

`k = 0` `a = 1` `p = [1]` `used = {1}`

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 1: **a = 2** is unused → chosen, used[2] = true.

Step 2 / 11

CALL STACK

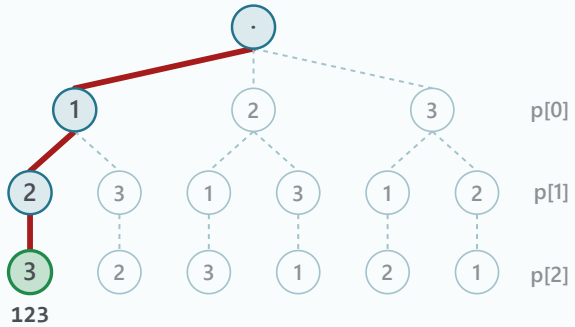
- **permute(k=2)**
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 1 | a = 2 | p = [1,2] | used = {1, 2}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!used[a]) {                     // valid
8        p[k] = a; used[a] = true;         // choose
9        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 2: only 3 is unused → chosen. k = 3 = p.length: **solution**, [1, 2, 3] is printed.

Step 3 / 11

CALL STACK

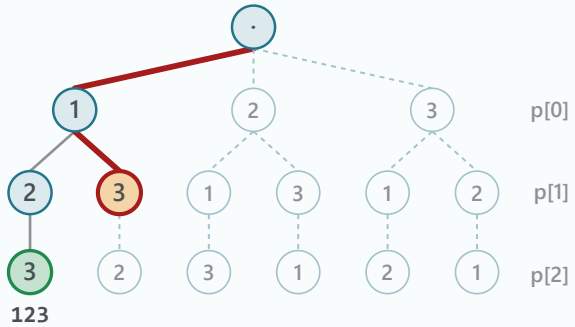
- **permute(k=3)**
- permute(k=2)
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 2 | a = 3 | p = [1,2,3] | used = {1, 2, 3}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
▶        used[a] = false;                  // undo
11     }
12   }
13 }
```



Undo: used[3] = false, used[2] = false. k = 1: **a = 3** is unused → chosen, used[3] = true.

Step 4 / 11

CALL STACK

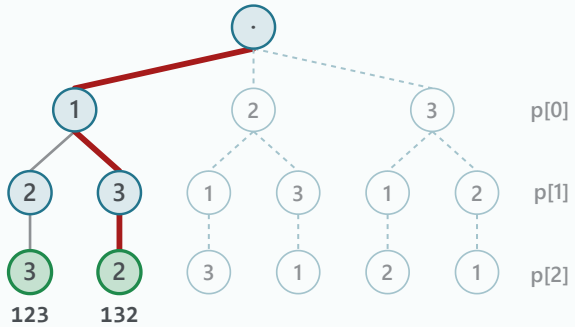
- **permute(k=2)**
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 1 | a = 3 | p = [1,3] | used = {1, 3}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
▶    if (k == p.length) {                  // solution
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!used[a]) {                     // valid
8        p[k] = a; used[a] = true;         // choose
9        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 2: only 2 is unused → chosen. k = 3 = p.length: **solution**, [1, 3, 2] is printed.

Step 5 / 11

CALL STACK

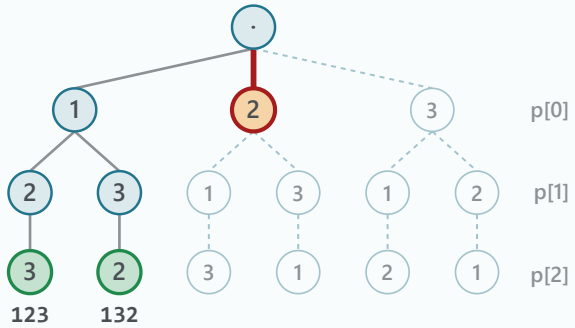
- **permute(k=3)**
- permute(k=2)
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 2 | a = 2 | p = [1,3,2] | used = {1, 2, 3}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
▶        used[a] = false;                  // undo
11     }
12   }
13 }
```



Undo: used[2] = false, used[3] = false, used[1] = false. k = 0: **a = 2** is unused → chosen, used[2] = true.

Step 6 / 11

CALL STACK

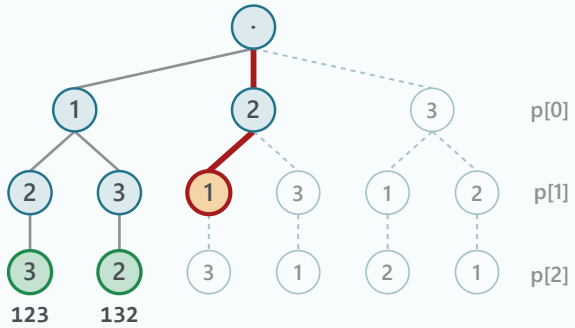
- `permute(k=1)`
- `permute(k=0)`
- `main()`

VARIABLES

`k = 0` `a = 2` `p = [2]` `used = {2}`

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 1: **a = 1** is unused → chosen, used[1] = true.

Step 7 / 11

CALL STACK

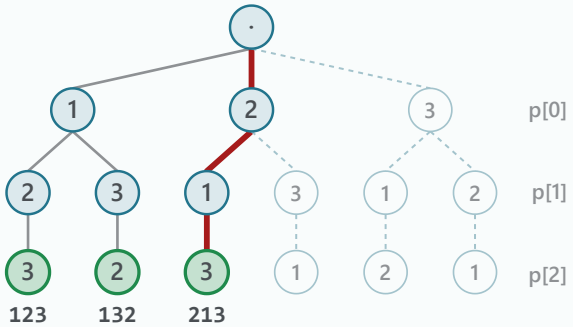
- **permute(k=2)**
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 1 | a = 1 | p = [2,1] | used = {1, 2}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
▶    if (k == p.length) {                  // solution
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!used[a]) {                     // valid
8        p[k] = a; used[a] = true;         // choose
9        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 2: only 3 is unused → chosen. k = 3 = p.length: **solution**, [2, 1, 3] is printed.

Step 8 / 11

CALL STACK

- **permute(k=3)**
- permute(k=2)
- permute(k=1)
- permute(k=0)
- main()

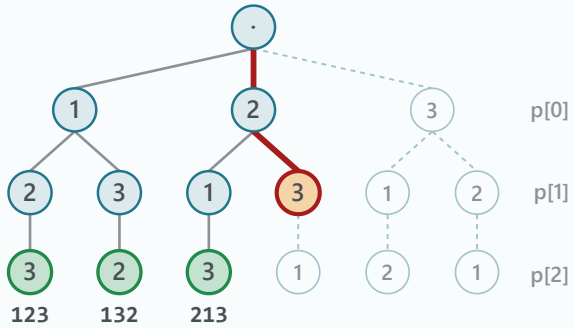
VARIABLES

k = **2** a = **3** p = **[2,1,3]**

used = **{1, 2, 3}**

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!used[a]) {                     // valid
▶        p[k] = a; used[a] = true;         // choose
▶        permute(p, used, k + 1);            // explore
▶        used[a] = false;                  // undo
11     }
12   }
13 }
```



Undo: used[3] = false, used[1] = false. k = 1: **a = 3** is unused → chosen, used[3] = true.

Step 9 / 11

CALL STACK

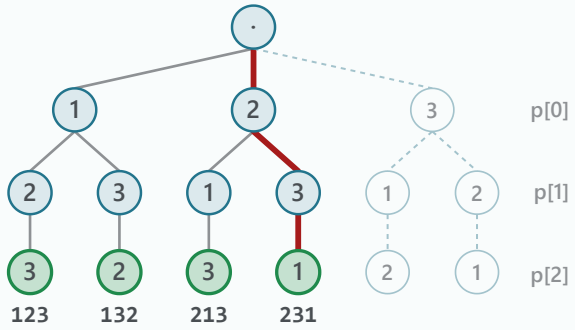
- **permute(k=2)**
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 1 | a = 3 | p = [2,3] | used = {2, 3}

The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!used[a]) {                     // valid
8        p[k] = a; used[a] = true;         // choose
9        permute(p, used, k + 1);            // explore
10       used[a] = false;                  // undo
11     }
12   }
13 }
```



k = 2: only 1 is unused → chosen. k = 3 = p.length: **solution**, [2, 3, 1] is printed.

Step 10 / 11

CALL STACK

- **permute(k=3)**
- permute(k=2)
- permute(k=1)
- permute(k=0)
- main()

VARIABLES

k = 2 | a = 1 | p = [2,3,1] | used = {1, 2, 3}

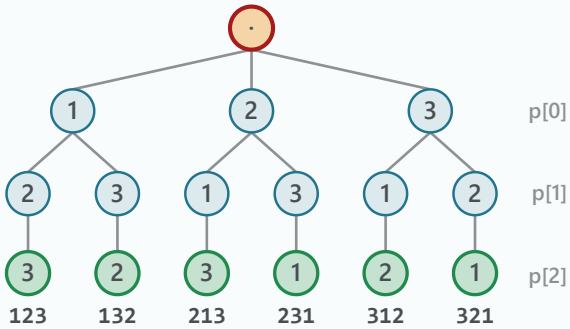
The Template in Action: Permutations of 1, 2, 3

```
1  void permute(int[] p, boolean[] used, int k) {
2    if (k == p.length) {                  // solution
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!used[a]) {                     // valid
8        p[k] = a; used[a] = true;         // choose
9        permute(p, used, k + 1);            // explore
▶        used[a] = false;                  // undo
11     }
12   }
13 }
```

CALL STACK

permute(k=0)

main()



After undoing, the branch starting with 3 is explored the same way: [3, 1, 2] and [3, 2, 1]. **3! = 6** permutations, 15 choose / 15 undo steps; used[] is restored on every return.

Step 11 / 11

VARIABLES

k = 0 | a = 3 | p = – | used = {}

CORE IDEA

Brute Force vs. Pruning: 8 Queens

Approach	Candidates examined	Description	log scale
Choose 8 of the 64 squares	4,426,165,368	C(64, 8): generate every placement, then test	
One queen per row	16,777,216	8^8 complete placements	
One queen per row, distinct columns	40,320	8! permutations	
Backtracking, **all** solutions	2056	safe placements (92 solutions)	
Backtracking, **first** solution	113	safe placements; 105 undos	

The power of pruning: because we back out as soon as a partial placement becomes invalid, only a few thousand **partial** placements are generated instead of 16.7 million complete ones. The worst case is still exponential, but in practice only a tiny part of the tree is visited.

CORE IDEA

Pros and Cons

Pros

+ A general method that applies to a **wide range** of problems.
+ Often efficient for problems whose solutions can be expressed as a **tree**.
+ **Easy** to understand and implement: short, recursive code.
+ Finds **all** solutions, or proves there are none (exhaustive search).

Cons

– Worst-case time is **exponential** (2^n, $n!$, 9^m ...).

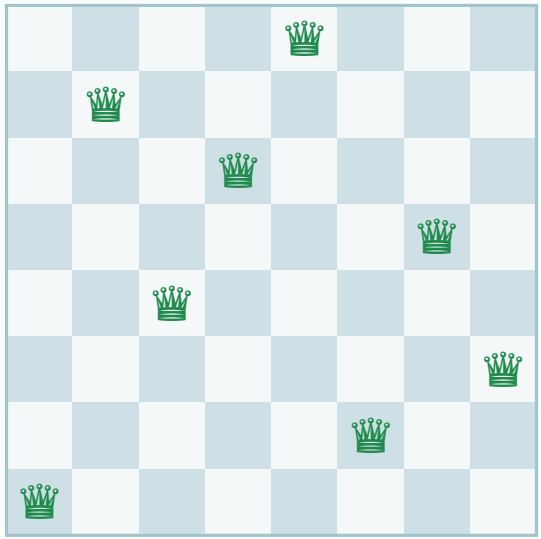
– **Slow** when the number of candidate solutions is huge; needs good pruning and heuristics.

– Deep recursion uses stack memory: O(depth).

– The same subproblem may be solved again and again along different paths (nothing is stored, unlike DP).

The Eight Queens Problem

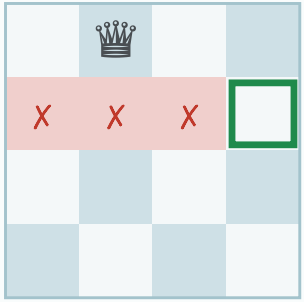
- Place **8 queens** on an 8×8 chessboard.
- Goal: a placement in which **no queen attacks another**. A queen attacks pieces on the same **row**, **column** and **diagonal**.
- General form: **n queens** on an n×n board (N-Queens). A solution exists for every $n \geq 1$ except $n = 2$ and $n = 3$.
- 8 queens has **92** solutions (12 fundamental ones up to symmetry).
- Backtracking is the standard way to solve problems like this.



One solution: a1, b7, c4, d6, e8, f2, g5, h3

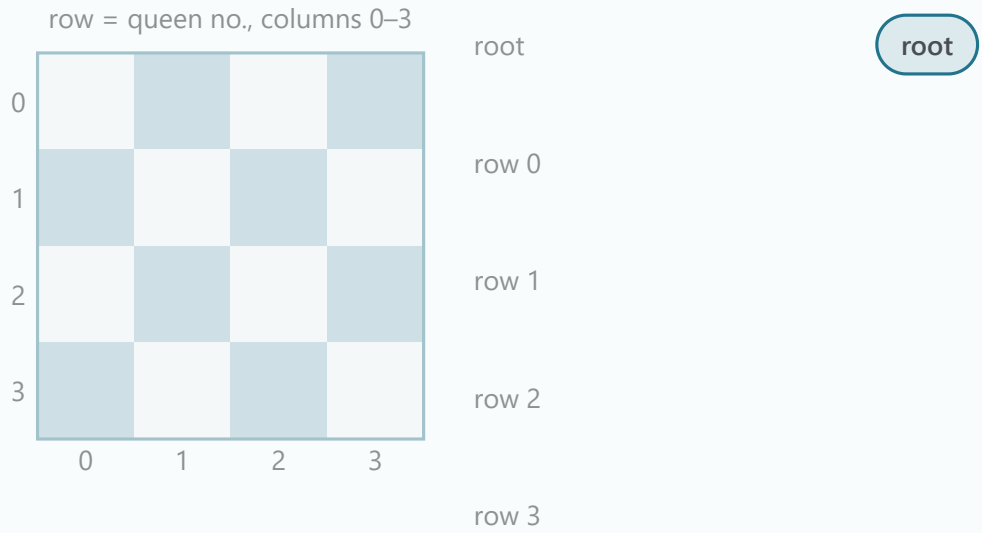
Backtracking Solution: Steps

1. Every square of the board starts **empty**.
2. The first queen goes into some column of the first row.
3. Check whether the new queen **conflicts** with earlier queens (column and both diagonals).
4. If there is no conflict, the position is **valid**; the next queen goes into the next row.
5. If **every column** of a row has been tried and none is safe, the previous queen is **moved** (backtrack).
6. Once all 8 queens are placed, a **solution** has been found.



Why row by row? Every row must hold exactly one queen. This observation **builds the row constraint into the structure**: only column and diagonal checks remain, and the search space drops from 64^8 to 8^8.

4 Queens: Board and State-Space Tree

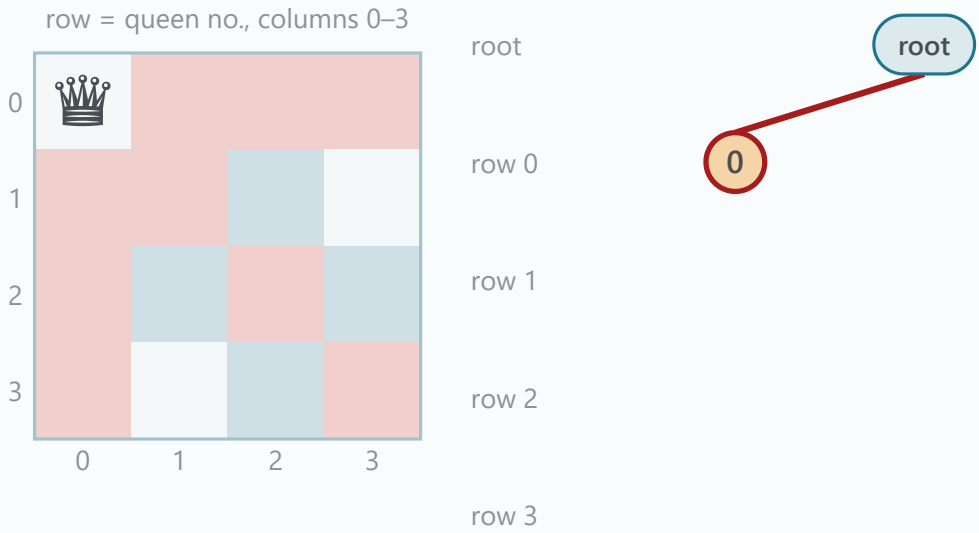


Empty board. Queens are placed **row by row**; in each row columns 0, 1, 2, 3 are tried in order. A tree node shows the **column** chosen in that row.

Step 1 / 13

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

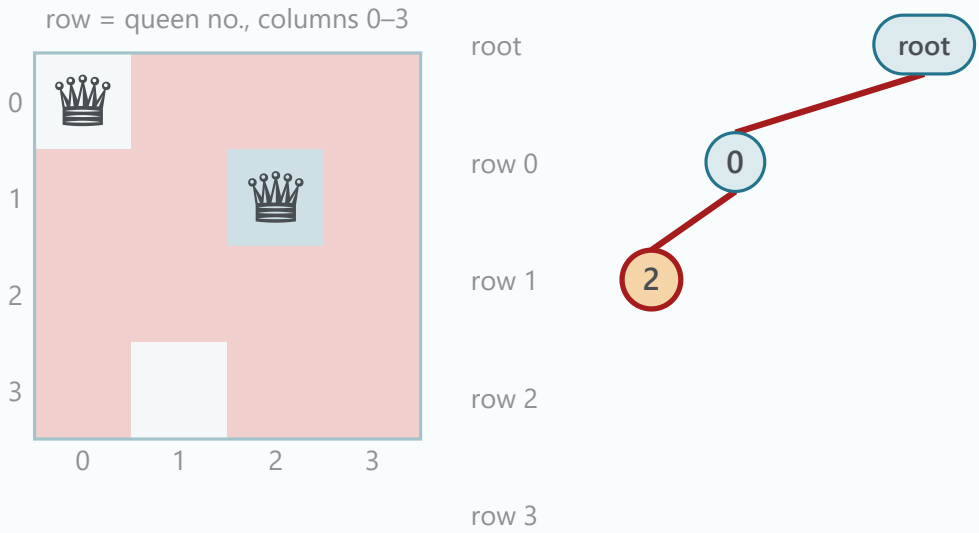
4 Queens: Board and State-Space Tree



Row 0: **column 0 is safe** → place a queen and go down a level (row 1).

Red square: attacked by the current queens · orange node: last placed · dashed ✗: abandoned branch · green: solution.

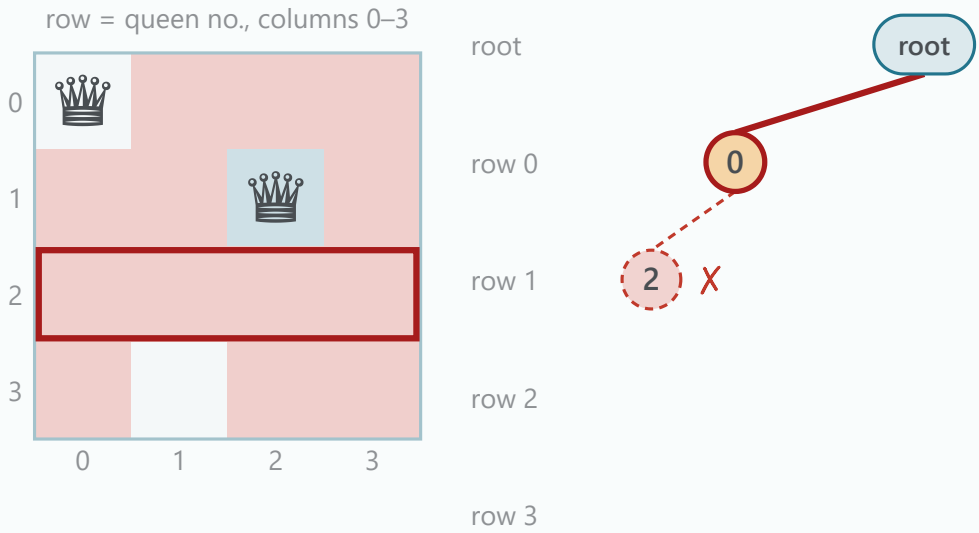
4 Queens: Board and State-Space Tree



Row 1: columns 0, 1 are attacked; **column 2 is safe** → place a queen and go down a level (row 2).

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

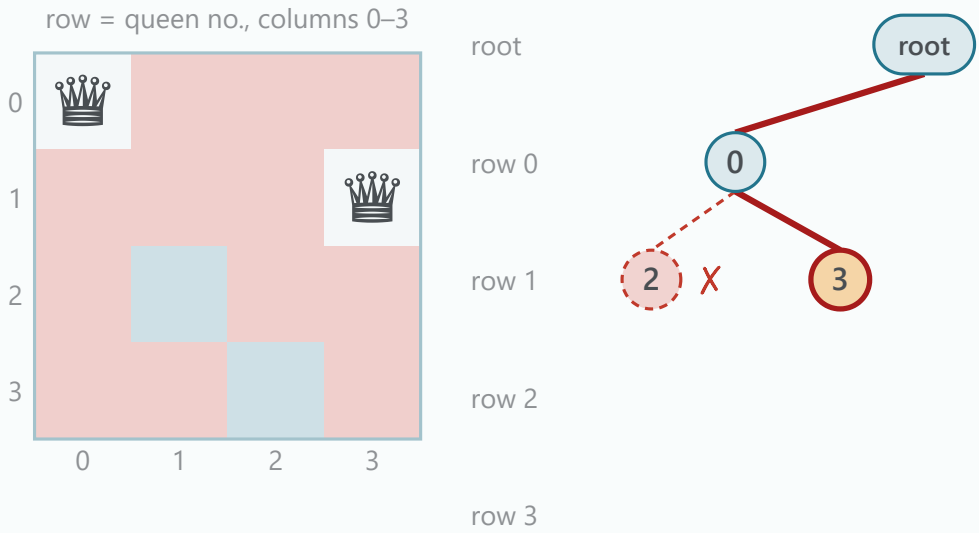
4 Queens: Board and State-Space Tree



Row 2: columns 0, 1, 2, 3 are attacked → **no safe column** (dead end). **Backtrack**: remove the queen in row 1 (column 2) and try the next column in that row.

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

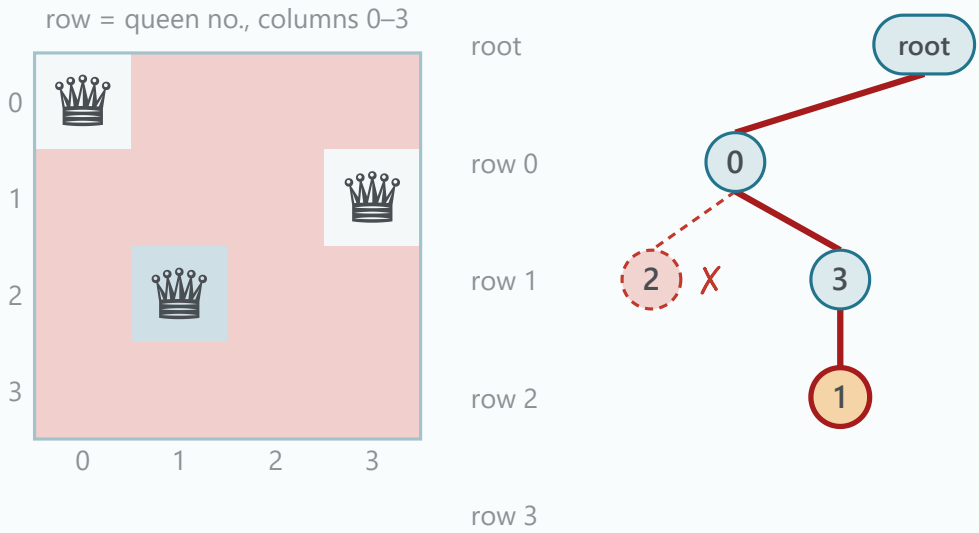
4 Queens: Board and State-Space Tree



Row 1: **column 3 is safe** → place a queen and go down a level (row 2).

Red square: attacked by the current queens · orange node: last placed · dashed X: abandoned branch · green: solution.

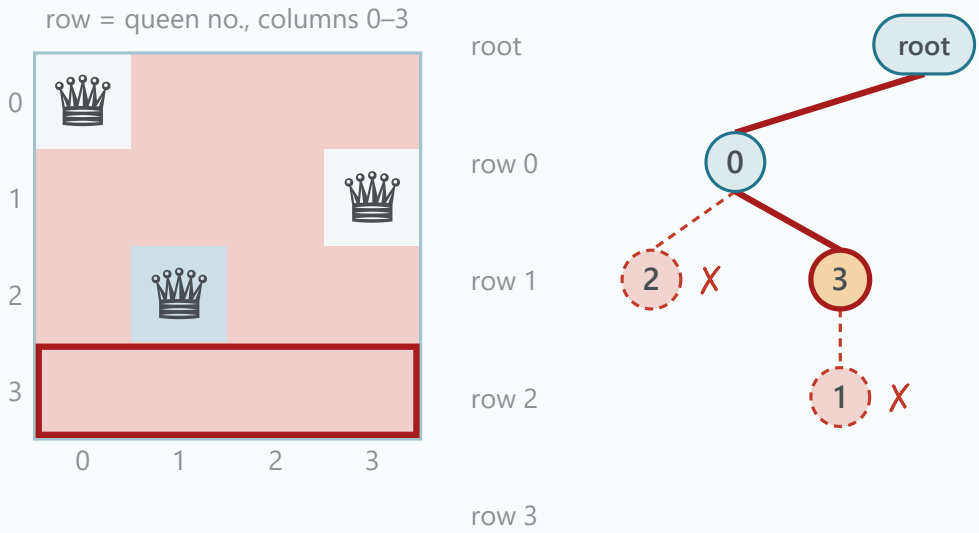
4 Queens: Board and State-Space Tree



Row 2: column 0 is attacked; **column 1 is safe** → place a queen and go down a level (row 3).

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

4 Queens: Board and State-Space Tree

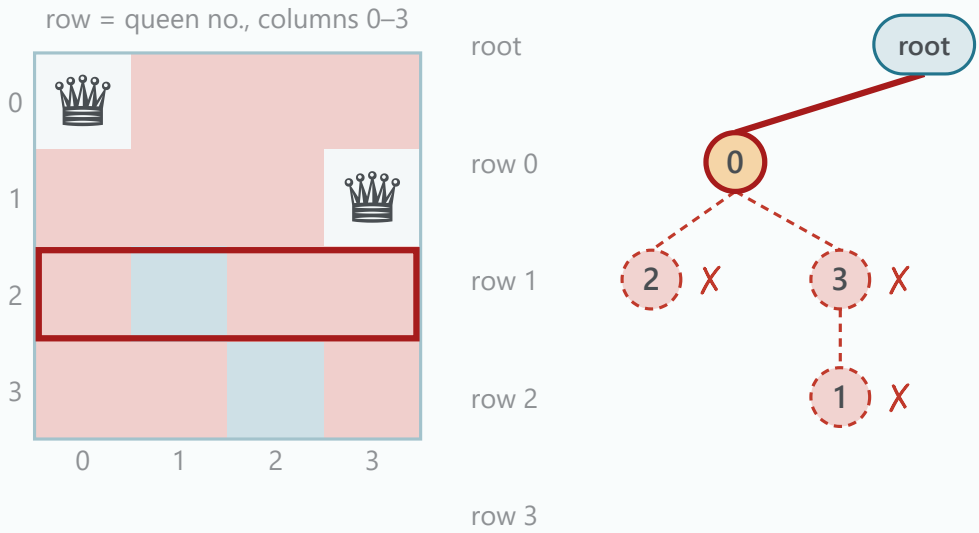


Row 3: columns 0, 1, 2, 3 are attacked → **no safe column** (dead end). **Backtrack**: remove the queen in row 2 (column 1) and try the next column in that row.

Step 7 / 13

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

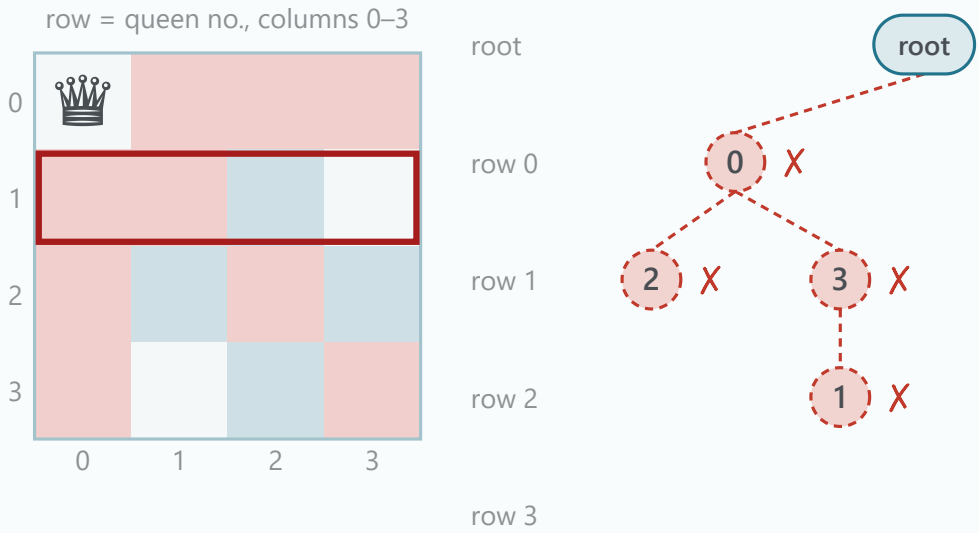
4 Queens: Board and State-Space Tree



Row 2: columns 2, 3 are attacked → **no safe column** (dead end). **Backtrack**: remove the queen in row 1 (column 3) and try the next column in that row.

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

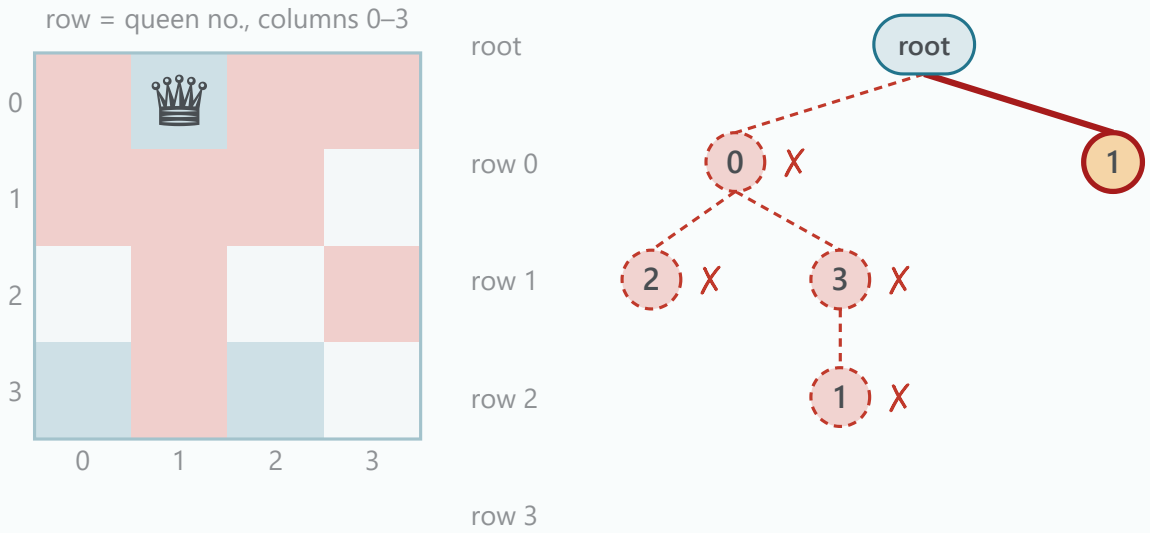
4 Queens: Board and State-Space Tree



Row 1 has no columns left to try. **Backtrack**: the queen in row 0 (column 0) is removed too.

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

4 Queens: Board and State-Space Tree

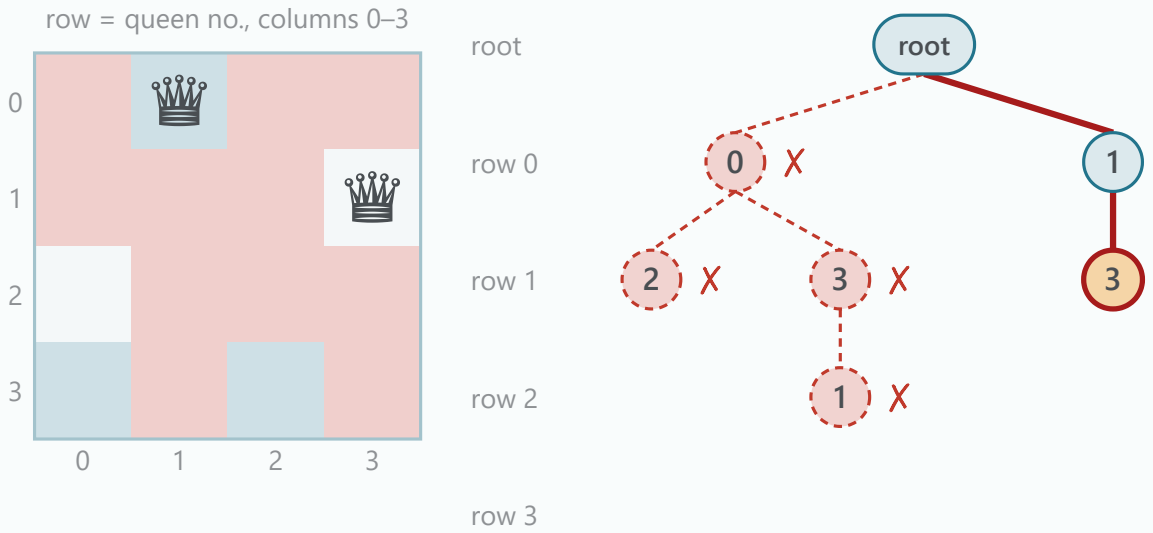


Row 0: **column 1 is safe** → place a queen and go down a level (row 1).

Step 10 / 13

Red square: attacked by the current queens · orange node: last placed · dashed X: abandoned branch · green: solution.

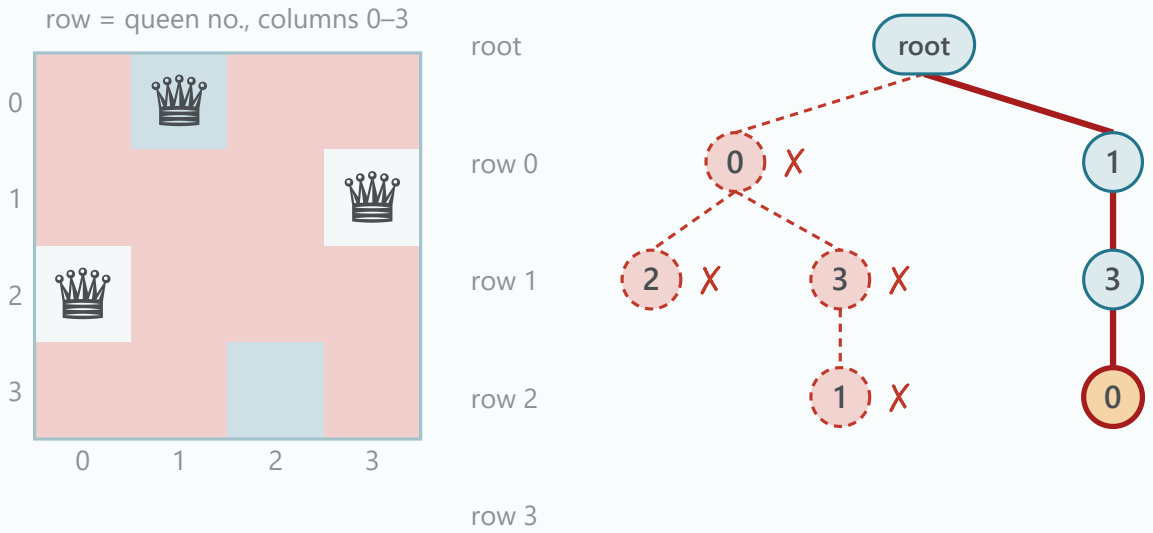
4 Queens: Board and State-Space Tree



Row 1: columns 0, 1, 2 are attacked; **column 3 is safe** → place a queen and go down a level (row 2).

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

4 Queens: Board and State-Space Tree

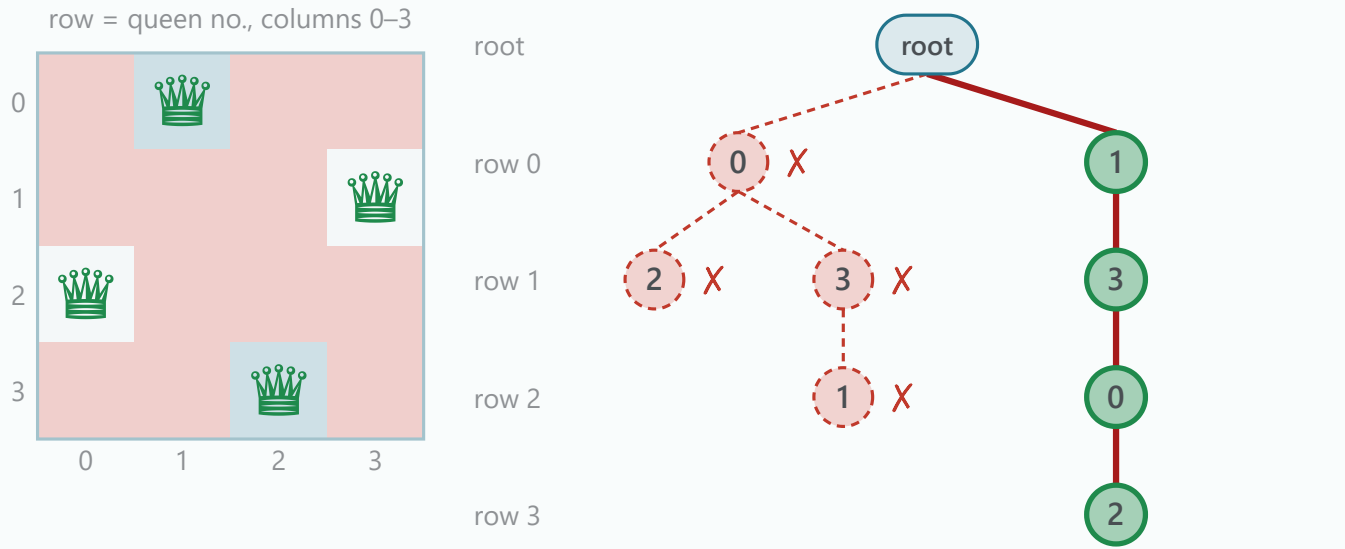


Row 2: **column 0 is safe** → place a queen and go down a level (row 3).

Step 12 / 13

Red square: attacked by the current queens · orange node: last placed · dashed X: abandoned branch · green: solution.

4 Queens: Board and State-Space Tree



Row 3: columns 0, 1 are attacked; **column 2 is safe** → the fourth queen is placed. `row == n` : **solution** found: columns [1, 3, 0, 2]. In total 8 placements, 4 backtracks.

Step 13 / 13

Red square: attacked by the current queens · orange node: last placed · dashed *X*: abandoned branch · green: solution.

n-Queens: Java Code

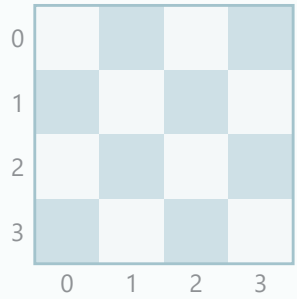
```java
boolean isSafe(int[][] b,
               int r, int c) {
  // same column
  for (int i = 0; i < r; i++)
    if (b[i][c] == 1) return false;
  // upper-left diagonal
  for (int i = r-1, j = c-1;
       i >= 0 && j >= 0; i--, j--)
    if (b[i][j] == 1) return false;
  // upper-right diagonal
  for (int i = r-1, j = c+1;
       i >= 0 && j < n; i--, j++)
    if (b[i][j] == 1) return false;
  return true;
}
```

```java
boolean solve(int[][] b, int row) {
  if (row == n)
    return true;
  for (int col = 0; col < n;
       col++) {
    if (isSafe(b, row, col)) {
      b[row][col] = 1; // place
      if (solve(b, row + 1))
        return true;
      b[row][col] = 0; // undo
    }
  }
  return false;
}
```

- `b[i][j] == 1`: a queen at (i, j); `n` is a field of the class. Queens exist only in the **rows above**, so `isSafe` looks in three directions: up, upper-left and upper-right.
- `solve` proceeds row by row; at the **first** solution it returns `true`, which propagates up the stack. `solve` visits O(n!) nodes; each node calls the O(n) `isSafe` n times → a rough bound of $O(n^2 \cdot n!)$.

solve(b, 0), n = 4: Code Trace

```
▶ boolean solve(int[][] b, int row) {
2   if (row == n) return true;
3   for (int col = 0; col < n; col++) {
4     if (isSafe(b, row, col)) {
5       b[row][col] = 1;
6       if (solve(b, row + 1)) return true;
7       b[row][col] = 0;  // undo
8     }
9   }
10  return false;
11 }
```



`solve(b, 0)` is called: the board is empty, `row` = 0.

Step 1 / 36

VARIABLES

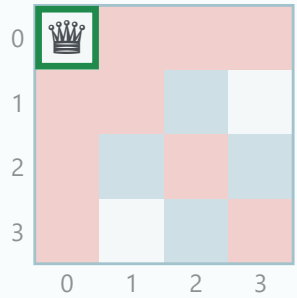
`row = 0` `col = –` `isSafe = –` `pos = [–,–,–,–]`

CALL STACK

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(0, 0) is safe → `b[0][0] = 1`, then `solve(b, 1)` is called.

Step 2 / 36

VARIABLES

`row = 0` `col = 0`

`isSafe = true`

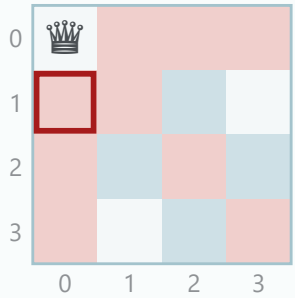
`pos = [0,-,-,-]`

CALL STACK

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 0) is attacked (same column: (0, 0)) → `isSafe` false, next column.

Step 3 / 36

VARIABLES

`row = 1` `col = 0` `isSafe = false` `pos = [0,-,-,-]`

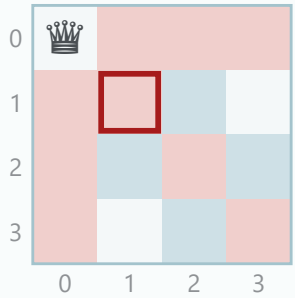
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 1) is attacked (diagonal: (0, 0)) → `isSafe` false, next column.

Step 4 / 36

VARIABLES

`row = 1` `col = 1`

`isSafe = false`

`pos = [0,-,-,-]`

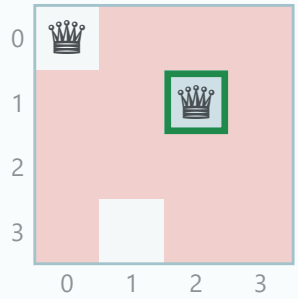
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 2) is safe → `b[1][2] = 1`, then `solve(b, 2)` is called.

Step 5 / 36

VARIABLES

`row = 1` `col = 2`

`isSafe = true`

`pos = [0,2,-,-]`

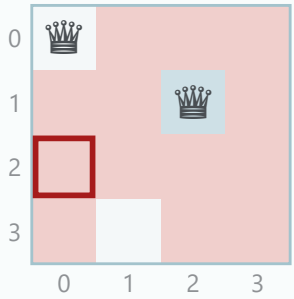
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 0) is attacked (same column: (0, 0)) → `isSafe` false, next column.

Step 6 / 36

VARIABLES

`row = 2` `col = 0`

`isSafe = false`

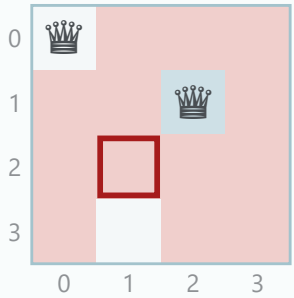
`pos = [0,2,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 1) is attacked (diagonal: (1, 2)) → `isSafe` false, next column.

Step 7 / 36

VARIABLES

`row = 2` `col = 1`

`isSafe = false`

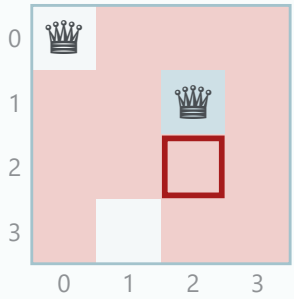
`pos = [0,2,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 2) is attacked (same column: (1, 2)) → `isSafe` false, next column.

Step 8 / 36

VARIABLES

`row = 2` `col = 2`

`isSafe = false`

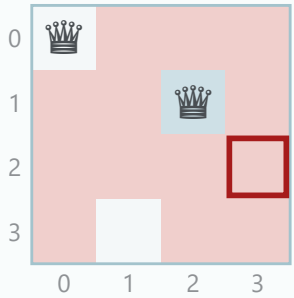
`pos = [0,2,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 3) is attacked (diagonal: (1, 2)) → `isSafe` false, next column.

Step 9 / 36

VARIABLES

`row = 2` `col = 3`

`isSafe = false`

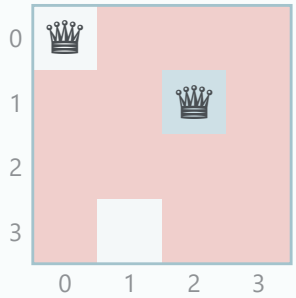
`pos = [0,2,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
▶    return false;
11 }
```



The loop for row 2 ends with no safe column → `return false`.

Step 10 / 36

VARIABLES

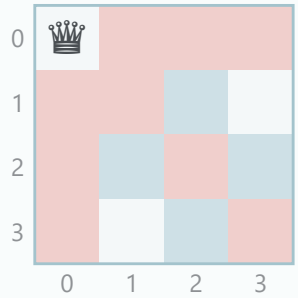
`row = 2` `col = 4` `isSafe = –` `pos = [0,2,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
▶        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



`solve(b, 2)` returned false → **undo**: `b[1][2] = 0` ; the loop continues with column 3.

Step 11 / 36

VARIABLES

`row = 1` `col = 2`

`isSafe = true`

`pos = [0,-,-,-]`

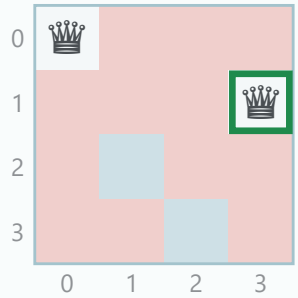
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 3) is safe → `b[1][3] = 1`, then `solve(b, 2)` is called.

Step 12 / 36

VARIABLES

`row = 1` `col = 3`

`isSafe = true`

`pos = [0,3,-,-]`

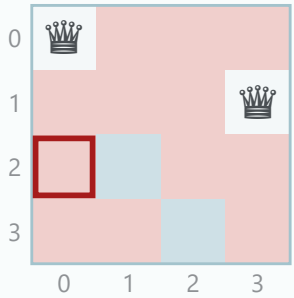
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 0) is attacked (same column: (0, 0)) → `isSafe` false, next column.

Step 13 / 36

VARIABLES

`row = 2` `col = 0`

`isSafe = false`

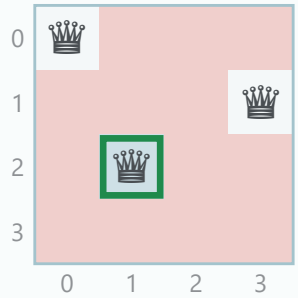
`pos = [0,3,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 1) is safe → `b[2][1] = 1`, then `solve(b, 3)` is called.

Step 14 / 36

VARIABLES

`row = 2` `col = 1`

`isSafe = true`

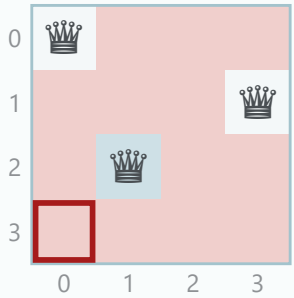
`pos = [0,3,1,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 0) is attacked (same column: (0, 0)) → `isSafe` false, next column.

Step 15 / 36

VARIABLES

`row = 3` `col = 0`

`isSafe = false`

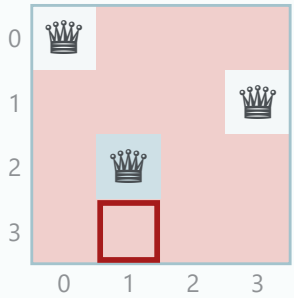
`pos = [0,3,1,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 1) is attacked (same column: (2, 1)) → `isSafe` false, next column.

Step 16 / 36

VARIABLES

`row = 3` `col = 1`

`isSafe = false`

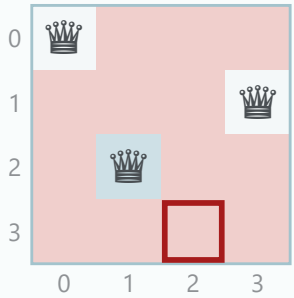
`pos = [0,3,1,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 2) is attacked (diagonal: (2, 1)) → `isSafe` false, next column.

Step 17 / 36

VARIABLES

`row = 3` `col = 2`

`isSafe = false`

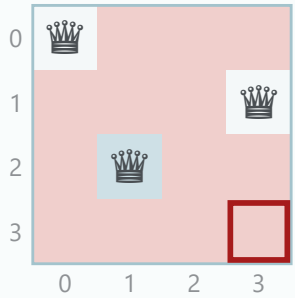
`pos = [0,3,1,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 3) is attacked (same column: (1, 3)) → `isSafe` false, next column.

Step 18 / 36

VARIABLES

`row = 3` `col = 3`

`isSafe = false`

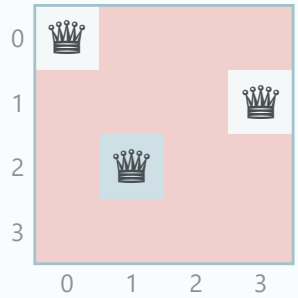
`pos = [0,3,1,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
▶    return false;
11 }
```



The loop for row 3 ends with no safe column → `return false`.

Step 19 / 36

VARIABLES

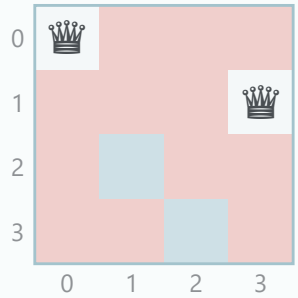
`row = 3` `col = 4` `isSafe = –` `pos = [0,3,1,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
▶        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



`solve(b, 3)` returned false → **undo**: `b[2][1] = 0` ; the loop continues with column 2.

Step 20 / 36

VARIABLES

`row = 2` `col = 1`

`isSafe = true`

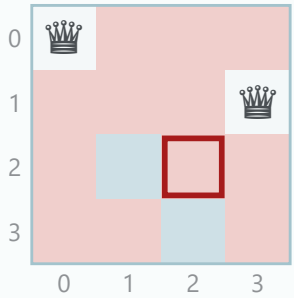
`pos = [0,3,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 2) is attacked (diagonal: (0, 0)) → `isSafe` false, next column.

Step 21 / 36

VARIABLES

`row = 2` `col = 2`

`isSafe = false`

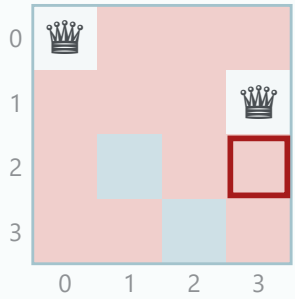
`pos = [0,3,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 3) is attacked (same column: (1, 3)) → `isSafe` false, next column.

Step 22 / 36

VARIABLES

`row = 2` `col = 3`

`isSafe = false`

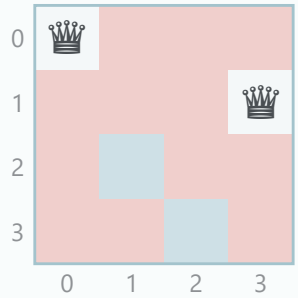
`pos = [0,3,-,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
▶    return false;
11 }
```



The loop for row 2 ends with no safe column → `return false`.

Step 23 / 36

VARIABLES

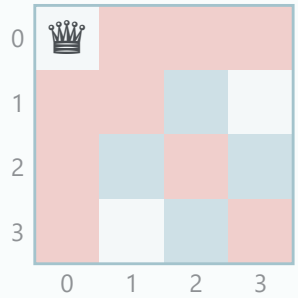
`row = 2` `col = 4` `isSafe = –` `pos = [0,3,–,–]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
▶        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



`solve(b, 2)` returned false → **undo**: `b[1][3] = 0` ; column 4 = n, so the loop ends.

Step 24 / 36

VARIABLES

`row = 1` `col = 3`

`isSafe = true`

`pos = [0,-,-,-]`

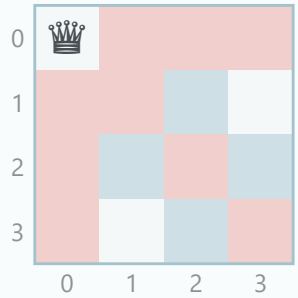
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
▶    return false;
11 }
```



The loop for row 1 ends with no safe column → `return false`.

Step 25 / 36

VARIABLES

`row = 1` `col = 4` `isSafe = –` `pos = [0,–,–,–]`

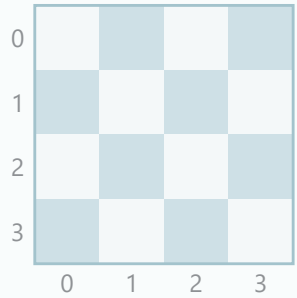
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
▶        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



`solve(b, 1)` returned false → **undo**: `b[0][0] = 0` ; the loop continues with column 1.

Step 26 / 36

VARIABLES

`row = 0` `col = 0`

`isSafe = true`

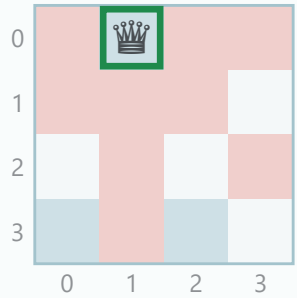
`pos = [-,-,-,-]`

CALL STACK

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(0, 1) is safe → `b[0][1] = 1`, then `solve(b, 1)` is called.

Step 27 / 36

VARIABLES

`row = 0` `col = 1`

`isSafe = true`

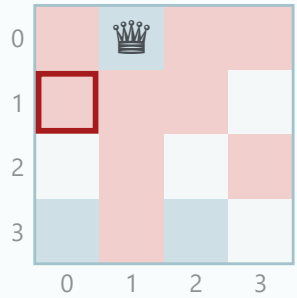
`pos = [1,-,-,-]`

CALL STACK

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 0) is attacked (diagonal: (0, 1)) → `isSafe` false, next column.

Step 28 / 36

VARIABLES

`row = 1` `col = 0` `isSafe = false` `pos = [1,-,-,-]`

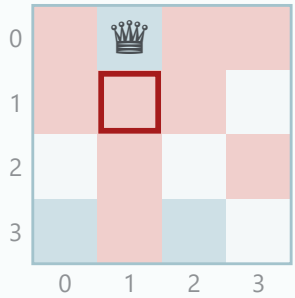
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 1) is attacked (same column: (0, 1)) → `isSafe` false, next column.

Step 29 / 36

VARIABLES

`row = 1` `col = 1`

`isSafe = false`

`pos = [1,-,-,-]`

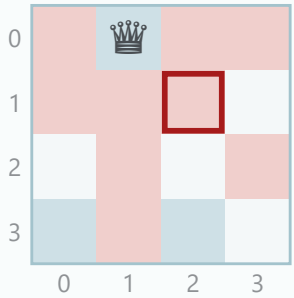
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 2) is attacked (diagonal: (0, 1)) → `isSafe` false, next column.

Step 30 / 36

VARIABLES

`row = 1` `col = 2`

`isSafe = false`

`pos = [1,-,-,-]`

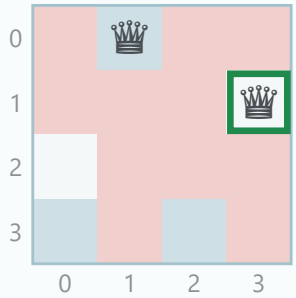
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(1, 3) is safe → `b[1][3] = 1`, then `solve(b, 2)` is called.

Step 31 / 36

VARIABLES

`row = 1` `col = 3`

`isSafe = true`

`pos = [1,3,-,-]`

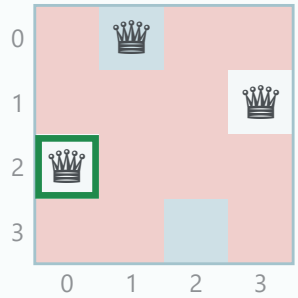
CALL STACK

`solve(b, 1)`

`solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(2, 0) is safe → `b[2][0] = 1`, then `solve(b, 3)` is called.

Step 32 / 36

VARIABLES

`row = 2` `col = 0`

`isSafe = true`

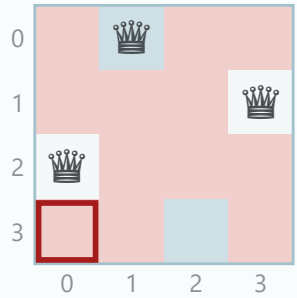
`pos = [1,3,0,-]`

CALL STACK

- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 0) is attacked (same column: (2, 0)) → `isSafe` false, next column.

Step 33 / 36

VARIABLES

`row = 3` `col = 0`

`isSafe = false`

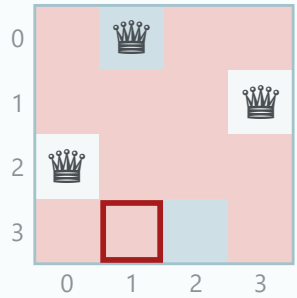
`pos = [1,3,0,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
▶    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 1) is attacked (same column: (0, 1)) → `isSafe` false, next column.

Step 34 / 36

VARIABLES

`row = 3` `col = 1`

`isSafe = false`

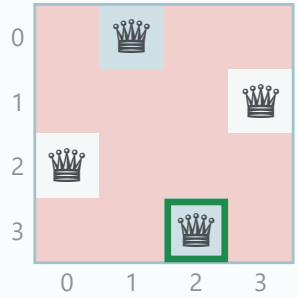
`pos = [1,3,0,-]`

CALL STACK

- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
2    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
▶      if (isSafe(b, row, col)) {
▶        b[row][col] = 1;
▶        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



(3, 2) is safe → `b[3][2] = 1`, then `solve(b, 4)` is called.

Step 35 / 36

VARIABLES

`row = 3` `col = 2`

`isSafe = true`

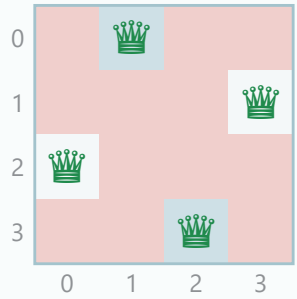
`pos = [1,3,0,2]`

CALL STACK

- **solve(b, 3)**
- solve(b, 2)
- solve(b, 1)
- solve(b, 0)

solve(b, 0), n = 4: Code Trace

```
1  boolean solve(int[][] b, int row) {
▶    if (row == n) return true;
3    for (int col = 0; col < n; col++) {
4      if (isSafe(b, row, col)) {
5        b[row][col] = 1;
6        if (solve(b, row + 1)) return true;
7        b[row][col] = 0;  // undo
8      }
9    }
10   return false;
11 }
```



`row == n` → `true`. The true value propagates up the stack: solution [1, 3, 0, 2].

Step 36 / 36

VARIABLES

`row = 4` `col = –`

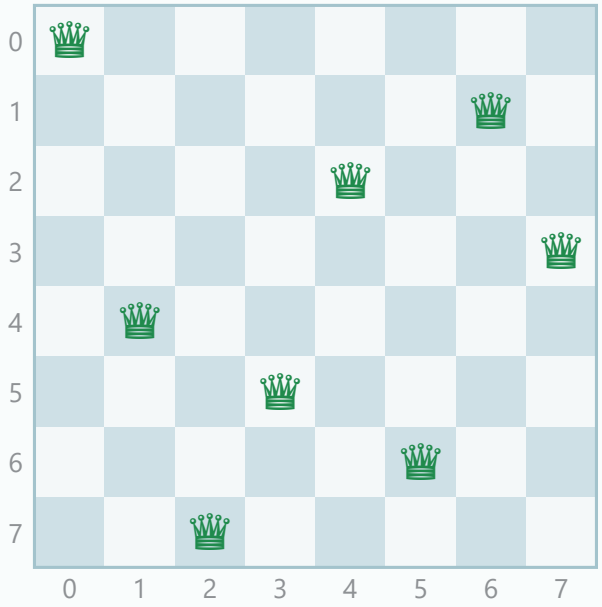
`isSafe = –`

`pos = [1,3,0,2]`

CALL STACK

- `solve(b, 4)`
- `solve(b, 3)`
- `solve(b, 2)`
- `solve(b, 1)`
- `solve(b, 0)`

8 Queens: The Algorithm's First Solution



Columns for rows 0–7: [0, 4, 7, 5, 2, 6, 1, 3]

- `solve(b, 0)` (n = 8) tries columns in increasing order, so it always finds the **lexicographically first** solution.
- Reaching it takes **876** calls to `isSafe`, **113** queen placements and **105** backtracks.
- The placement (a1, b7, c4, d6, e8, f2, g5, h3) is also a solution; all 92 solutions are valid.
- To find all solutions, record the solution and return `false` (keep searching) when `row == n`.

n-Queens: Solutions and Placements

n	Solutions	n^n (brute force)	Placements (all)	To first solution
1	1	1	1	1
2	0	4	2	—
3	0	27	5	—
4	2	256	16	8
5	10	3,125	53	5
6	4	46,656	152	31
7	40	823,543	551	9
8	92	16,777,216	2056	113
9	352	387,420,489	8393	41
10	724	10,000,000,000	35538	102

- No solution for n = 2 and 3: the whole tree is visited and the result is `false`.
- Placements: queens placed safely (non-root tree nodes).
- Finding all costs far less than n^n, yet still grows roughly like n!.
- Work to the first solution is irregular: 31 placements for n = 6, only 9 for n = 7.

A Faster Safety Test: O(1)

```
boolean[] cols = new boolean[n];
boolean[] d1 = new boolean[2*n-1]; // r+c
boolean[] d2 = new boolean[2*n-1]; // r-c+n-1

void solve(int r) {
  if (r == n) { count++; return; }
  for (int c = 0; c < n; c++) {
    int a = r + c, b = r - c + n - 1;
    if (cols[c] || d1[a] || d2[b])
      continue;
    cols[c] = d1[a] = d2[b] = true;
    solve(r + 1);
    cols[c] = d1[a] = d2[b] = false;
  }
}
```

k1: r + c

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

k2: r − c + n − 1

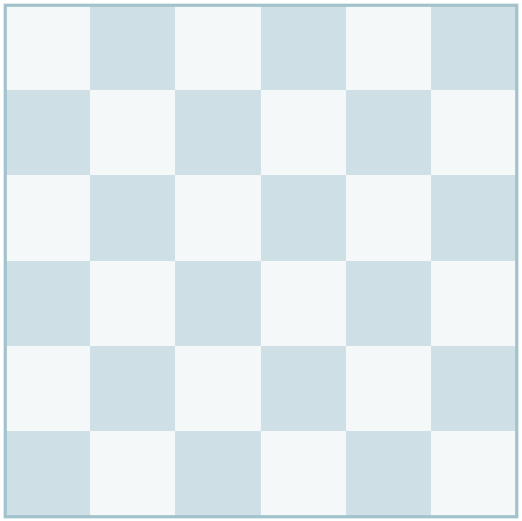
3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

- Cells on the same **anti-diagonal** have a constant r + c (0 … 2n − 2).
- Cells on the same **diagonal** have a constant r − c; adding n − 1 makes the index start at 0.

Each queen sets three flags: column, anti-diagonal, diagonal. `isSafe` becomes **O(1)** instead of O(n), and the board array is no longer needed (just `pos[row] = col`).

Try It: Backtracking for n-Queens

n 6 step play next solution reset



Empty board (n = 6). Queens are placed row by row; in each row columns are tried left to right. · placements: 0, backtracks: 0, solutions: 0

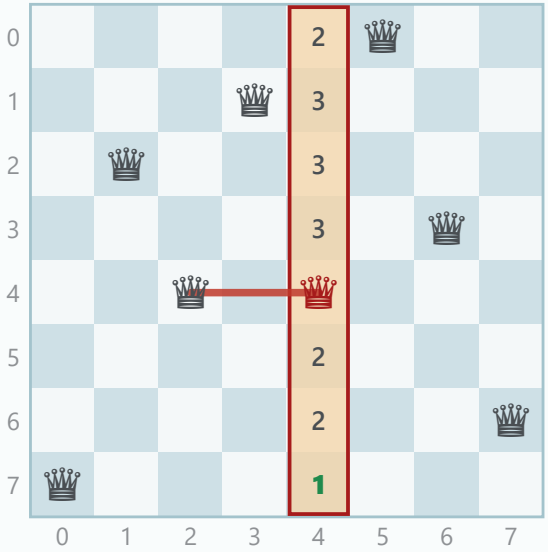
Red square: attacked · green frame: safe try · red frame: rejected try · green queens: solution.

Min-Conflicts Heuristic

- A different idea: first place **all the queens** (one per column), then repair conflicts with **local changes**.
- The number in each cell: **how many queens** a queen moved to that row **would conflict with** (row + diagonals).
- **Start:** column by column, pick the row with the fewest conflicts (ties broken at random).
- **Repair:** pick a conflicted queen and move it, within its column, to the row with the fewest conflicts. Stop when no conflicts remain.
- This is a **local search** method: the state is a complete but possibly invalid placement, and there is no backtracking stack. It is not complete: it can wander on a plateau, needs a step limit, finds only one solution and cannot prove that none exists. On the other hand, even for n = 1,000,000 it solves the problem in a few dozen steps on average.

```
minConflicts(n, maxSteps):
  // start: greedy placement
  for k = 0 .. n-1:
    pos[k] = row with fewest conflicts
  for step = 1 .. maxSteps:
    if no conflicts: return pos
    k = random conflicted column
    pos[k] = argmin_r conflicts(k, r)
             // ties broken at random
  return failure
```

Min-Conflicts: Repair Steps

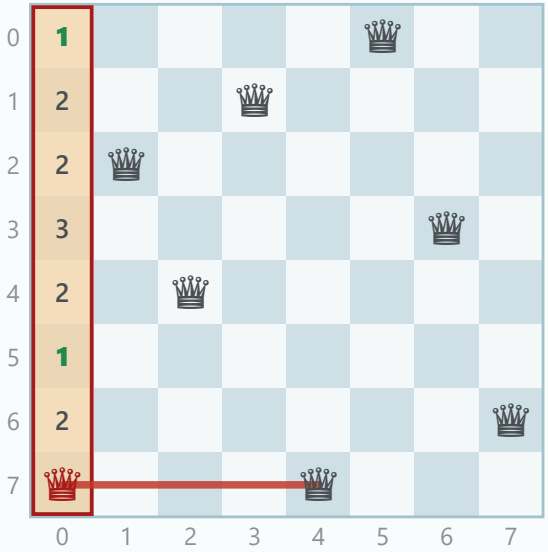


Greedy start [7, 2, 4, 1, 4, 0, 3, 6]: only columns 2 and 4 (row 4) conflict. Column 4 is chosen; the minimum is **1** (row 7) → the queen moves to **row 7**.

Step 1 / 9

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

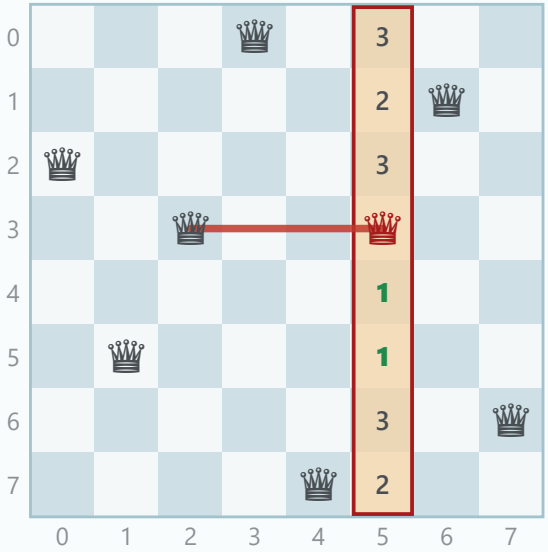
Min-Conflicts: Repair Steps



The move created a new conflict: columns 0 and 4 are now in the same row (7). Column 0's options: minimum 1 (rows 0, 5). Conflicting pairs in total: 1.

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

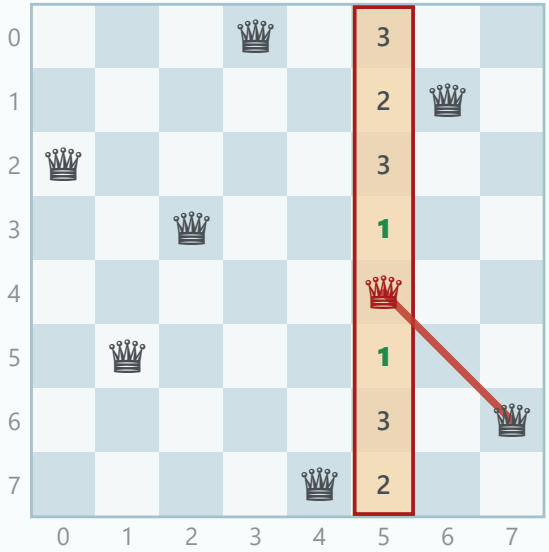
Min-Conflicts: Repair Steps



A few repair steps later: the queen in column 5 is in row 3, conflicting with column 2. The minimum is 1 (rows 4, 5) → row 4.

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

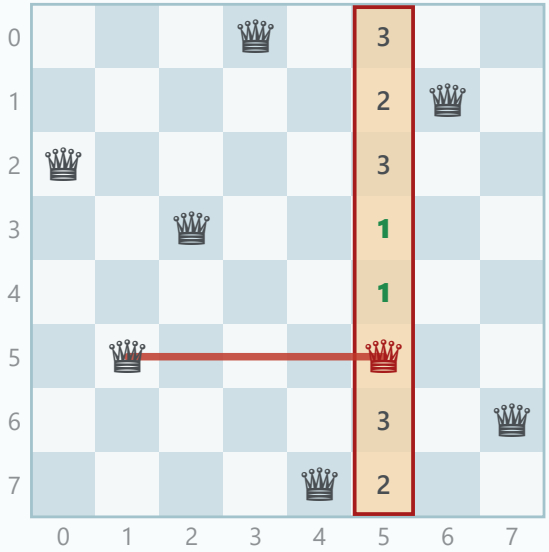
Min-Conflicts: Repair Steps



In row 4 there is now a diagonal conflict with (6, 7) (1). Rows 3, 4, 5 all have 1 conflict: a **plateau**. A random choice moves it to row 5.

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

Min-Conflicts: Repair Steps

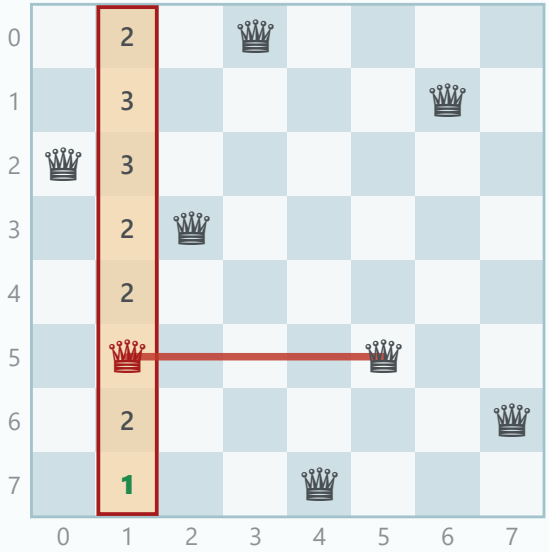


In row 5 it shares a row with column 1. Column 5 goes back and forth on the plateau 3 ↔ 4 ↔ 5 a few times; thanks to random choice, another queen is eventually picked.

Step 5 / 9

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

Min-Conflicts: Repair Steps

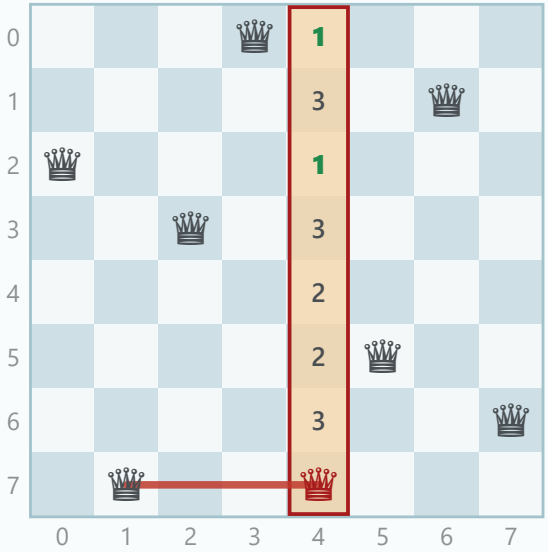


Column 1 is chosen (row 5, 1 conflict). The minimum is **1** (row 7) → row 7.

Step 6 / 9

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

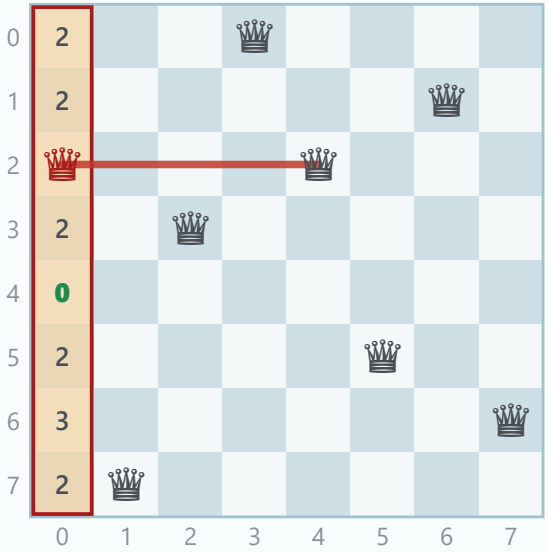
Min-Conflicts: Repair Steps



Now columns 1 and 4 conflict in row 7. Column 4's minimum is 1 (rows 0, 2) → row 2.

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

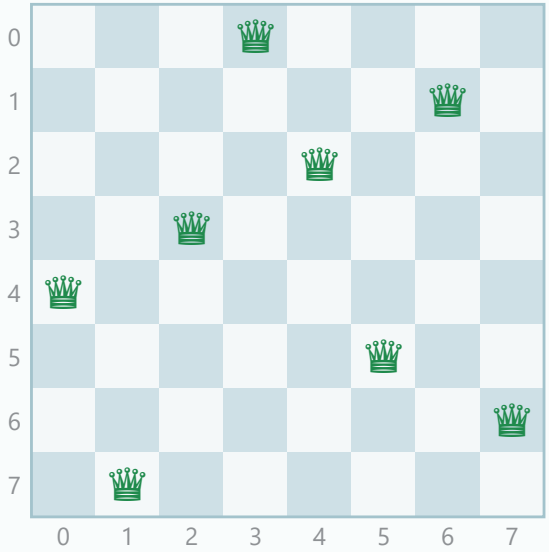
Min-Conflicts: Repair Steps



Columns 0 and 4 conflict in row 2. In column 0, row 4 has **0 conflicts** → the queen moves there.

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

Min-Conflicts: Repair Steps



No conflicted queens remain: **solution** [4, 7, 3, 0, 2, 5, 1, 6]. The algorithm placed all 8 queens up front, then removed the conflicts using only **local changes**.

Step 9 / 9

The numbers in the highlighted column show how many queens the red queen would conflict with if moved to that row (the queen's own row is left blank).

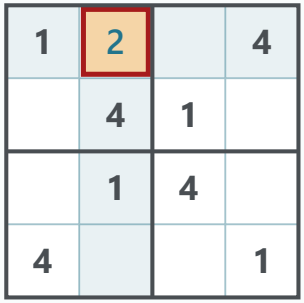
Sudoku

- The goal of Sudoku is to use the digits 1 to 9 **exactly once** in **every row**, **every column** and **every 3×3 box** of a 9×9 grid.
- Some cells are given up front (**clues**); the rest must be filled in.
- It is a **constraint satisfaction problem**: the variables are the empty cells, the domain is {1, ..., 9}, and the constraints say "all different within a unit".

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku: Backtracking Algorithm

1. Assign a digit to an **empty** cell.
2. If the digit is **valid** (not in its row, column or box), **move on** to the next empty cell.
3. If not, try the **next digit**.
4. If **every digit** has been tried in a cell, **go back to the previous cell** (backtrack) and try a different digit there.
5. Repeat until every cell is filled.



For (0, 1):

1 ✗ in row

2 ✓ write, move on

at a dead end 2 is erased, 3 is tried

This is the general template applied directly: **partial solution** = filled cells, **candidate** = 1–9, **constraint** = `isValid`, **goal** = no empty cells left.

Sudoku: Java Code

```java
boolean solve(int[][] g) {
  for (int r = 0; r < 9; r++)
    for (int c = 0; c < 9; c++)
      if (g[r][c] == 0) {
        for (int d = 1; d <= 9; d++)
          if (isValid(g, r, c, d)) {
            g[r][c] = d;
            if (solve(g)) return true;
            g[r][c] = 0;   // undo
          }
        return false;  // no digit fits
      }
  return true;  // no empty cell
}
```

```java
boolean isValid(int[][] g,
    int r, int c, int d) {
  for (int i = 0; i < 9; i++)
    if (g[r][i] == d || g[i][c] == d)
      return false;
  int br = r / 3 * 3;
  int bc = c / 3 * 3;
  for (int i = br; i < br + 3; i++)
    for (int j = bc; j < bc + 3; j++)
      if (g[i][j] == d)
        return false;
  return true;
}
```

Each call to `solve` scans from the start for the first empty cell (O(81)); if no digit fits a cell, `return false` triggers the backtrack.

Sudoku: Backtracking Step by Step

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Puzzle: 30 clues, **51 empty cells**. Empty cells are filled row by row, left to right; in each cell 1–9 are tried in order.

Step 1 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red X: cell with no digits left (dead end).

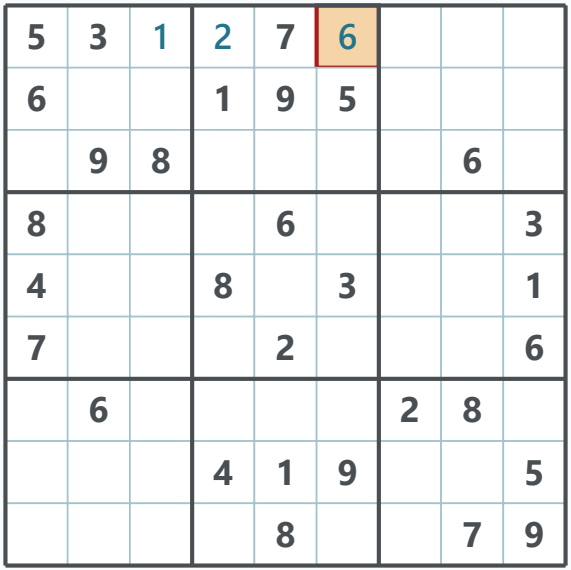
Sudoku: Backtracking Step by Step

5	3	1	2	7	4	8	9	X
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

The first five empty cells get 1, 2, 4, 8, 9. For (0, 8) **no digit is valid**: only 6 is missing from the row, and it is already in the column → dead end. (so far 5 placements, 0 undos)

Black: clue · blue: digit written by the algorithm · orange: last cell written · red X: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step



Backtrack: the 9 at (0, 7) is erased, (0, 6) is tried with 9, another dead end; backtracking reaches (0, 5), where **6** is tried instead of 4. (so far 7 placements, 4 undos)

Step 3 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red X: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	1	2	7	6	4	9	8
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Row 0 is complete for the first time: 5 3 1 2 7 6 4 9 8. The row is valid **on its own**, but we do not yet know whether it leads to a solution. (so far 10 placements, 4 undos)

Step 4 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red *X*: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	1	2	7	6	4	9	X
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Every attempt in row 1 hit a dead end. Backtracking reaches **row 0**: the 8 at (0, 8) is erased. An early decision can turn out wrong much later. (so far 31 placements, 26 undos)

Step 5 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red X: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	1	6	7	8	4	9	2
6	2	4	1	9	5	7	3	8
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Row 1 is complete for the first time. (so far 229 placements, 218 undos)

Step 6 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	2	6	7	8	1	9	4
6	4	7	1	9	5	3	2	8
1	9	8	2	3	4	5	6	7
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Row 2 is complete for the first time; the first three rows now match the final solution. (so far 377 placements, 360 undos)

Step 7 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red *X*: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	2	6	7	8	1	9	4
6	4	7	1	9	5	3	2	8
1	9	8	2	3	4	5	6	7
8	2	1	5	6	7	9	4	3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Row 3 is complete for the first time. (so far 395 placements, 372 undos)

Black: clue · blue: digit written by the algorithm · orange: last cell written · red X: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	2	6	7	8	9	1	4
6	4	7	1	9	5	8	3	2
1	9	8	2	3	4	5	6	7
8	1	5	9	6	7	4	2	3
4	2	6	8	5	3	7	9	1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Row 4 is complete for the first time. (so far 1520 placements, 1492 undos)

Step 9 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
	6					2	8	
			4	1	9			5
				8			7	9

Row 5 is complete for the first time. (so far 4185 placements, 4151 undos)

Step 10 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
			4	1	9			5
				8			7	9

Row 6 is complete for the first time. (so far 4197 placements, 4157 undos)

Step 11 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
				8			7	9

Row 7 is complete for the first time. (so far 4202 placements, 4157 undos)

Step 12 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Backtracking Step by Step

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Solution found: all 51 empty cells are filled and every constraint holds. In total **4208 placements** and **4157 undos**.

Step 13 / 13

Black: clue · blue: digit written by the algorithm · orange: last cell written · red ✗: cell with no digits left (dead end).

Sudoku: Solution and Cost

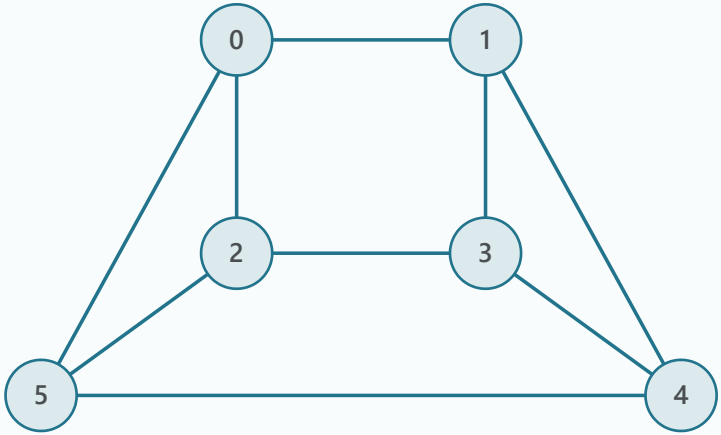
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Strategy	Placements	Undos
Row by row, first empty cell	4208	4157
MRV: cell with fewest candidates	51	0

- Brute force: $9^{51} \approx 4.6 \cdot 10^{48}$ assignments for 51 empty cells. Pruning brings this down to a few thousand.
- **MRV** (minimum remaining values): at each step, fill the cell with the **fewest candidates left**. In this puzzle there is always a cell with a single candidate: no backtracking at all.

Graph Coloring (m-Coloring)

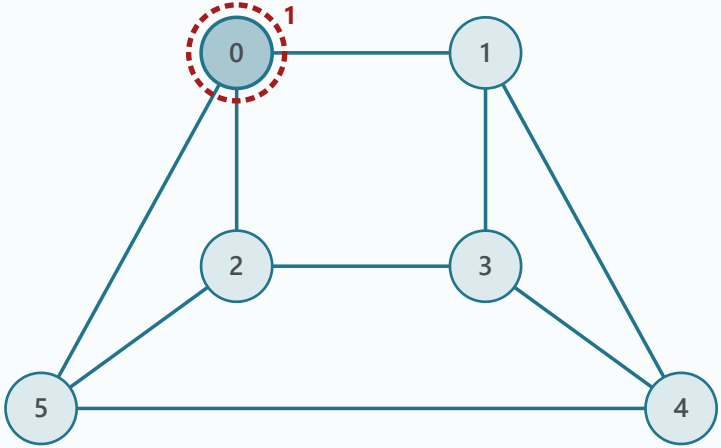
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



A six-vertex graph, **m = 3** colors (1 blue, 2 green, 3 orange). Vertices are colored in the order 0, 1, 2, …; at each vertex colors 1, 2, 3 are tried in order.

Graph Coloring (m-Coloring)

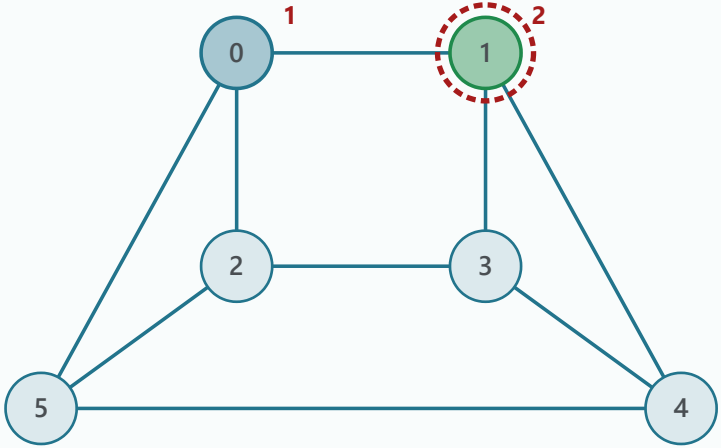
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 0: **color 1 (blue)** fits → assigned.

Graph Coloring (m-Coloring)

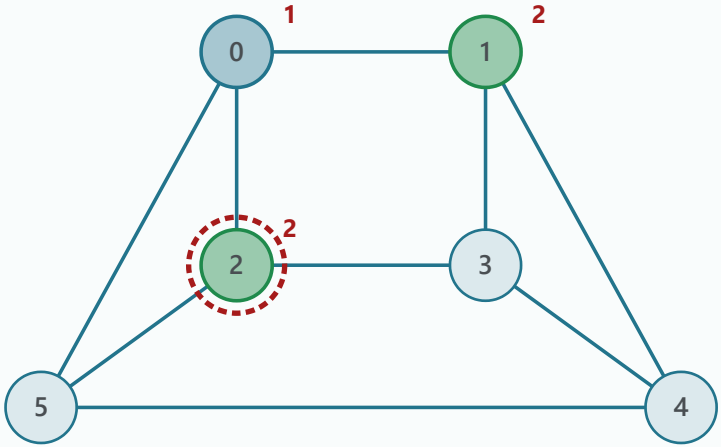
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 1: color 1 is on a neighbor; **color 2 (green)** fits → assigned.

Graph Coloring (m-Coloring)

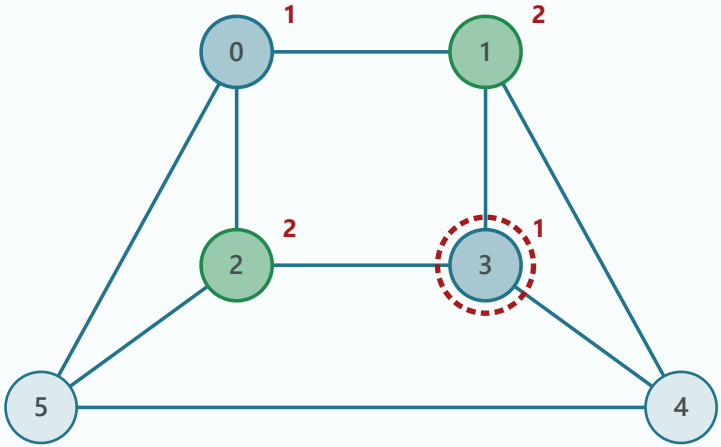
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 2: color 1 is on a neighbor; **color 2 (green)** fits → assigned.

Graph Coloring (m-Coloring)

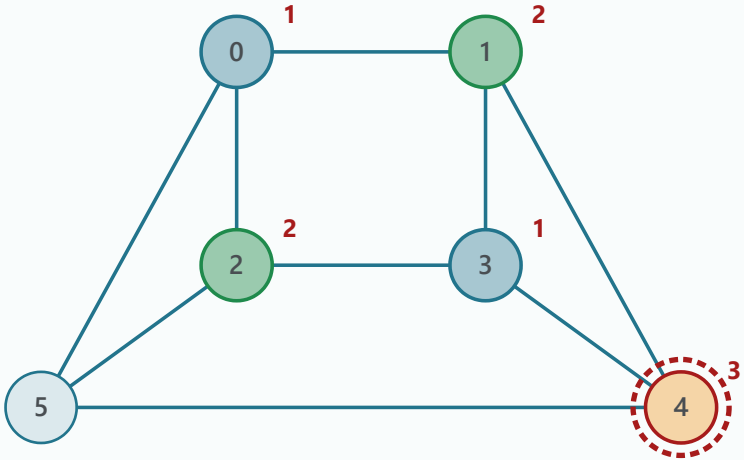
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 3: **color 1 (blue)** fits → assigned.

Graph Coloring (m-Coloring)

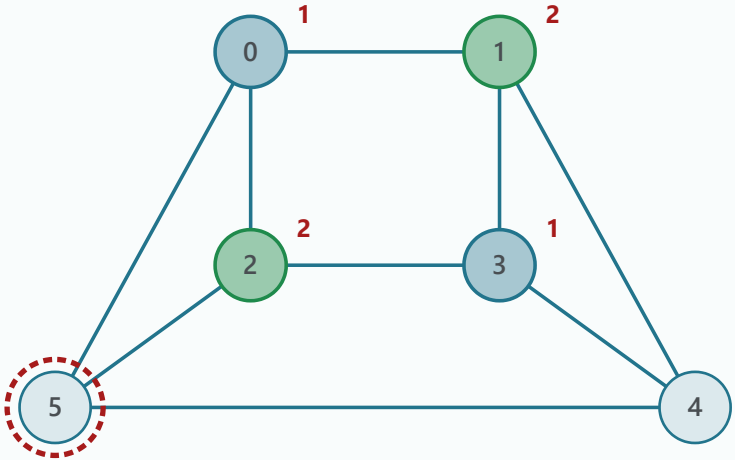
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 4: colors 1, 2 are on neighbors; **color 3 (orange)** fits → assigned.

Graph Coloring (m-Coloring)

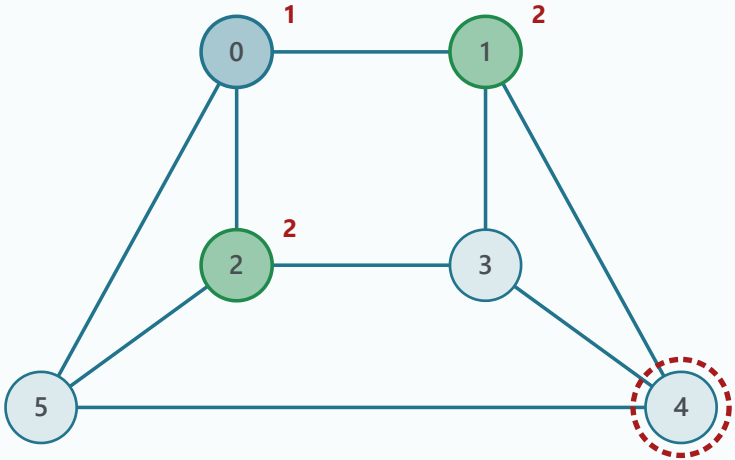
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 5: colors 1, 2, 3 are all used by neighbors → **no color fits** (dead end). **Backtrack**: vertex 4's color (3) is erased; the next color will be tried.

Graph Coloring (m-Coloring)

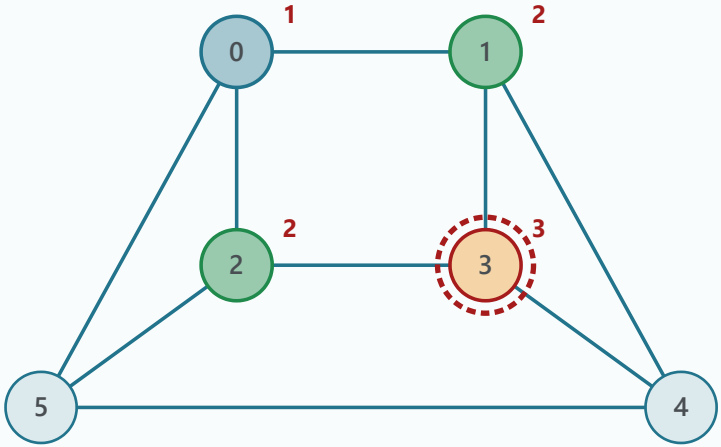
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 4: no colors left to try (dead end). **Backtrack**: vertex 3's color (1) is erased; the next color will be tried.

Graph Coloring (m-Coloring)

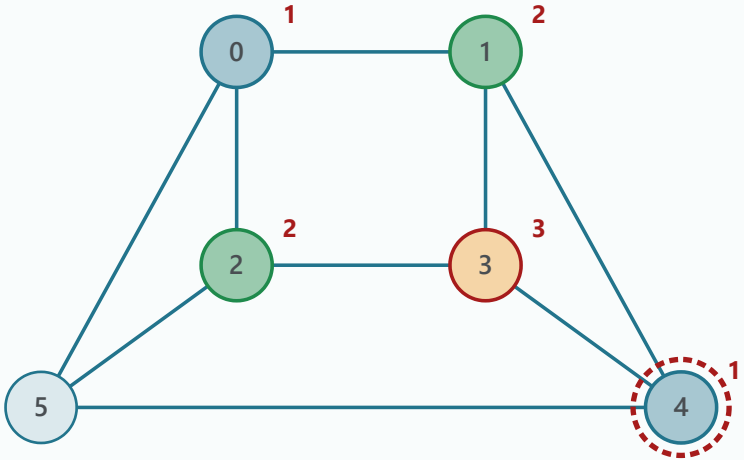
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 3: color 2 is on a neighbor; **color 3 (orange)** fits → assigned.

Graph Coloring (m-Coloring)

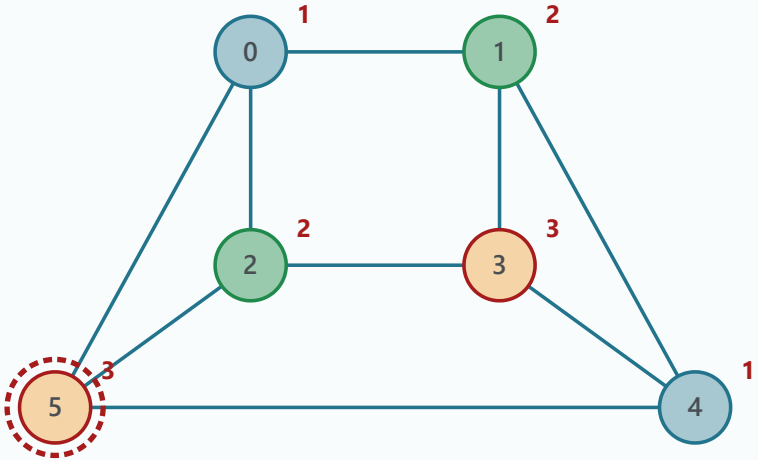
Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



Vertex 4: **color 1 (blue)** fits → assigned.

Graph Coloring (m-Coloring)

Assign at most **m** colors to the vertices so that no two adjacent vertices share a color. Partial solution: the colors of the first k vertices; constraint: no neighbor with the same color.



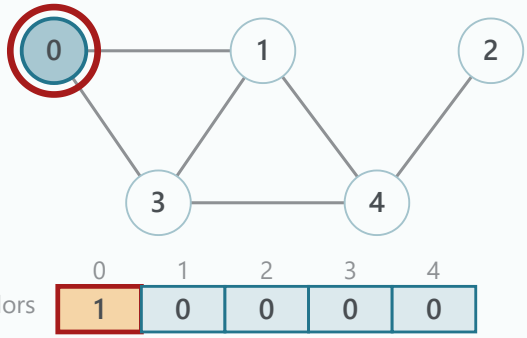
Vertex 5: colors 1, 2 are on neighbors; **color 3 (orange)** fits → assigned. All six vertices are colored: a **valid 3-coloring** has been found.

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;            // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                  // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                  // undo
8      }
9    }
10   return false;                       // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```

CALL STACK

- `color(0)`
- `main()`



Five vertices, m = 3 colors (1 blue, 2 green, 3 orange); vertices are tried in the order 0, 1, 2, …, colors in the order 1, 2, 3. Vertex 0: **color 1** is assigned.

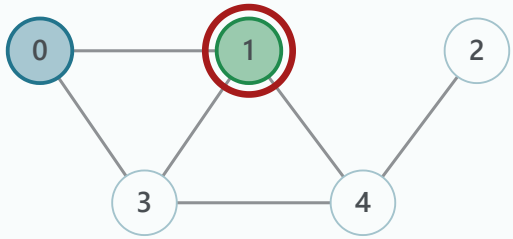
Step 1 / 9

VARIABLES

`v = 0` `c = 1` `colors = [1,0,0,0,0]`

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;            // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                  // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                  // undo
8      }
9    }
10   return false;                       // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```



Color 1 is on a neighbor; vertex 1: **color 2 (green)** fits → assigned.

Step 2 / 9

CALL STACK

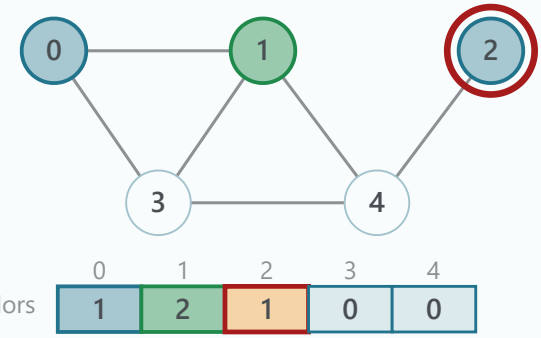
- color(1)
- color(0)
- main()

VARIABLES

v = 1 | c = 2 | colors = [1,2,0,0,0]

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;            // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                  // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                  // undo
8      }
9    }
10   return false;                       // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```



Vertex 2: **color 1 (blue)** fits → assigned.

CALL STACK

- color(2)
- color(1)
- color(0)
- main()

Step 3 / 9

VARIABLES

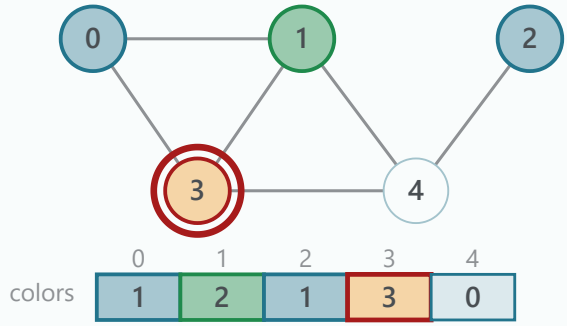
v = 2 | c = 1 | colors = [1,2,1,0,0]

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;            // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                  // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                  // undo
8      }
9    }
10   return false;                       // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```

CALL STACK

- color(3)
- color(2)
- color(1)
- color(0)
- main()



Colors 1, 2 are on neighbors; vertex 3: **color 3 (orange)** fits → assigned.

Step 4 / 9

VARIABLES

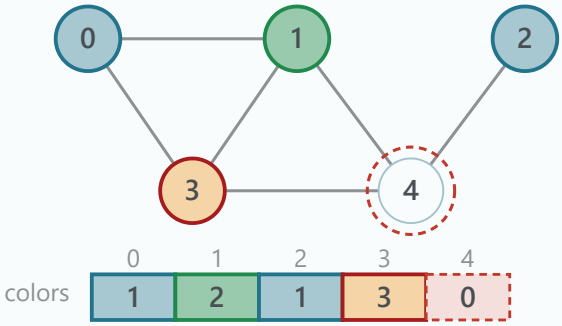
`v = 3` `c = 3` `colors = [1,2,1,3,0]`

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;          // done
3    for (int c = 1; c <= m; c++) {
4      if (isSafe(v, c)) {
5        colors[v] = c;                 // choose
6        if (color(v + 1)) return true; // explore
▶        colors[v] = 0;                 // undo
8      }
9    }
▶    return false;                     // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```

CALL STACK

- color(3)
- color(2)
- color(1)
- color(0)
- main()



Vertex 4: colors 1, 2, 3 are all on its neighbors → **dead end**, `return false`. Back up: colors[3] = 0; the next color for vertex 3 will be tried.

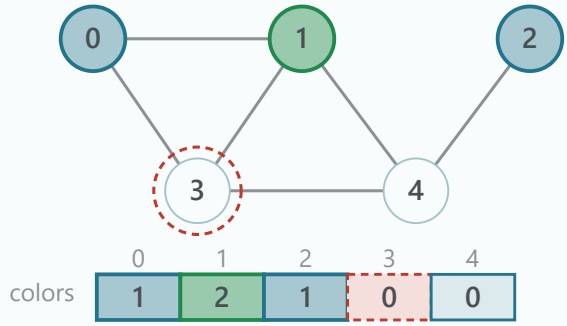
Step 5 / 9

VARIABLES

`v = 3` `c = 3` `colors = [1,2,1,0,0]`

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;           // done
3    for (int c = 1; c <= m; c++) {
4      if (isSafe(v, c)) {
5        colors[v] = c;                  // choose
6        if (color(v + 1)) return true; // explore
▶        colors[v] = 0;                  // undo
8      }
9    }
▶    return false;                      // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```



Vertex 3: no colors left to try → **dead end**. Back up: colors[2] = 0 (color 1 is erased).

Step 6 / 9

CALL STACK

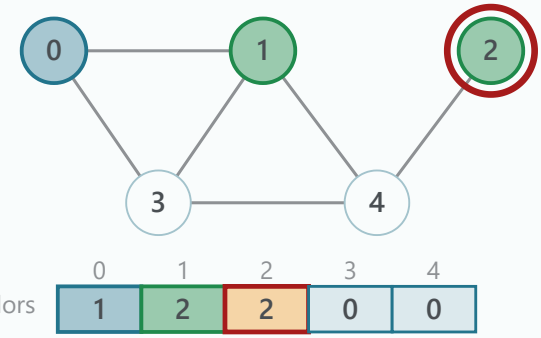
- color(2)
- color(1)
- color(0)
- main()

VARIABLES

v = 2 | c = 1 | colors = [1,2,0,0,0]

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;            // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                  // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                  // undo
8      }
9    }
10   return false;                       // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```



Vertex 2: **color 2 (green)** fits → assigned.

CALL STACK

- color(2)
- color(1)
- color(0)
- main()

Step 7 / 9

VARIABLES

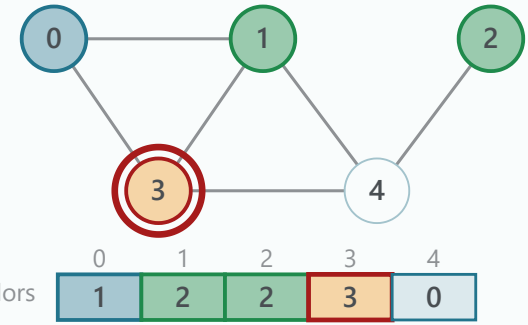
v = 2 | c = 2 | colors = [1,2,2,0,0]

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
2    if (v == n) return true;           // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                 // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                 // undo
8      }
9    }
10   return false;                      // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```

CALL STACK

- color(3)
- color(2)
- color(1)
- color(0)
- main()



Colors 1, 2 are on neighbors; vertex 3: **color 3 (orange)** fits → assigned.

Step 8 / 9

VARIABLES

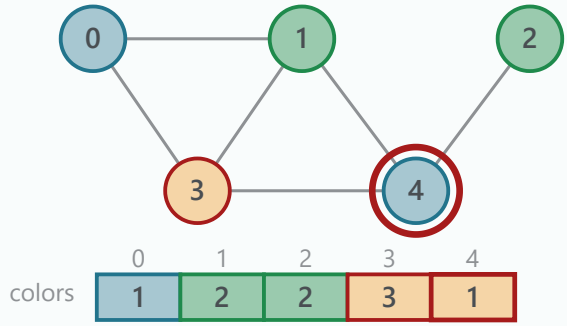
`v = 3` `c = 3` `colors = [1,2,2,3,0]`

color(0) with m = 3: Code Trace

```
1  boolean color(int v) {
▶    if (v == n) return true;          // done
3    for (int c = 1; c <= m; c++) {
▶      if (isSafe(v, c)) {
▶        colors[v] = c;                // choose
▶        if (color(v + 1)) return true; // explore
7        colors[v] = 0;                // undo
8      }
9    }
10   return false;                     // dead end
11 }
12 // isSafe: true if v has no neighbor with color c
```

CALL STACK

- color(4)
- color(3)
- color(2)
- color(1)
- color(0)
- main()



Vertex 4: **color 1 (blue)** fits → assigned. v = 5 = n: every vertex is colored; the **valid 3-coloring** [1, 2, 2, 3, 1] has been found.

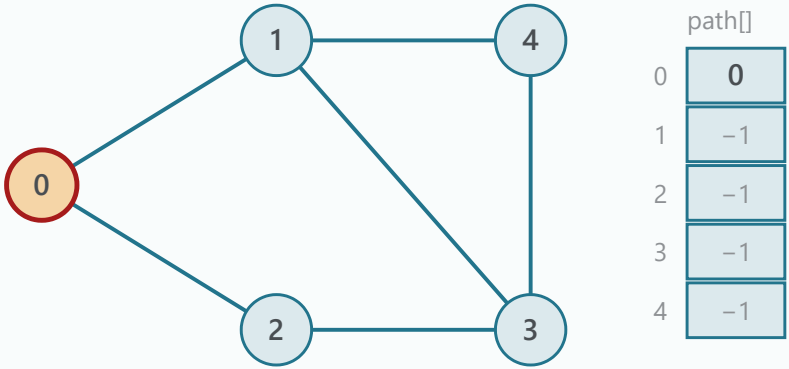
Step 9 / 9

VARIABLES

`v = 4` `c = 1` `colors = [1,2,2,3,1]`

Hamiltonian Path

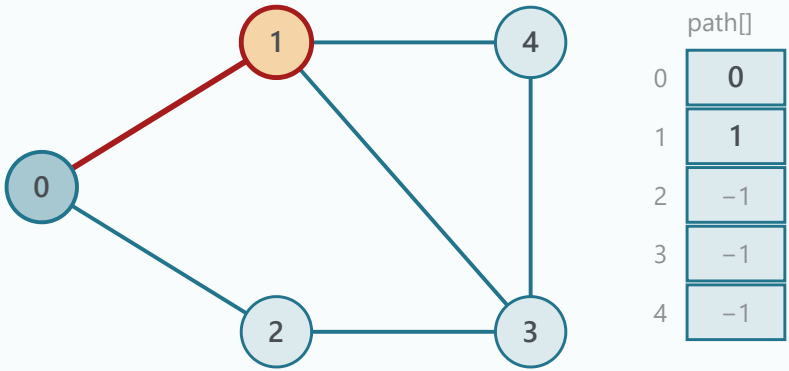
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



`path[0] = 0`; `pos = 1`. At each position vertices 0, 1, 2, … are tried in order: a vertex can be added if it is **adjacent** to the previous one and **not** on the path.

Hamiltonian Path

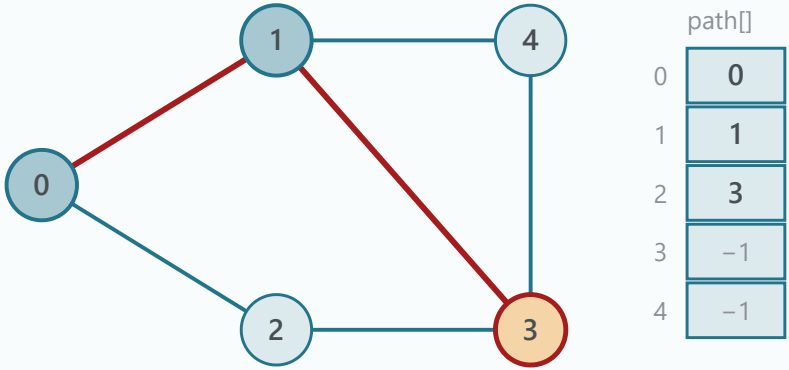
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 1: vertex **1** fits → `path[1] = 1`.

Hamiltonian Path

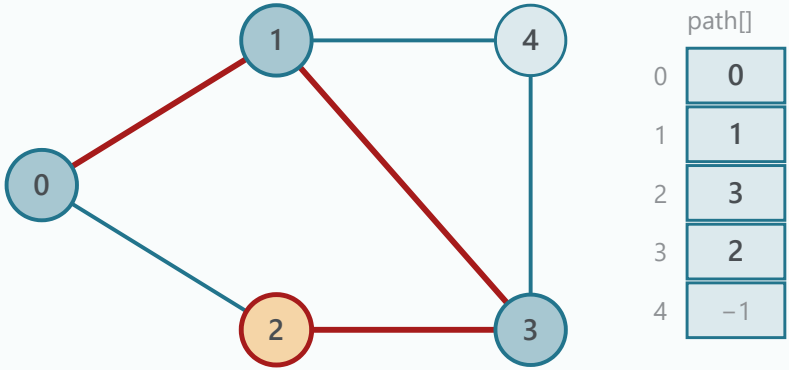
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 2: vertex **3** fits → `path[2] = 3`.

Hamiltonian Path

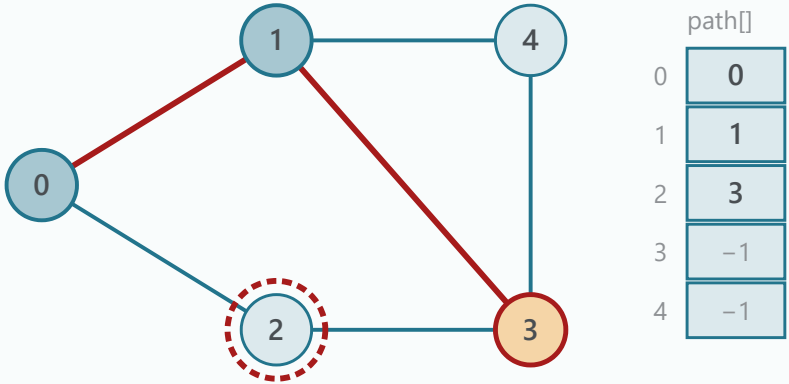
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 3: vertex **2** fits → `path[3] = 2`.

Hamiltonian Path

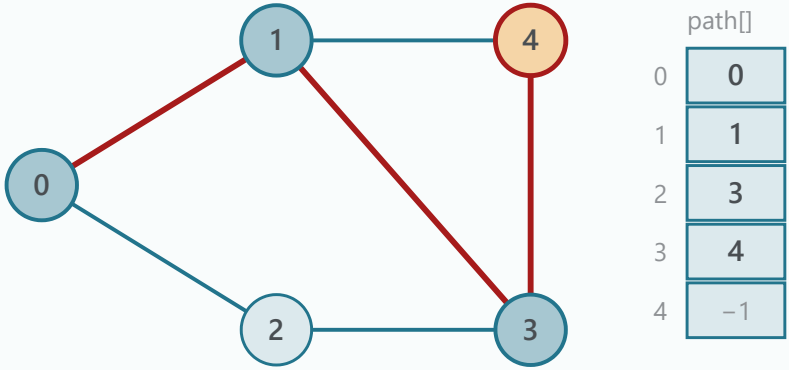
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 4: the last vertex, 2, has no (untried) neighbor off the path → **dead end**. Backtrack: `path[3] = -1` (vertex 2 is removed).

Hamiltonian Path

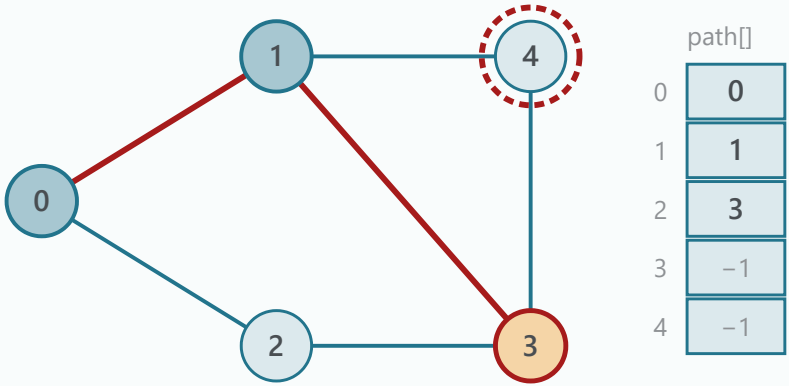
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 3: vertex **4** fits → `path[3] = 4`.

Hamiltonian Path

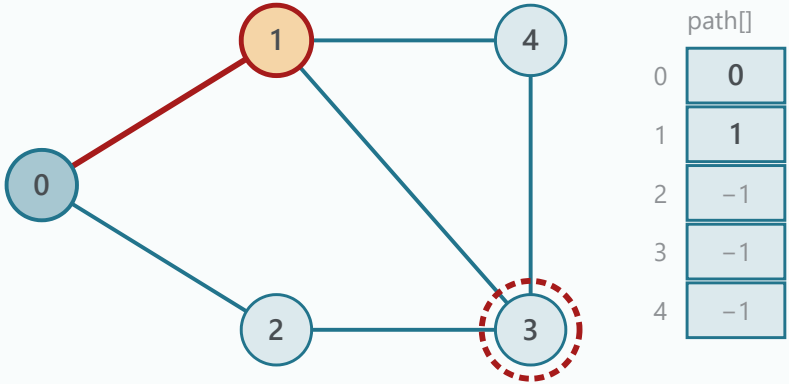
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 4: the last vertex, 4, has no (untried) neighbor off the path → **dead end**. Backtrack: `path[3] = -1` (vertex 4 is removed).

Hamiltonian Path

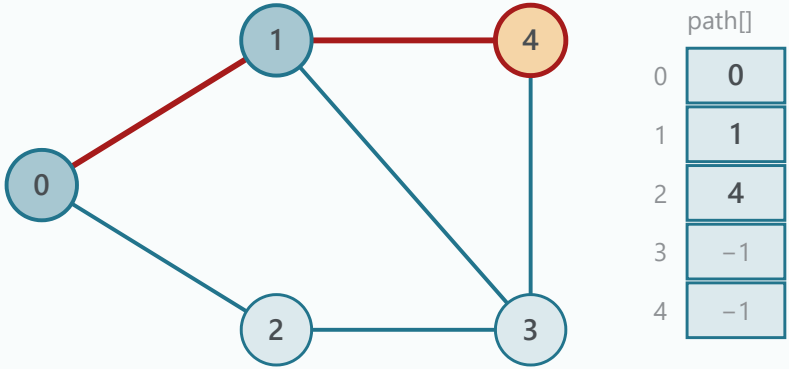
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 3: the last vertex, 3, has no (untried) neighbor off the path → **dead end**. Backtrack: `path[2] = -1` (vertex 3 is removed).

Hamiltonian Path

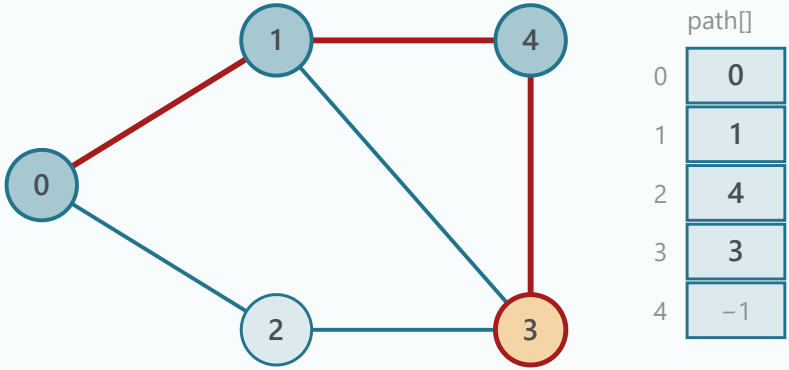
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 2: vertex **4** fits → `path[2] = 4`.

Hamiltonian Path

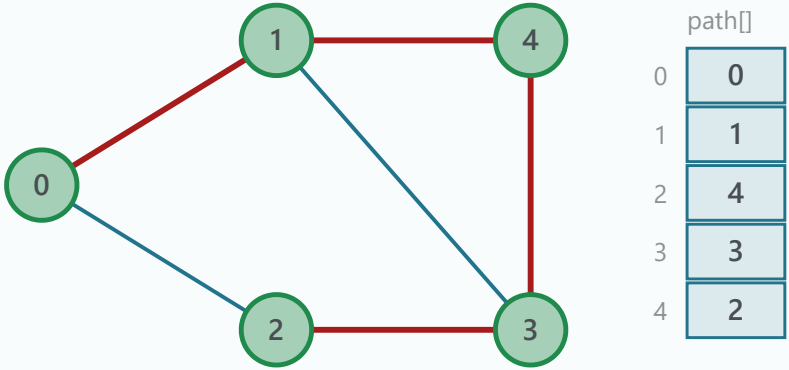
A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 3: vertex **3** fits → `path[3] = 3`.

Hamiltonian Path

A path that visits **every vertex exactly once**. Partial solution: `path[0..pos-1]`; constraint: the new vertex is adjacent to the last one and not already on the path.



Position 4: vertex **2** fits → `path[4] = 2`. `pos == 5`: every vertex is on the path; **Hamiltonian path** found: 0 → 1 → 4 → 3 → 2.

Hamiltonian Path: Java Code

```
boolean find(int[][] graph, int[] path, int pos) {
  if (pos == graph.length) return true;
  for (int v = 0; v < graph.length; v++) {
    if (isSafe(v, graph, path, pos)) {
      path[pos] = v;
      if (find(graph, path, pos + 1)) return true;
      path[pos] = -1;      // undo
    }
  }
  return false;
}

boolean isSafe(int v, int[][] graph,
               int[] path, int pos) {
  if (graph[path[pos - 1]][v] == 0) return false;
  for (int i = 0; i < pos; i++)
    if (path[i] == v) return false;
  return true;
}
```

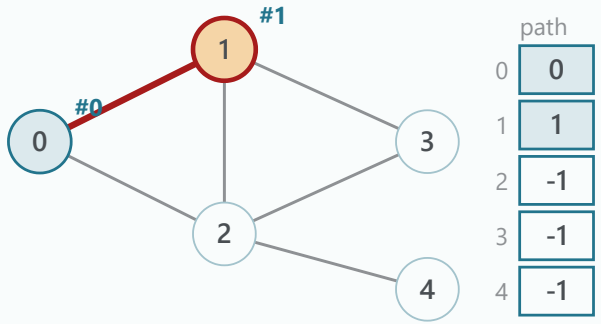
- `graph` : adjacency matrix; start with `path[0] = 0` and call `find(graph, path, 1)` .
- In the example path 0 → 1 → 4 → 3 → 2, the last vertex 2 is also adjacent to the start 0; such a path is also a **Hamiltonian cycle**.
- Worst case O(n!) paths; `isSafe` is O(n) → O(n · n!).

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=1)**
- main()



A new graph (5 vertices); `find(graph, path, 1)` with path[0] = 0. Position 1: vertex 0 is on the path; the first vertex adjacent to 0 is **1** → path[1] = 1.

Step 1 / 10

VARIABLES

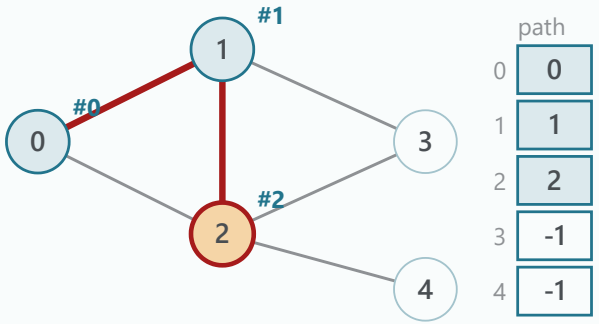
`pos = 1` `v = 1` `path = [0,1,-1,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=2)**
- find(pos=1)
- main()



Position 2: the first vertex adjacent to 1 and not on the path is **2** → path[2] = 2.

Step 2 / 10

VARIABLES

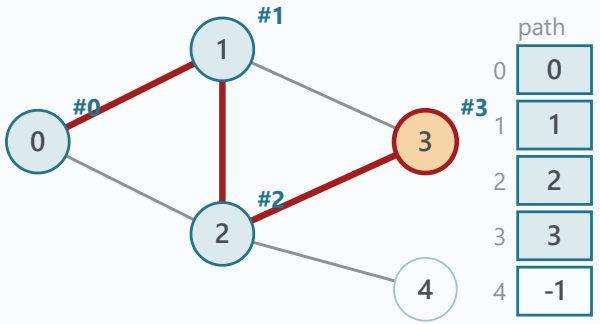
`pos = 2` `v = 2` `path = [0,1,2,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=3)**
- find(pos=2)
- find(pos=1)
- main()



Position 3: the first vertex adjacent to 2 and not on the path is **3** → path[3] = 3.

Step 3 / 10

VARIABLES

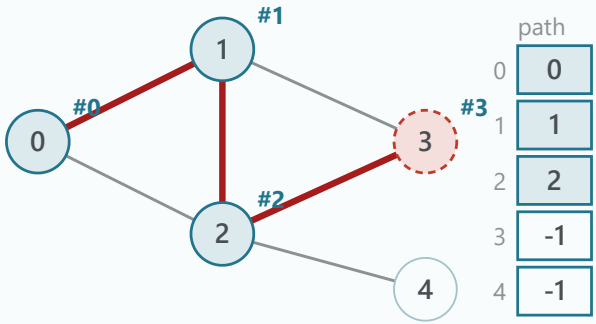
`pos = 3` `v = 3` `path = [0,1,2,3,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
4      if (isSafe(v, graph, path, pos)) {
5        path[pos] = v;
6        if (find(graph, path, pos + 1)) return true;
▶        path[pos] = -1;       // undo
8      }
9    }
▶    return false;
11 }
```

CALL STACK

- **find(pos=3)**
- find(pos=2)
- find(pos=1)
- main()



Position 4: the last vertex, 3, has no neighbor off the path → **dead end**, false. Undo: path[3] = −1 (vertex 3 is removed).

Step 4 / 10

VARIABLES

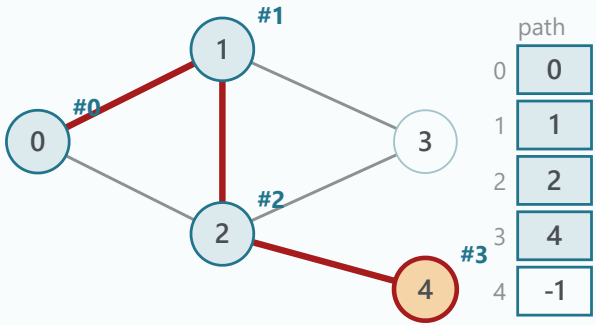
`pos = 3` `v = 3` `path = [0,1,2,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=3)**
- find(pos=2)
- find(pos=1)
- main()



Position 3: the first vertex adjacent to 2 and not on the path is **4** → path[3] = 4.

Step 5 / 10

VARIABLES

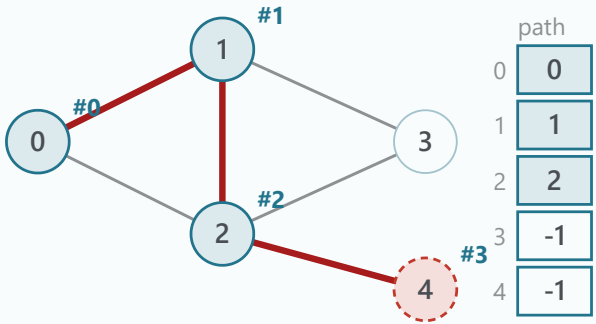
`pos = 3` `v = 4` `path = [0,1,2,4,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
4      if (isSafe(v, graph, path, pos)) {
5        path[pos] = v;
6        if (find(graph, path, pos + 1)) return true;
▶        path[pos] = -1;       // undo
8      }
9    }
▶    return false;
11 }
```

CALL STACK

- **find(pos=3)**
- find(pos=2)
- find(pos=1)
- main()



Position 4: the last vertex, 4, has no neighbor off the path → **dead end**, false. Undo: path[3] = −1 (vertex 4 is removed).

Step 6 / 10

VARIABLES

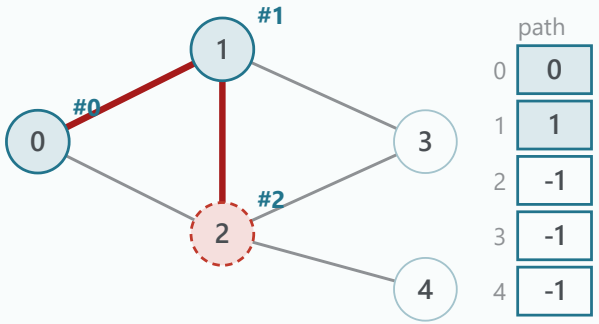
`pos = 3` `v = 4` `path = [0,1,2,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
4      if (isSafe(v, graph, path, pos)) {
5        path[pos] = v;
6        if (find(graph, path, pos + 1)) return true;
▶        path[pos] = -1;       // undo
8      }
9    }
▶    return false;
11 }
```

CALL STACK

- `find(pos=2)`
- `find(pos=1)`
- `main()`



Position 3: the last vertex, 2, has no neighbor off the path → **dead end**, false. Undo: path[2] = −1 (vertex 2 is removed).

Step 7 / 10

VARIABLES

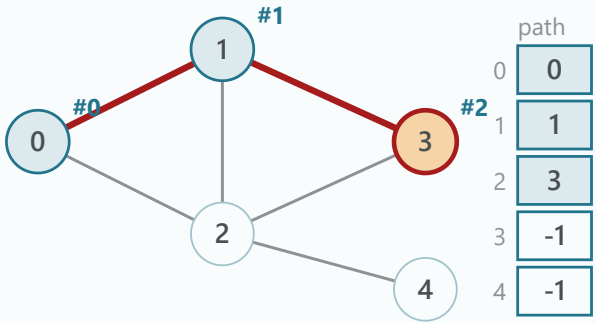
`pos = 2` `v = 2` `path = [0,1,-1,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=2)**
- find(pos=1)
- main()



Position 2: the first vertex adjacent to 1 and not on the path is **3** → path[2] = 3.

Step 8 / 10

VARIABLES

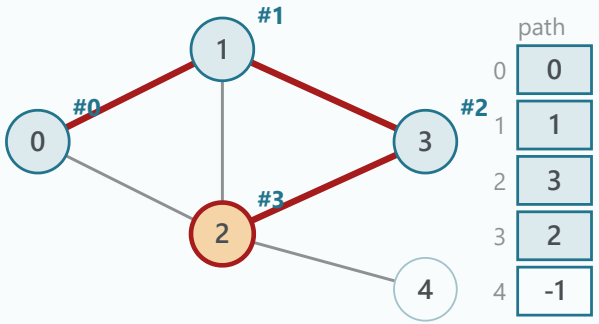
`pos = 2` `v = 3` `path = [0,1,3,-1,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
2    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
▶      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- **find(pos=3)**
- find(pos=2)
- find(pos=1)
- main()



Position 3: the first vertex adjacent to 3 and not on the path is **2** → path[3] = 2.

Step 9 / 10

VARIABLES

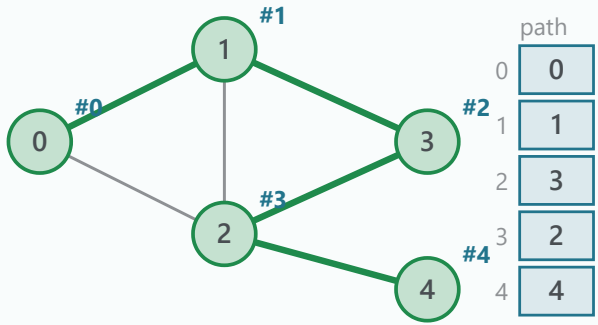
`pos = 3` `v = 2` `path = [0,1,3,2,-1]`

find(graph, path, 1): Code Trace

```
1  boolean find(int[][] graph, int[] path, int pos) {
▶    if (pos == graph.length) return true;
3    for (int v = 0; v < graph.length; v++) {
4      if (isSafe(v, graph, path, pos)) {
▶        path[pos] = v;
▶        if (find(graph, path, pos + 1)) return true;
7        path[pos] = -1;      // undo
8      }
9    }
10   return false;
11 }
```

CALL STACK

- find(pos=5)
- find(pos=4)
- find(pos=3)
- find(pos=2)
- find(pos=1)
- main()



Position 4: the first vertex adjacent to 2 and not on the path is **4** → path[4] = 4. pos = 5: every vertex is on the path → **Hamiltonian path 0 → 1 → 3 → 2 → 4**; true propagates up.

Step 10 / 10

VARIABLES

`pos = 4` `v = 4` `path = [0,1,3,2,4]`

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	**B**	**C**	**E**
S	**F**	**C**	**S**
A	**D**	**E**	**E**

The word is **"SEE"**. Every cell is tried as a start (row by row); neighbor directions: up, down, left, right. A cell cannot be used twice on the same path.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	E
0 S	F	C	S
A	D	E	E

(1, 0) = 'S' matches letter 0 of the word → marked as visited.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	E
S	F	C	S
A	D	E	E

No neighbor of (1, 0) continues with 'E' → **backtrack**, the mark is removed.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	E
S	F	C	0 S
A	D	E	E

(1, 3) = 'S' matches letter 0 of the word → marked as visited.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	1 E
S	F	C	0 S
A	D	E	E

(0, 3) = 'E' matches letter 1 of the word → marked as visited.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	E
S	F	C	0 S
A	D	E	E

No neighbor of (0, 3) continues with 'E' → **backtrack**, the mark is removed.

Word Search

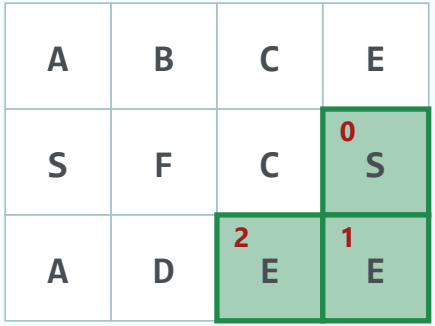
Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.

A	B	C	E
S	F	C	0 S
A	D	E	1 E

(2, 3) = 'E' matches letter 1 of the word → marked as visited.

Word Search

Is there a path through adjacent (up/down/left/right) cells of a letter grid that spells the word? Each cell can be used at most once on a path.



(2, 2) = 'E' matches letter 2 of the word → marked as visited. The last letter matched too: **"SEE" found**.

Word Search: Java Code

```
boolean search(char[][] grid, String word, int i,
               int r, int c, boolean[][] visited) {
  if (i == word.length()) return true;
  if (r < 0 || r >= grid.length || c < 0
      || c >= grid[0].length || visited[r][c]
      || grid[r][c] != word.charAt(i))
    return false;
  visited[r][c] = true;
  for (int d = 0; d < 4; d++)
    if (search(grid, word, i + 1, r + dr[d],
               c + dc[d], visited))
      return true;
  visited[r][c] = false;    // undo
  return false;
}
```

Word	Result
ABCCED	found
SEE	found
ABCB	not found: B cannot be used twice
SADE	found

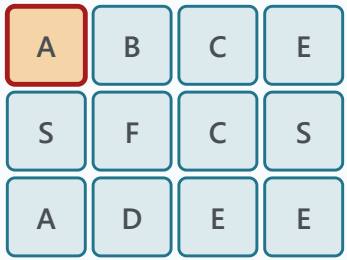
Grid: ABCE / SFCS / ADEE. When backtracking, the `visited` mark **must be cleared**; otherwise the cell cannot be used on another path.

search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2              int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
4    if (r < 0 || r >= grid.length || c < 0
5        || c >= grid[0].length || visited[r][c]
6        || grid[r][c] != word.charAt(i))
7      return false;
▶    visited[r][c] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (search(grid, word, i + 1, r + dr[d],
11             c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- `search(i=0, 0, 0)`
- `main()`





Grid ABCE / SFCS / ADEE, word **"ABCB"**; start at (0, 0). (0, 0) = 'A' = word[0] → marked visited; directions: up, down, left, right.

Step 1 / 8

VARIABLES

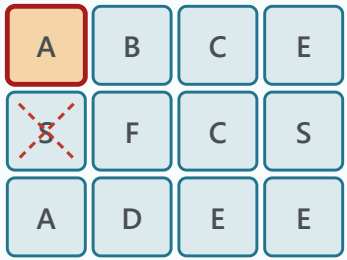
`i = 0` `s = 0` `k = 0`

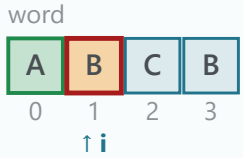
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2                 int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
▶    if (r < 0 || r >= grid.length || c < 0
▶        || c >= grid[0].length || visited[r][c]
▶        || grid[r][c] != word.charAt(i))
▶      return false;
8    visited[r][c] = true;
9    for (int d = 0; d < 4; d++)
10     if (search(grid, word, i + 1, r + dr[d],
11                c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- `search(i=0, 0, 0)`
- `main()`





Looking for word[1] = 'B': up: off the grid; down: 'S' ≠ 'B'; left: off the grid → false.

Step 2 / 8

VARIABLES

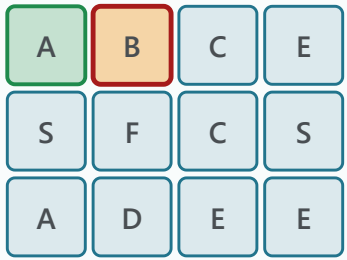
`i = 1` `s = 0` `k = 0`

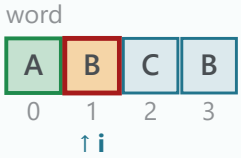
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2              int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
4    if (r < 0 || r >= grid.length || c < 0
5        || c >= grid[0].length || visited[r][c]
6        || grid[r][c] != word.charAt(i))
7      return false;
▶    visited[r][c] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (search(grid, word, i + 1, r + dr[d],
11             c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- `search(i=1, 0, 1)`
- `search(i=0, 0, 0)`
- `main()`





(0, 1) = 'B' = word[1] → marked as visited.

Step 3 / 8

VARIABLES

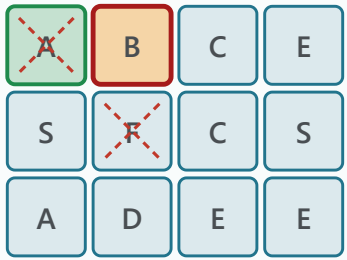
`i = 1` `s = 0` `k = 1`

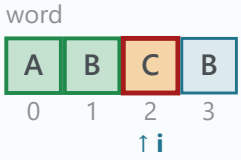
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2                 int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
▶    if (r < 0 || r >= grid.length || c < 0
▶        || c >= grid[0].length || visited[r][c]
▶        || grid[r][c] != word.charAt(i))
▶      return false;
8    visited[r][c] = true;
9    for (int d = 0; d < 4; d++)
10     if (search(grid, word, i + 1, r + dr[d],
11                c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- `search(i=1, 0, 1)`
- `search(i=0, 0, 0)`
- `main()`





Looking for word[2] = 'C': up: off the grid; down: 'F' ≠ 'C'; left: (0, 0) **visited** → false.

Step 4 / 8

VARIABLES

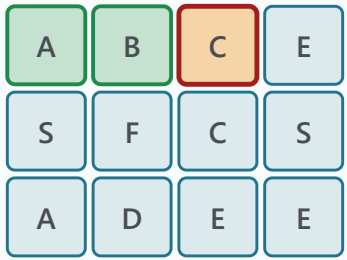
`i = 2` `s = 0` `k = 1`

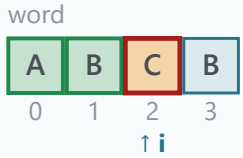
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2              int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
4    if (r < 0 || r >= grid.length || c < 0
5        || c >= grid[0].length || visited[r][c]
6        || grid[r][c] != word.charAt(i))
7      return false;
▶    visited[r][c] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (search(grid, word, i + 1, r + dr[d],
11             c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- search(i=2, 0, 2)
- search(i=1, 0, 1)
- search(i=0, 0, 0)
- main()





(0, 2) = 'C' = word[2] → marked as visited.

Step 5 / 8

VARIABLES

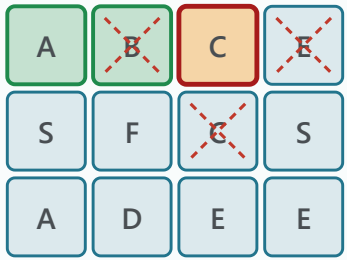
i = 2 s = 0 k = 2

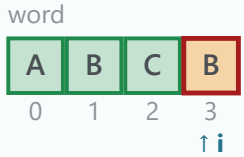
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2                 int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
▶    if (r < 0 || r >= grid.length || c < 0
▶        || c >= grid[0].length || visited[r][c]
▶        || grid[r][c] != word.charAt(i))
▶      return false;
8    visited[r][c] = true;
9    for (int d = 0; d < 4; d++)
10     if (search(grid, word, i + 1, r + dr[d],
11                c + dc[d], visited))
12       return true;
13   visited[r][c] = false;    // undo
14   return false;
15 }
```

CALL STACK

- search(i=2, 0, 2)
- search(i=1, 0, 1)
- search(i=0, 0, 0)
- main()





Looking for word[3] = 'B': up: off the grid; down: 'C' ≠ 'B'; left: (0, 1) **visited**; right: 'E' ≠ 'B' → false. The only B is at (0, 1), and that cell is already on this path.

Step 6 / 8

VARIABLES

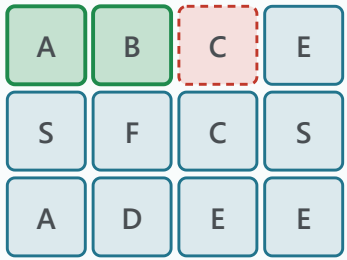
i = 3 | s = 0 | k = 2

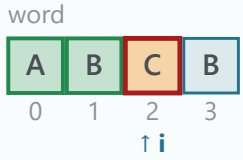
search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2                 int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
4    if (r < 0 || r >= grid.length || c < 0
5        || c >= grid[0].length || visited[r][c]
6        || grid[r][c] != word.charAt(i))
7      return false;
8    visited[r][c] = true;
9    for (int d = 0; d < 4; d++)
10     if (search(grid, word, i + 1, r + dr[d],
11                c + dc[d], visited))
12       return true;
▶    visited[r][c] = false;    // undo
▶    return false;
15 }
```

CALL STACK

- search(i=2, 0, 2)
- search(i=1, 0, 1)
- search(i=0, 0, 0)
- main()





All four directions fail for (0, 2) → **undo**: visited[0][2] = false, return false.

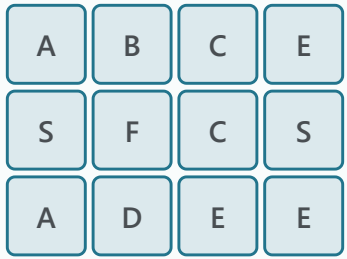
Step 7 / 8

VARIABLES

i = 2 | s = 0 | k = 2

search(..., "ABCB", 0, 0, 0, ...): Code Trace

```
1  boolean search(char[][] grid, String word, int i,
2              int r, int c, boolean[][] visited) {
3    if (i == word.length()) return true;
4    if (r < 0 || r >= grid.length || c < 0
5        || c >= grid[0].length || visited[r][c]
6        || grid[r][c] != word.charAt(i))
7      return false;
8    visited[r][c] = true;
9    for (int d = 0; d < 4; d++)
10     if (search(grid, word, i + 1, r + dr[d],
11             c + dc[d], visited))
12       return true;
▶    visited[r][c] = false;    // undo
▶    return false;
15 }
```



The remaining directions fail too: (0, 1) and (0, 0) are undone in turn and their marks cleared. `search(…, 0, 0, 0, …)` = **false**. The outer loop also tries the A at (2, 0); it has no B neighbor: "ABCB" is **not found**.

Step 8 / 8

CALL STACK

`main()`

VARIABLES

`i = 0` `s = 0` `k = 0`

Other Problems Solved by Backtracking

Problem	Choice at each level	Pruning (constraint)
Subset sum / knapsack	take item i? (2 branches)	sum / weight limit exceeded
Traveling salesman (TSP)	next city	path longer than the best known tour
Bin packing	which bin for the item?	bin capacity exceeded
Maximum coverage	pick the set?	remaining sets cannot improve the result
Satisfiability (SAT)	variable = true / false	a clause became false
Minesweeper	is the cell a mine?	contradicts the revealed numbers
Generating permutations / combinations	next item	item already used

In optimization problems (TSP, knapsack), cutting branches that are "worse than the best known" is the basis of **branch and bound**.

ANALYSIS

Complexity

Problem	Worst-case time	Description
n-queens	$O(n^2 \cdot n!)$	O(n!) nodes × n columns × O(n) `isSafe`
Sudoku	$O(9^m)$	m: number of empty cells; 9 candidates per cell
Graph m-coloring	$O(n \cdot m^n)$	n vertices, m colors each; check is O(degree)
Hamiltonian path	$O(n \cdot n!)$	remaining vertices for each position on the path
Word search	$O(R \cdot C \cdot 3^L)$	after the first letter at most 3 new directions (the cell we came from is visited)
Subsets / permutations	$O(2^n)$ / $O(n \cdot n!)$	number of leaves in the tree

Time = number of nodes × work per node

Pruning reduces the number of nodes but (usually) not the worst case.

Memory: O(depth)

Only one partial solution and the stack frames are kept (e.g. O(n) for n-queens).

ANALYSIS

Stronger Pruning: Heuristics

Variable ordering (MRV)

Pick the variable with the fewest options left first. In Sudoku it can remove backtracking entirely.

Value ordering

Try the least constraining value first (the min-conflicts idea). Solutions are found sooner.

Forward checking

After an assignment, remove conflicting values from the neighbors' domains; if a domain becomes empty, backtrack at once.

Fast constraint test

Column/diagonal flags in n-queens, row/column/box bitmasks in Sudoku: the test is O(1).

Symmetry breaking

In n-queens, try the first queen only in the left half and mirror the results: about half the work.

Bound

In optimization, cut a branch if the best possible value of its partial solution is worse than the best known solution: branch and bound.

ANALYSIS

What Sets Backtracking Apart?

Paradigm	Idea	Difference from backtracking
Divide and conquer	Split into independent subproblems, combine	No choosing or trying; each subproblem is solved exactly once
Greedy	Locally best choice at each step; **never goes back**	Follows a single branch; a wrong choice cannot be fixed
Dynamic programming	Solve overlapping subproblems once and store them	All options are tried but results are stored; subproblems must be polynomially many
Backtracking	Extend the partial solution, undo if invalid	Choices are tried and undone; constraint satisfaction, all solutions, decision problems

Backtracking is **depth-first search** (DFS) on the state-space tree; the tree is not built in advance but generated during the walk, and invalid branches are never generated.

Summary

Topic	Key points
Hard problems	NP-complete / NP-hard: if P ≠ NP, no polynomial-time exact algorithm
Idea	build the partial solution step by step; if a constraint breaks, undo the last decision and try the next one
Template	goal test → for each candidate: if valid, choose, recurse, undo
State space	a tree walked by DFS; pruning skips invalid subtrees
n-queens	row by row; column + 2 diagonal checks; 92 solutions for 8, first: [0, 4, 7, 5, 2, 6, 1, 3]
Min-conflicts	complete placement + local repair; fast but not complete
Sudoku	1–9 in an empty cell; row/column/box check; much faster with MRV
Others	graph coloring, Hamiltonian path, word search (undo the visited mark!)
Cost	exponential in the worst case (2^n, $n!$, 9^m); memory O(depth)

SUMMARY

Common Mistakes

Forgetting to undo

If `g[r][c] = 0` or `visited = false` is skipped after the recursive call returns, the old choice corrupts later branches.

Not propagating the result

Writing just `solve(...)` instead of `if (solve(...)) return true;`: the search continues even after a solution and the board is undone.

Early `return false`

Leaving the loop at the first invalid candidate; the remaining candidates are never tried.

Recording a shared list

When collecting all solutions, add a copy (`new ArrayList<>(path)`) instead of `results.add(path)`.

Missing constraint checks

Forgetting one diagonal in n-queens or the 3×3 box in Sudoku: invalid "solutions".

Treating local search as complete

Min-conflicts **cannot prove** that no solution exists; hitting the step limit does not mean "no solution".

SUMMARY

Sample Problems

1. 4 queens: how many queens does `solve(b, 0)` place, and how many times does it backtrack, before finding the first solution?
2. What is the algorithm's first solution for 5 queens? How many backtracks?
3. Find a subset of {3, 5, 6, 7} that sums to 15 ("take" branch first; prune when the sum is exceeded). Which is found first?
4. Can the 5-vertex cycle graph (C_5) be 2-colored? What is the minimum number of colors?
5. In the grid ABCE / SFCS / ADEE, why is "ABCB" not found, and along which path is "SADE" found?
6. 8 queens: compare testing all 8^8 complete placements with backtracking (all solutions).

▶ **Show answers**

Exercises

1. Modify the n-queens code to count **all** solutions (n = 8).
2. Generate the permutations of {1, 2, 3} by backtracking. How many nodes does the tree have, including the root?
3. Modify the Hamiltonian path code to look for a **Hamiltonian cycle**; what is the result on the 5-vertex graph of the Hamiltonian path example?
4. Add MRV (cell with fewest candidates first) to the Sudoku solver.
5. How many distinct simple paths lead from the entrance to the exit in the maze example?
6. Add the pruning "cut if the remaining items cannot reach the target" to subset sum.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
- Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012 (Chapter 12: Backtracking).
- Russell and Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., 2020 (Chapter 6: constraint satisfaction, min-conflicts).
- S. Minton, M. Johnston, A. Philips and P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems", *Artificial Intelligence* 58, 1992.
- D. E. Knuth, *The Art of Computer Programming*, Vol. 4B, 2022 (Backtrack programming, Dancing Links).

The End

Chapter 4: Backtracking

Previous: Chapter 3: Dynamic Programming · Next: Chapter 5: Branch and Bound