Computer Search

Alexander S. Kulikov

Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences
and
University of California, San Diego

Computer Search

- There are cases when a solution exists, but it is not easy to construct it by hand

Computer Search

- There are cases when a solution exists, but it is not easy to construct it by hand
- Let computers do the job!

Computer Search

- There are cases when a solution exists, but it is not easy to construct it by hand
- Let computers do the job!
- For two puzzles (*n* queens and 16 diagonals), we will implement a program that will solve a puzzle on your laptop in blink of an eye

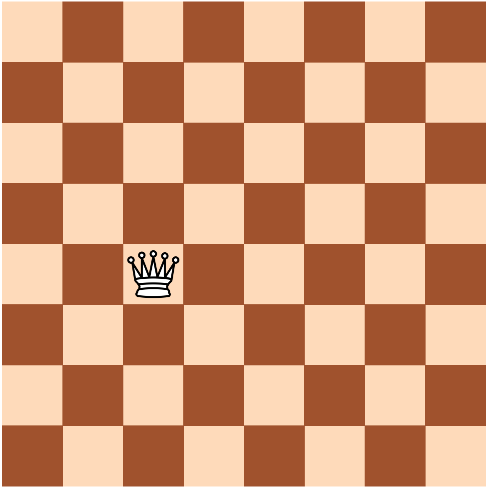
Outline

N Queens: Brute Force Search

N Queens: Backtracking

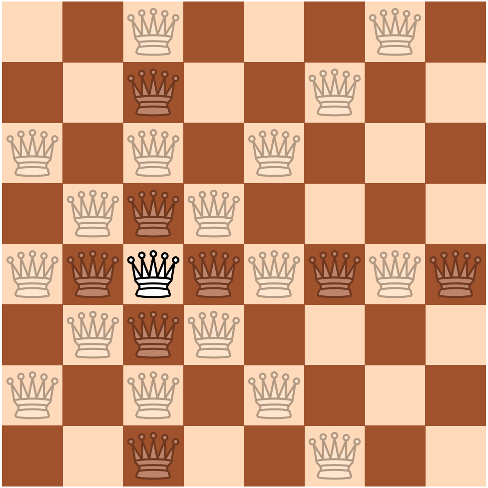
16 Diagonals

Chess Queen



A chess queen moves vertically, horizontally, or diagonally

Chess Queen



A chess queen moves vertically, horizontally, or diagonally

N Queens Problem

Problem

Is it possible to place n queens on an $n \times n$ chessboard such that no two queens attack each other?

Computer Search

- It is known that this is possible for all $n \geq 4$

Computer Search

- It is known that this is possible for all $n \geq 4$
- But already for $n = 8$ it is not easy to construct a solution by hand

Speculating

- Since we are placing n queens on an $n \times n$ board, there should be exactly one queen in each column

Speculating

- Since we are placing n queens on an $n \times n$ board, there should be exactly one queen in each column
    - If there are two queens in a column, they attack each other. Hence at most one queen in each column

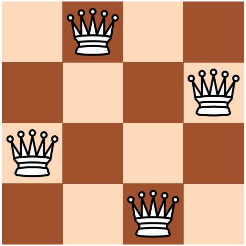
Speculating

- Since we are placing n queens on an $n \times n$ board, there should be exactly one queen in each column
    - If there are two queens in a column, they attack each other. Hence at most one queen in each column
    - If there is a column without a queen, then the total number of queens is less than n

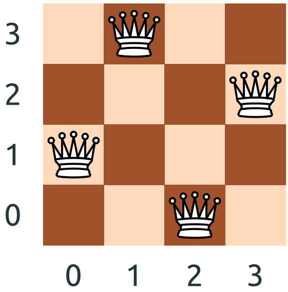
Speculating

- Since we are placing n queens on an $n \times n$ board, there should be exactly one queen in each column
  - If there are two queens in a column, they attack each other. Hence at most one queen in each column
  - If there is a column without a queen, then the total number of queens is less than n
- For the same reason, there should be exactly one queen in each row

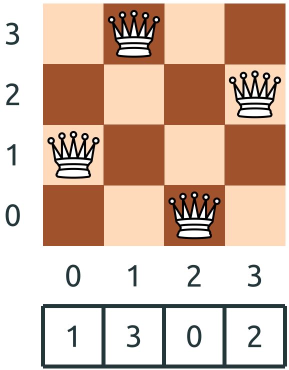
Solution is a Permutation



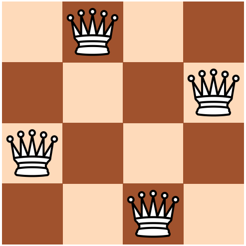
Solution is a Permutation



Solution is a Permutation

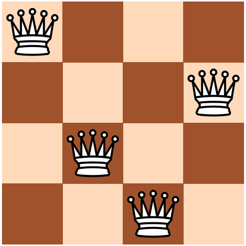


But Not Every Permutation is a Solution



1	3	0	2

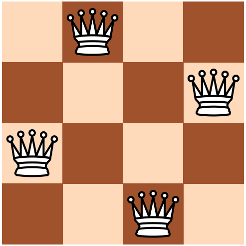
✓



3	1	0	2

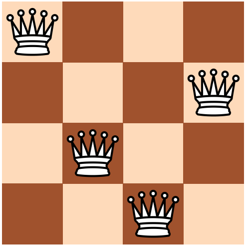
✗

But Not Every Permutation is a Solution



1	3	0	2

✓

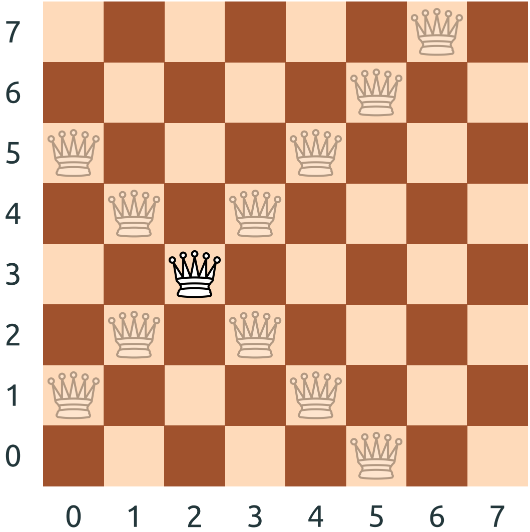


3	1	0	2

✗

How to check whether a permutation is a solution?

Cells in the Same Diagonal



cells $[i_1, j_1]$ and $[i_2, j_2]$ are on the same diagonal, if and only if $|i_1 - i_2| = |j_1 - j_2|$

Is a Solution?

```
import itertools as it

def is_solution(perm):
    for (i1, i2) in it.combinations(range(len(perm)), 2):
        if abs(i1 - i2) == abs(perm[i1] - perm[i2]):
            return False

    return True

assert(is_solution([1, 3, 0, 2]) == True)
assert(is_solution([3, 1, 0, 2]) == False)
```

Brute Force Search Program

```
import itertools as it

def is_solution(perm):
    for (i1, i2) in it.combinations(range(len(perm)), 2):
        if abs(i1 - i2) == abs(perm[i1] - perm[i2]):
            return False

    return True

for perm in it.permutations(range(8)):
    if is_solution(perm):
        print(perm)
        exit()
```

Conclusion

- The program finds a solution for $n = 8$ immediately

Conclusion

- The program finds a solution for $n = 8$ immediately
- But already for $n = 13$ takes too long

Conclusion

- The program finds a solution for $n = 8$ immediately
- But already for $n = 13$ takes too long
- Next part: will optimize the program so that it is able to find a solution quickly even for $n = 20$

Outline

N Queens: Brute Force Search

N Queens: Backtracking

16 Diagonals

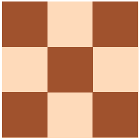
Main Idea of Backtracking

- Construct a permutation piece by piece

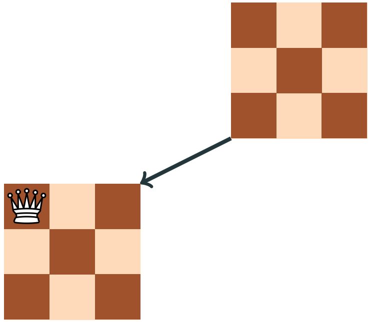
Main Idea of Backtracking

- Construct a permutation piece by piece
- Backtrack if the current partial permutation cannot be extended to a valid solution

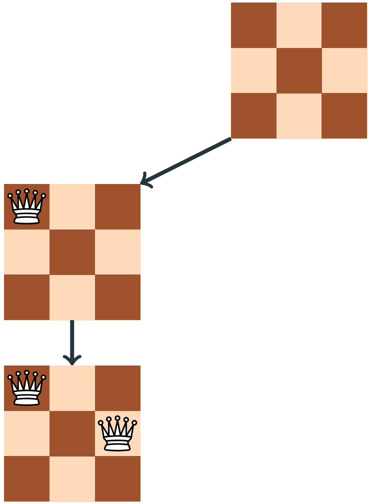
Backtracking: 3×3 **Board**



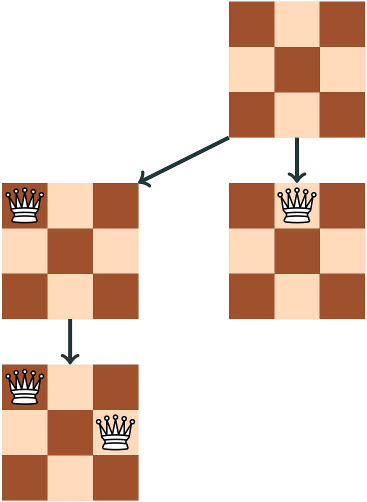
Backtracking: 3×3 Board



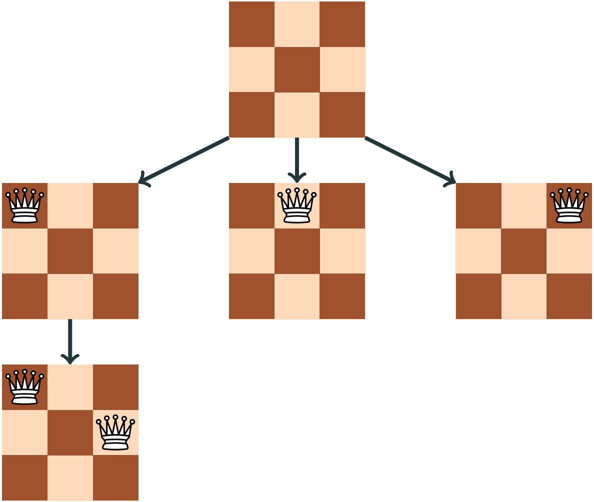
Backtracking: 3×3 Board



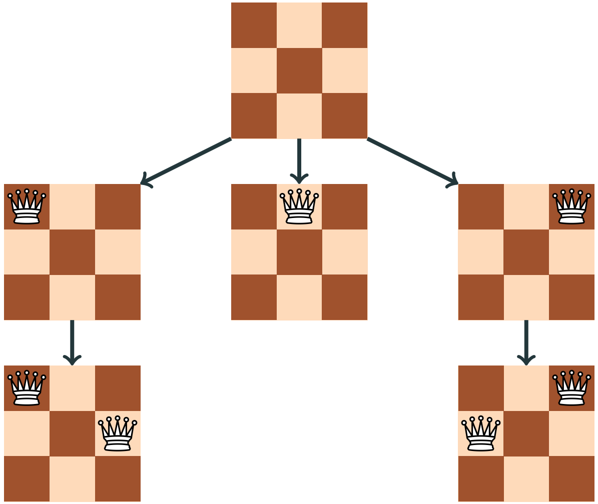
Backtracking: 3×3 **Board**

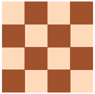


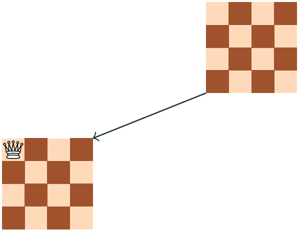
Backtracking: 3×3 Board

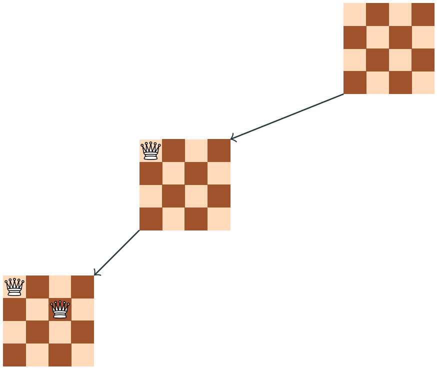


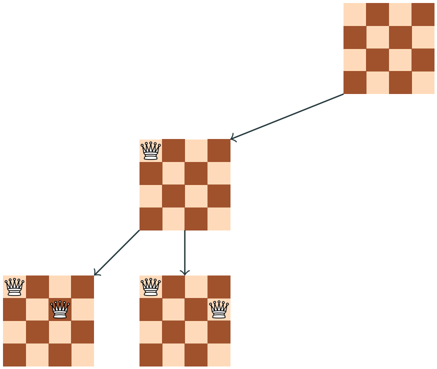
Backtracking: 3×3 **Board**

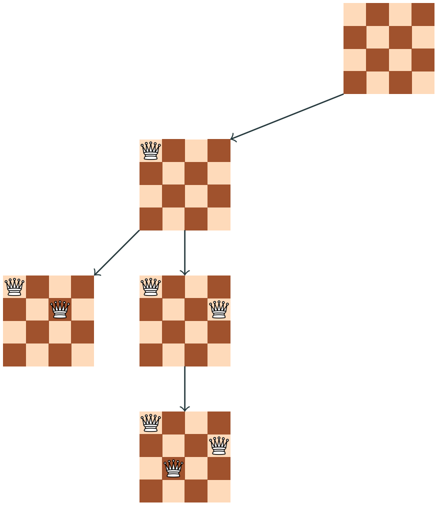


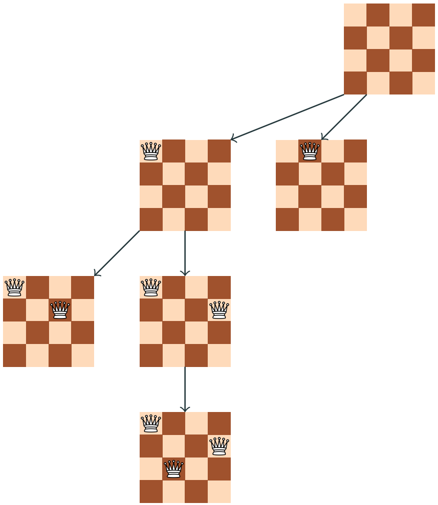


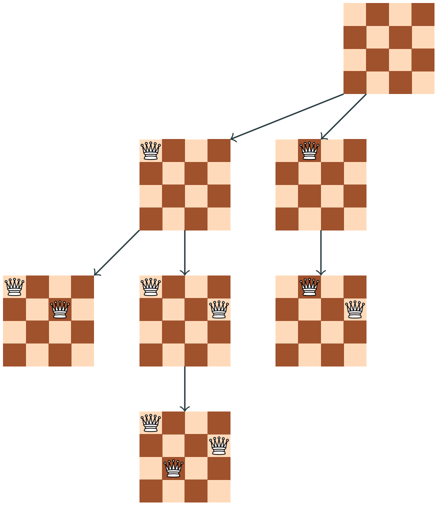


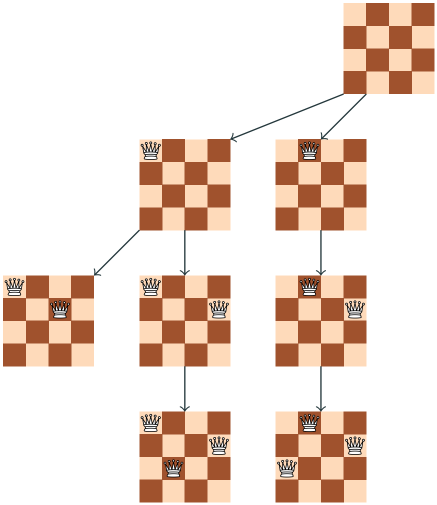


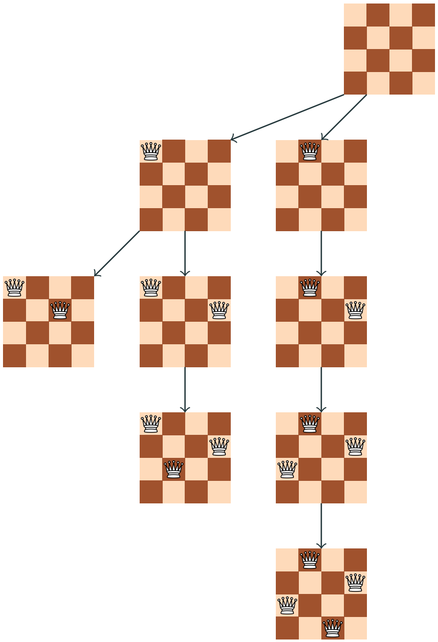


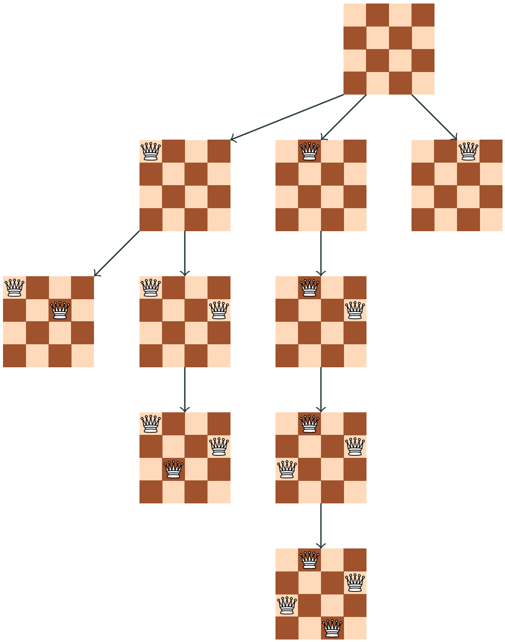


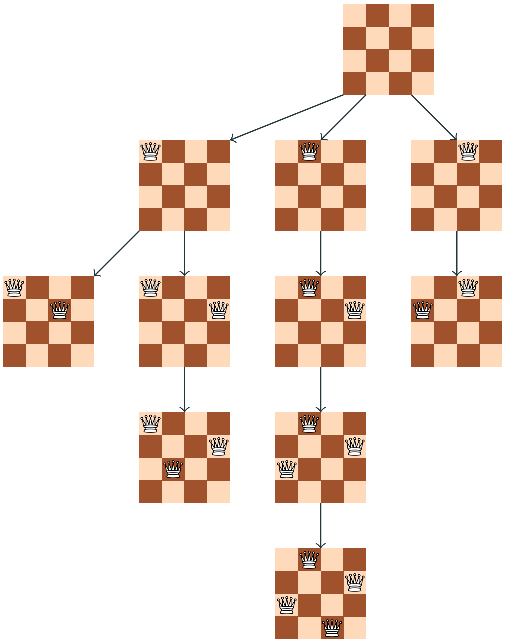


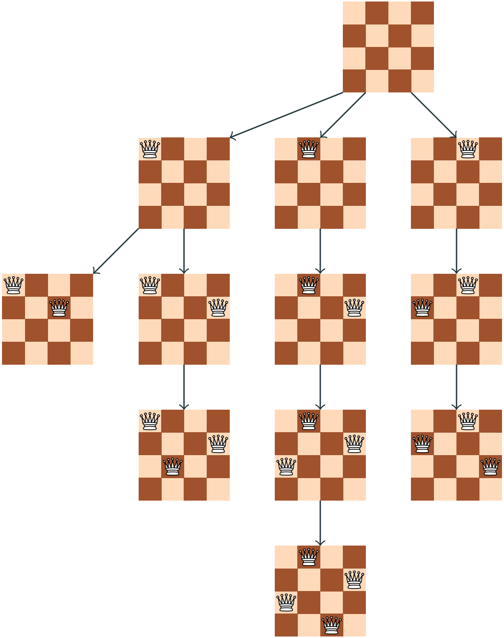


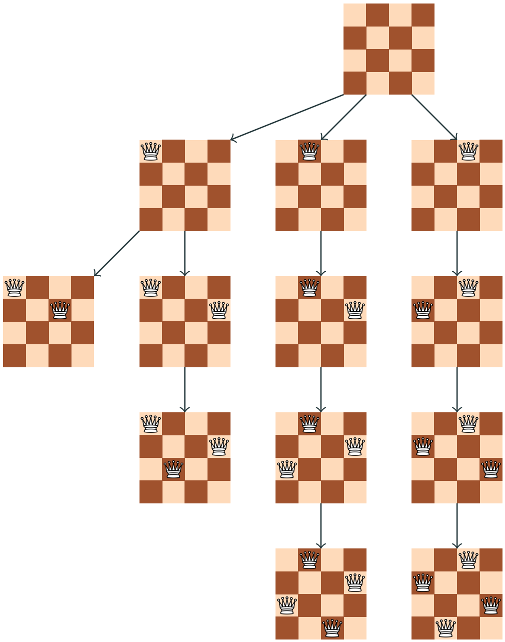


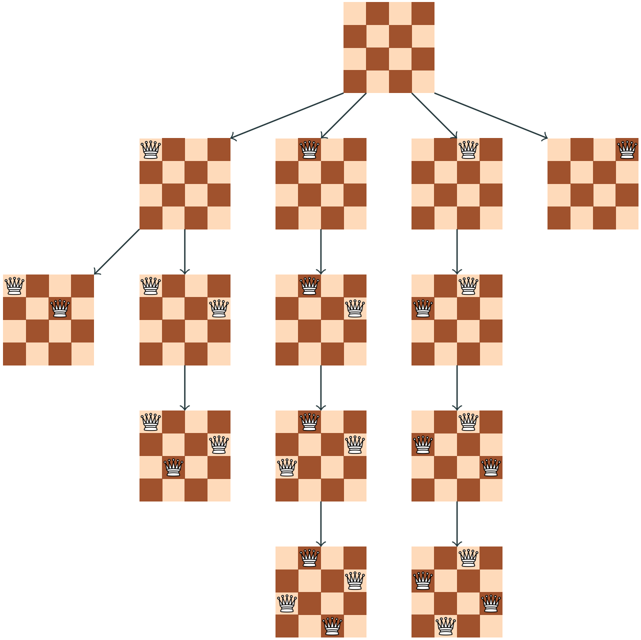


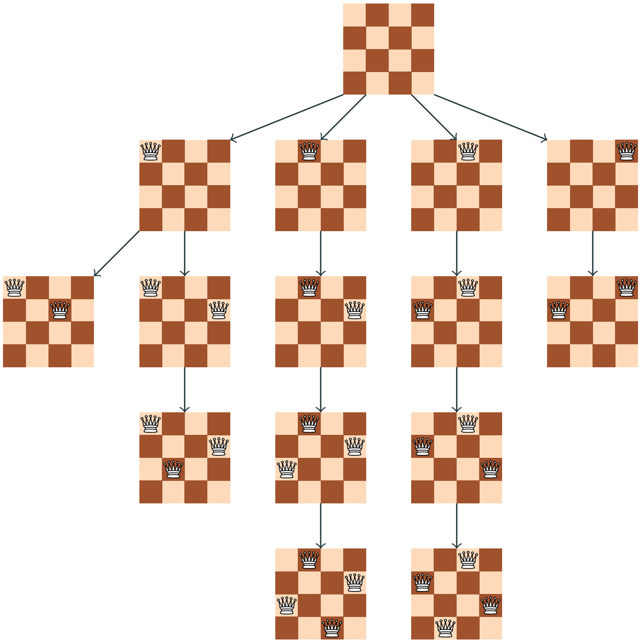


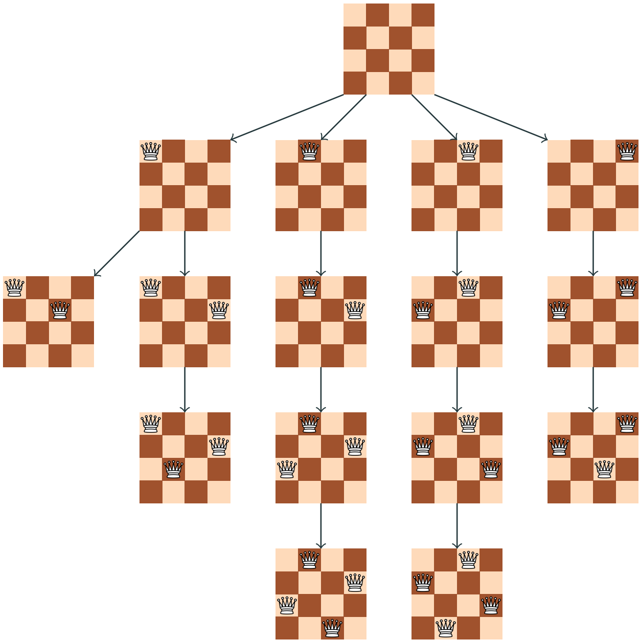


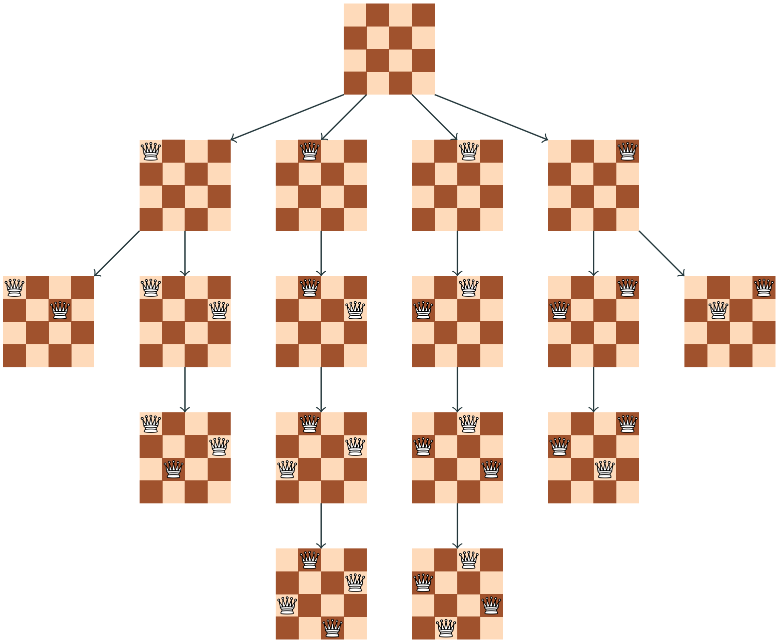












Generating All Permutations

```
def generate_permutations(perm, n):
    if len(perm) == n:
        print(perm)
        return

    for k in range(n):
        if k not in perm:
            perm.append(k)
            generate_permutations(perm, n)
            perm.pop()


generate_permutations(perm = [], n = 4)
```

Output

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

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

Idea

If the current (partial) permutation cannot be extended to a solution (i.e., it already contains two queens that attack each other), stop trying to extend it

Idea

If the current (partial) permutation cannot be extended to a solution (i.e., it already contains two queens that attack each other), stop trying to extend it

```
def can_be_extended_to_solution(perm):
    i = len(perm) - 1
    for j in range(i):
        if i - j == abs(perm[i] - perm[j]):
            return False
    return True
```

Resulting Program

```
def extend(perm, n):
    if len(perm) == n:
        print(perm)
        exit()

    for k in range(n):
        if k not in perm:
            perm.append(k)

            if can_be_extended_to_solution(perm):
                extend(perm, n)

            perm.pop()

extend(perm = [], n = 20)
```

Summary

- Main idea of backtracking: cut dead ends of the recursion tree

Summary

- Main idea of backtracking: cut dead ends of the recursion tree
- Since many ends are dead, it works faster than a naive enumeration of all permutations

Outline

N Queens: Brute Force Search

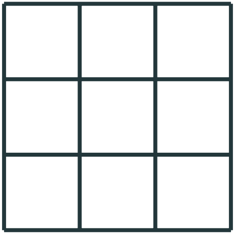
N Queens: Backtracking

16 Diagonals

3×3 **Grid: 6 Diagonals**

Problem

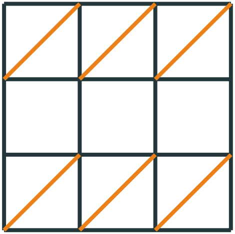
In a 3×3 grid, draw 6 diagonals that do not touch each other.



3×3 Grid: 6 Diagonals

Problem

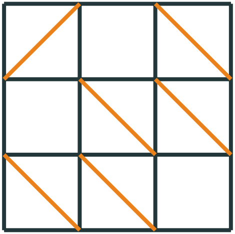
In a 3×3 grid, draw 6 diagonals that do not touch each other.



3×3 **Grid: 6 Diagonals**

Problem

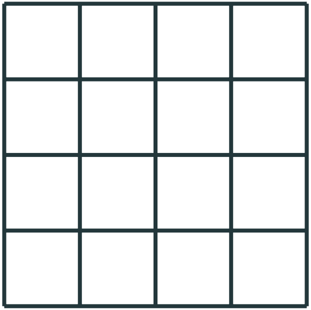
In a 3×3 grid, draw 6 diagonals that do not touch each other.



4×4 Grid: 10 Diagonals

Problem

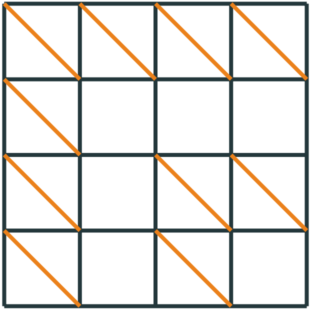
In a 4×4 grid, draw 10 diagonals that do not touch each other.



4×4 Grid: 10 Diagonals

Problem

In a 4×4 grid, draw 10 diagonals that do not touch each other.



5×5 Grid: 16 Diagonals

Problem

In a 5×5 grid, draw 16 diagonals that do not touch each other.

5×5 Grid: 16 Diagonals

Problem

In a 5×5 grid, draw 16 diagonals that do not touch each other.

Exercise

Implement a backtracking procedure for solving this problem.

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)
- For each cell consider three possibilities:

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)
- For each cell consider three possibilities:
    1. Diagonal from BL corner to TR corner

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)
- For each cell consider three possibilities:
  1. Diagonal from BL corner to TR corner
  2. Diagonal from BR corner to TL corner

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)
- For each cell consider three possibilities:
  1. Diagonal from BL corner to TR corner
  2. Diagonal from BR corner to TL corner
  3. No diagonal

Suggestion

- Fill in the grid gradually (say, from bottom to top, from left to right)
- For each cell consider three possibilities:
    1. Diagonal from BL corner to TR corner
    2. Diagonal from BR corner to TL corner
    3. No diagonal
- Each time when a new diagonal is placed, check whether it conflicts with other diagonals. If it does, backtrack