Problem 1222: Queens That Can Attack the King

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

On a

0-indexed

8 x 8

chessboard, there can be multiple black queens and one white king.

You are given a 2D integer array

queens

where

queens[i] = [xQueen

i

, yQueen

i

]

represents the position of the

i

th

black queen on the chessboard. You are also given an integer array

king

of length

2

where

king = [xKing, yKing]

represents the position of the white king.

Return

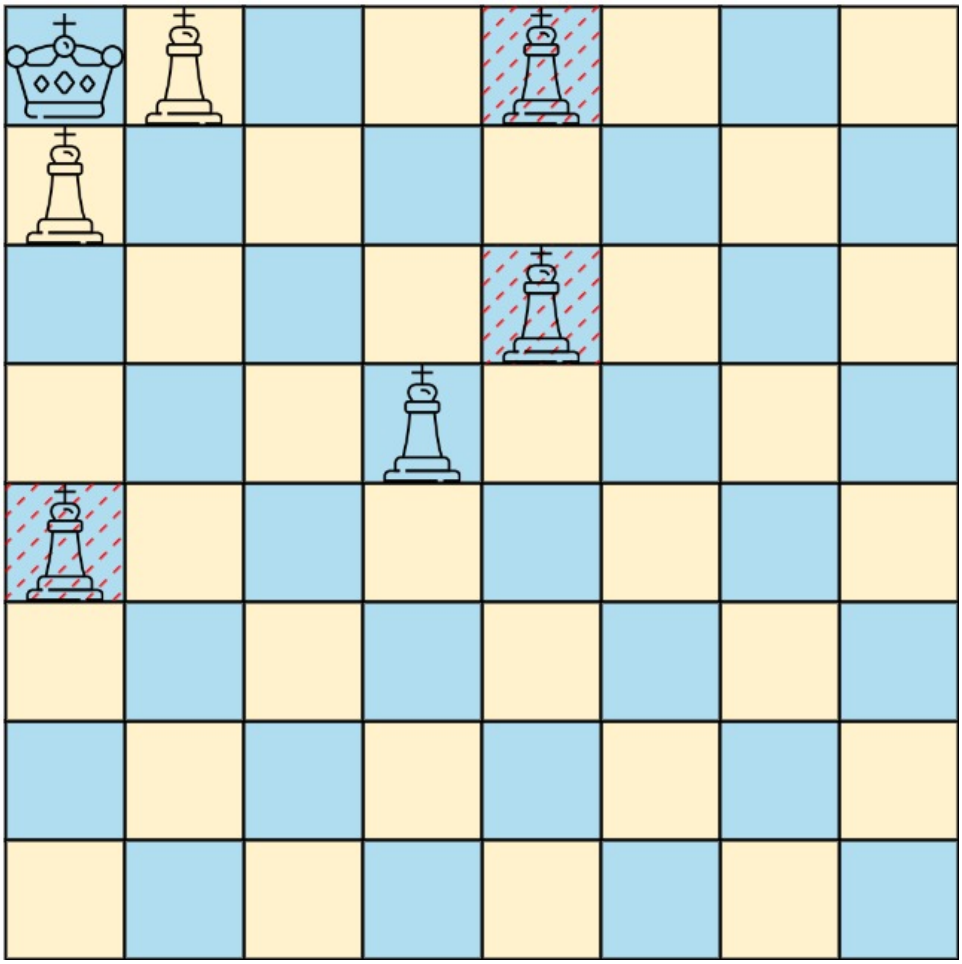
the coordinates of the black queens that can directly attack the king

. You may return the answer in

any order

.

Example 1:



Input:

queens = [[0,1],[1,0],[4,0],[0,4],[3,3],[2,4]], king = [0,0]

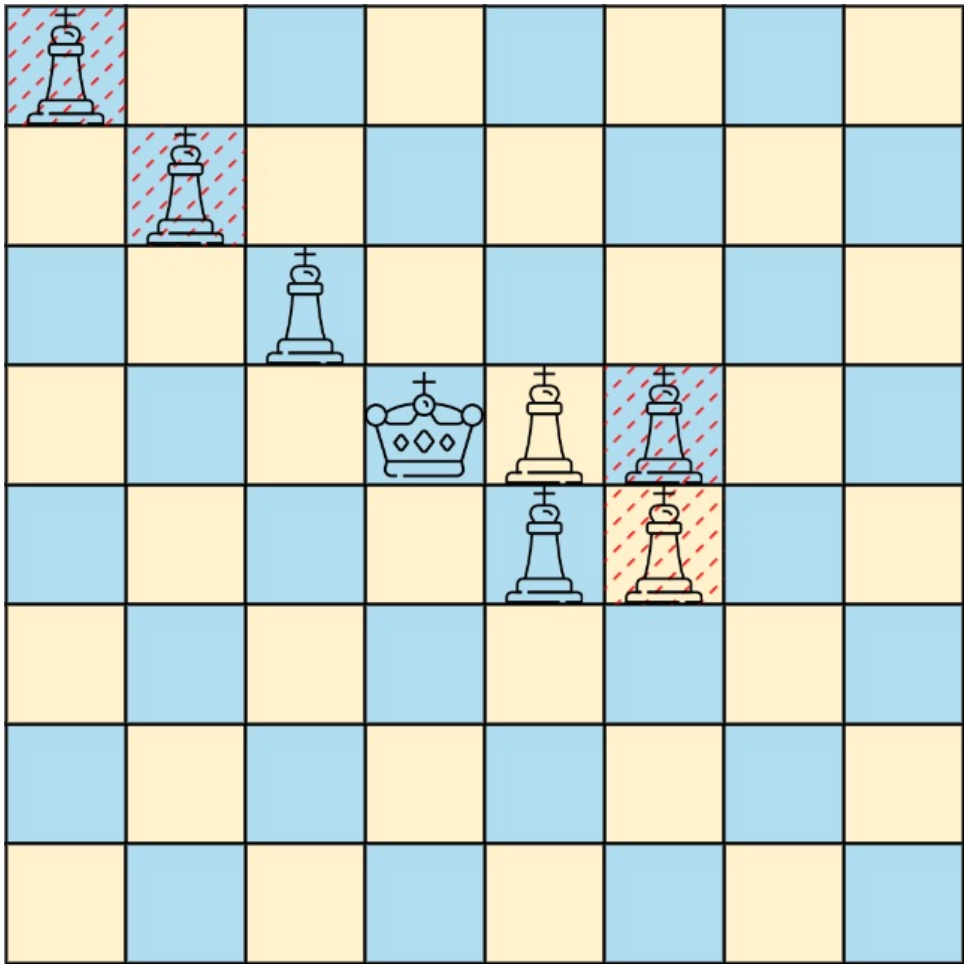
Output:

[[0,1],[1,0],[3,3]]

Explanation:

The diagram above shows the three queens that can directly attack the king and the three queens that cannot attack the king (i.e., marked with red dashes).

Example 2:



Input:

queens = [[0,0],[1,1],[2,2],[3,4],[3,5],[4,4],[4,5]], king = [3,3]

Output:

[[2,2],[3,4],[4,4]]

Explanation:

The diagram above shows the three queens that can directly attack the king and the three queens that cannot attack the king (i.e., marked with red dashes).

Constraints:

1 <= queens.length < 64

queens[i].length == king.length == 2

0 <= xQueen

i

, yQueen

i

, xKing, yKing < 8

All the given positions are

unique

.

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> queensAttacktheKing(vector<vector<int>>& queens,
vector<int>& king) {

}
};
```

Java:

```
class Solution {
public List<List<Integer>> queensAttacktheKing(int[][] queens, int[] king) {

}
}
```

Python3:

```
class Solution:
def queensAttacktheKing(self, queens: List[List[int]], king: List[int]) ->
List[List[int]]:
```

Python:

```
class Solution(object):
def queensAttacktheKing(self, queens, king):
"""
:type queens: List[List[int]]
:type king: List[int]
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} queens
* @param {number[]} king
* @return {number[][]}
*/
var queensAttacktheKing = function(queens, king) {

};
```

TypeScript:

```
function queensAttacktheKing(queens: number[][], king: number[]): number[][]
{

};
```

C#:

```
public class Solution {
public IList<IList<int>> QueensAttacktheKing(int[][] queens, int[] king) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** queensAttacktheKing(int** queens, int queensSize, int* queensColSize,
int* king, int kingSize, int* returnSize, int** returnColumnSizes) {

}
```

Go:

```
func queensAttacktheKing(queens [][]int, king []int) [][]int {

}
```

Kotlin:

```
class Solution {
fun queensAttacktheKing(queens: Array<IntArray>, king: IntArray):
List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func queensAttacktheKing(_ queens: [[Int]], _ king: [Int]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn queens_attackthe_king(queens: Vec<Vec<i32>>, king: Vec<i32>) ->
Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} queens
# @param {Integer[]} king
# @return {Integer[][]}
def queens_attackthe_king(queens, king)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $queens
* @param Integer[] $king
* @return Integer[][]
*/
function queensAttacktheKing($queens, $king) {

}
}
```

Dart:

```
class Solution {
List<List<int>> queensAttacktheKing(List<List<int>> queens, List<int> king) {

}
}
```

Scala:

```
object Solution {
def queensAttacktheKing(queens: Array[Array[Int]], king: Array[Int]):
List[List[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec queens_attackthe_king(queens :: [[integer]], king :: [integer]) ::
[[integer]]
def queens_attackthe_king(queens, king) do

end
end
```

Erlang:

```
-spec queens_attackthe_king(Queens :: [[integer()]], King :: [integer()]) ->
[[integer()]].
queens_attackthe_king(Queens, King) ->
.
```

Racket:

```
(define/contract (queens-attackthe-king queens king)
(-> (listof (listof exact-integer?)) (listof exact-integer?) (listof (listof
exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<vector<int>> queensAttacktheKing(vector<vector<int>>& queens,
vector<int>& king) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public List<List<Integer>> queensAttacktheKing(int[][] queens, int[] king) {

}
}
```

Python3 Solution:

```
"""
Problem: Queens That Can Attack the King
Difficulty: Medium
Tags: array

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def queensAttacktheKing(self, queens: List[List[int]], king: List[int]) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def queensAttacktheKing(self, queens, king):
"""
:type queens: List[List[int]]
:type king: List[int]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} queens
* @param {number[]} king
* @return {number[][]}
*/
var queensAttacktheKing = function(queens, king) {

};
```

TypeScript Solution:

```
/**
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function queensAttacktheKing(queens: number[][], king: number[]): number[][]
```

```
{

};
```

C# Solution:

```
/*
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public IList<IList<int>> QueensAttacktheKing(int[][] queens, int[] king) {

}
}
```

C Solution:

```
/*
* Problem: Queens That Can Attack the King
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** queensAttacktheKing(int** queens, int queensSize, int* queensColSize,
int* king, int kingSize, int* returnSize, int** returnColumnSizes) {
```

```
}
```

Go Solution:

```
// Problem: Queens That Can Attack the King
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func queensAttacktheKing(queens [][]int, king []int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun queensAttacktheKing(queens: Array<IntArray>, king: IntArray):
List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func queensAttacktheKing(_ queens: [[Int]], _ king: [Int]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Queens That Can Attack the King
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
```

```
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn queens_attackthe_king(queens: Vec<Vec<i32>>, king: Vec<i32>) ->
Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} queens
# @param {Integer[]} king
# @return {Integer[][]}
def queens_attackthe_king(queens, king)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $queens
* @param Integer[] $king
* @return Integer[][]
*/
function queensAttacktheKing($queens, $king) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> queensAttacktheKing(List<List<int>> queens, List<int> king) {

}
}
```

Scala Solution:

```
object Solution {
def queensAttacktheKing(queens: Array[Array[Int]], king: Array[Int]):
List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec queens_attackthe_king(queens :: [[integer]], king :: [integer]) ::
[[integer]]
def queens_attackthe_king(queens, king) do

end
end
```

Erlang Solution:

```
-spec queens_attackthe_king(Queens :: [[integer()]], King :: [integer()]) ->
[[integer()]].
queens_attackthe_king(Queens, King) ->
.
```

Racket Solution:

```
(define/contract (queens-attackthe-king queens king)
(-> (listof (listof exact-integer?)) (listof exact-integer?) (listof (listof
exact-integer?)))
)
```