Problem 3283: Maximum Number of Moves to Kill All Pawns

Problem Information

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

Problem Description

There is a

50 x 50

chessboard with

one

knight and some pawns on it. You are given two integers

kx

and

ky

where

(kx, ky)

denotes the position of the knight, and a 2D array

positions

where

positions[i] = [x

i

, y

i

]

denotes the position of the pawns on the chessboard.

Alice and Bob play a

turn-based

game, where Alice goes first. In each player's turn:

The player

selects

a pawn that still exists on the board and captures it with the knight in the

fewest

possible

moves

.

Note

that the player can select

any

pawn, it

might not

be one that can be captured in the

least

number of moves.

In the process of capturing the

selected

pawn, the knight

may

pass other pawns

without

capturing them

.

Only

the

selected

pawn can be captured in

this

turn.

Alice is trying to

maximize

the

sum

of the number of moves made by

both

players until there are no more pawns on the board, whereas Bob tries to

minimize

them.

Return the

maximum

total

number of moves made during the game that Alice can achieve, assuming both players play

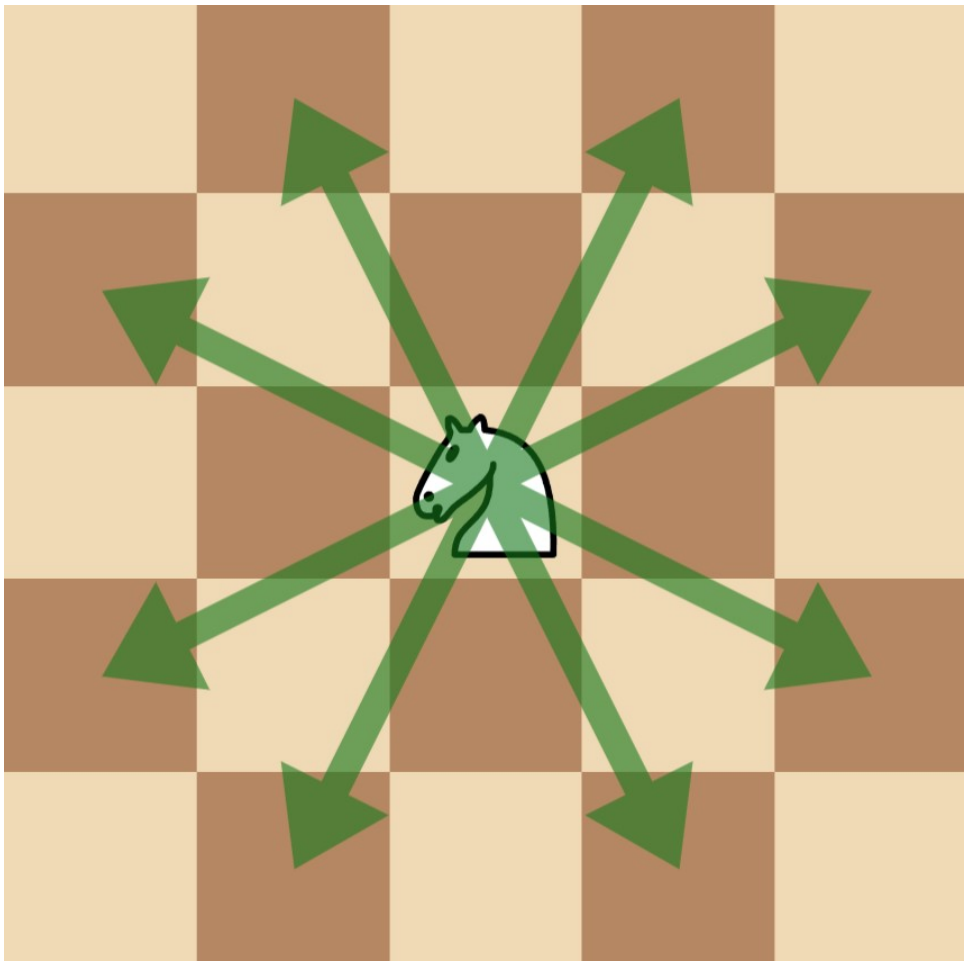
optimally

.

Note that in one

move,

a chess knight has eight possible positions it can move to, as illustrated below. Each move is two cells in a cardinal direction, then one cell in an orthogonal direction.



Example 1:

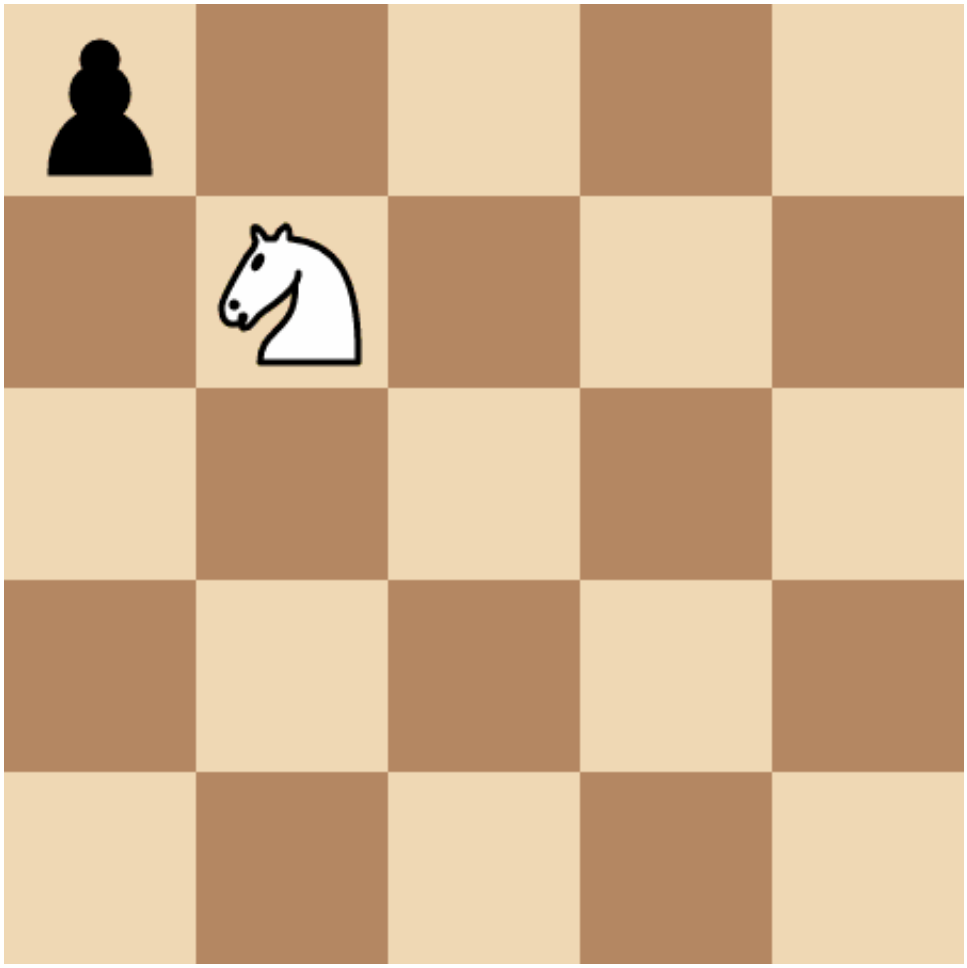
Input:

kx = 1, ky = 1, positions = [[0,0]]

Output:

4

Explanation:



The knight takes 4 moves to reach the pawn at

(0, 0)

.

Example 2:

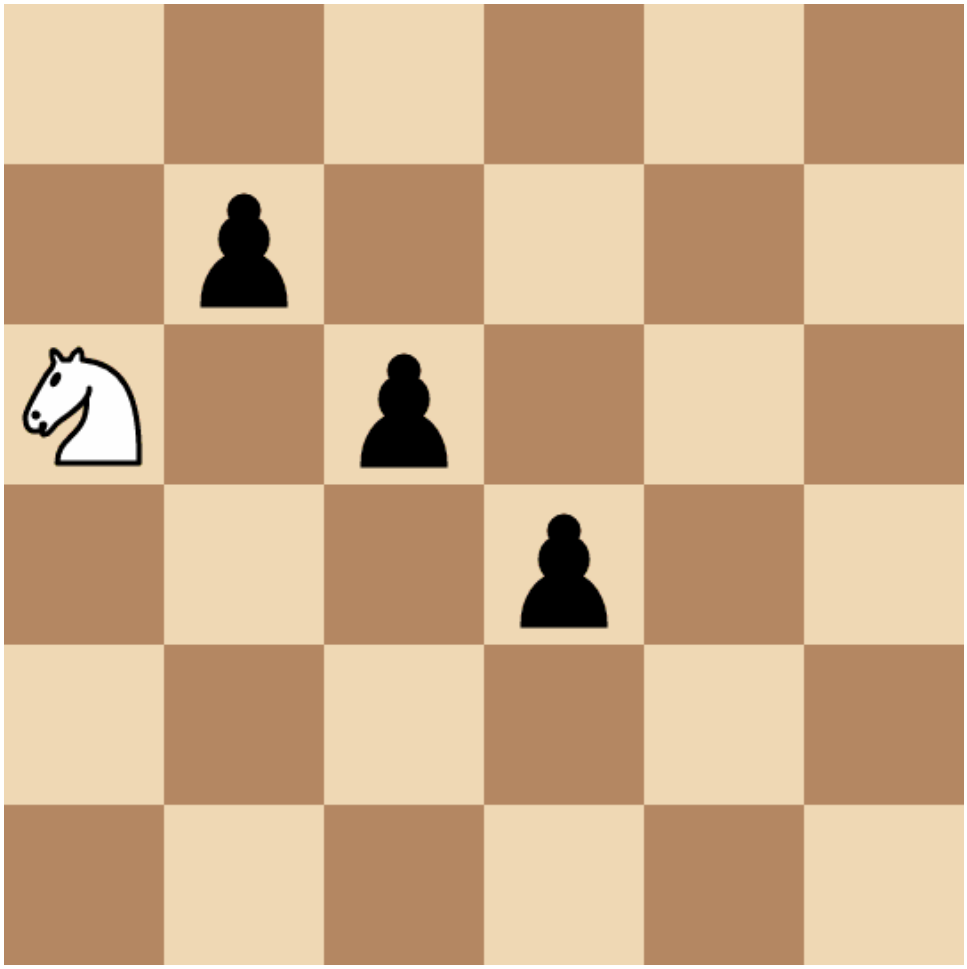
Input:

kx = 0, ky = 2, positions = [[1,1],[2,2],[3,3]]

Output:

8

Explanation:



Alice picks the pawn at

(2, 2)

and captures it in two moves:

(0, 2) -> (1, 4) -> (2, 2)

.

Bob picks the pawn at

(3, 3)

and captures it in two moves:

(2, 2) -> (4, 1) -> (3, 3)

.

Alice picks the pawn at

(1, 1)

and captures it in four moves:

(3, 3) -> (4, 1) -> (2, 2) -> (0, 3) -> (1, 1)

.

Example 3:

Input:

kx = 0, ky = 0, positions = [[1,2],[2,4]]

Output:

3

Explanation:

Alice picks the pawn at

(2, 4)

and captures it in two moves:

(0, 0) -> (1, 2) -> (2, 4)

. Note that the pawn at

(1, 2)

is not captured.

Bob picks the pawn at

(1, 2)

and captures it in one move:

(2, 4) -> (1, 2)

.

Constraints:

0 <= kx, ky <= 49

1 <= positions.length <= 15

positions[i].length == 2

0 <= positions[i][0], positions[i][1] <= 49

All

positions[i]

are unique.

The input is generated such that

positions[i] != [kx, ky]

for all

0 <= i < positions.length

.

Code Snippets

C++:

```
class Solution {
public:
int maxMoves(int kx, int ky, vector<vector<int>>& positions) {

}
};
```

Java:

```
class Solution {
public int maxMoves(int kx, int ky, int[][] positions) {

}
}
```

Python3:

```
class Solution:
def maxMoves(self, kx: int, ky: int, positions: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def maxMoves(self, kx, ky, positions):
"""
:type kx: int
:type ky: int
:type positions: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} kx
* @param {number} ky
* @param {number[][]} positions
* @return {number}
*/
var maxMoves = function(kx, ky, positions) {
```

```
};
```

TypeScript:

```
function maxMoves(kx: number, ky: number, positions: number[][]): number {

};
```

C#:

```
public class Solution {
public int MaxMoves(int kx, int ky, int[][] positions) {

}
}
```

C:

```
int maxMoves(int kx, int ky, int** positions, int positionsSize, int*
positionsColSize) {

}
```

Go:

```
func maxMoves(kx int, ky int, positions [][]int) int {

}
```

Kotlin:

```
class Solution {
fun maxMoves(kx: Int, ky: Int, positions: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func maxMoves(_ kx: Int, _ ky: Int, _ positions: [[Int]]) -> Int {
```

```
}
}
```

Rust:

```
impl Solution {
pub fn max_moves(kx: i32, ky: i32, positions: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} kx
# @param {Integer} ky
# @param {Integer[][]} positions
# @return {Integer}
def max_moves(kx, ky, positions)

end
```

PHP:

```
class Solution {

/**
* @param Integer $kx
* @param Integer $ky
* @param Integer[][] $positions
* @return Integer
*/
function maxMoves($kx, $ky, $positions) {

}
}
```

Dart:

```
class Solution {
int maxMoves(int kx, int ky, List<List<int>> positions) {

}
```

```
}
```

Scala:

```
object Solution {
def maxMoves(kx: Int, ky: Int, positions: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_moves(kx :: integer, ky :: integer, positions :: [[integer]]) ::
integer
def max_moves(kx, ky, positions) do

end
end
```

Erlang:

```
-spec max_moves(Kx :: integer(), Ky :: integer(), Positions :: [[integer()]])
-> integer().
max_moves(Kx, Ky, Positions) ->
.
```

Racket:

```
(define/contract (max-moves kx ky positions)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Number of Moves to Kill All Pawns
```

```
* Difficulty: Hard
* Tags: array, math, search
*
* 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:
int maxMoves(int kx, int ky, vector<vector<int>>& positions) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Number of Moves to Kill All Pawns
* Difficulty: Hard
* Tags: array, math, search
*
* 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 int maxMoves(int kx, int ky, int[][] positions) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Number of Moves to Kill All Pawns
Difficulty: Hard
Tags: array, math, search

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 maxMoves(self, kx: int, ky: int, positions: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxMoves(self, kx, ky, positions):
"""
:type kx: int
:type ky: int
:type positions: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Number of Moves to Kill All Pawns
* Difficulty: Hard
* Tags: array, math, search
*
* 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} kx
* @param {number} ky
* @param {number[][]} positions
* @return {number}
*/
var maxMoves = function(kx, ky, positions) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Number of Moves to Kill All Pawns
* Difficulty: Hard
* Tags: array, math, search
*
* 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 maxMoves(kx: number, ky: number, positions: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Number of Moves to Kill All Pawns
* Difficulty: Hard
* Tags: array, math, search
*
* 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 int MaxMoves(int kx, int ky, int[][] positions) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Number of Moves to Kill All Pawns
* Difficulty: Hard
* Tags: array, math, search
*
* 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
*/

int maxMoves(int kx, int ky, int** positions, int positionsSize, int*
positionsColSize) {

}
```

Go Solution:

```
// Problem: Maximum Number of Moves to Kill All Pawns
// Difficulty: Hard
// Tags: array, math, search
//
// 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 maxMoves(kx int, ky int, positions [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxMoves(kx: Int, ky: Int, positions: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func maxMoves(_ kx: Int, _ ky: Int, _ positions: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Number of Moves to Kill All Pawns
// Difficulty: Hard
// Tags: array, math, search
//
// 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 max_moves(kx: i32, ky: i32, positions: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} kx
# @param {Integer} ky
# @param {Integer[][]} positions
# @return {Integer}
def max_moves(kx, ky, positions)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $kx
* @param Integer $ky
* @param Integer[][] $positions
* @return Integer
*/
function maxMoves($kx, $ky, $positions) {

}
}
```

Dart Solution:

```
class Solution {
int maxMoves(int kx, int ky, List<List<int>> positions) {

}
}
```

Scala Solution:

```
object Solution {
def maxMoves(kx: Int, ky: Int, positions: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_moves(kx :: integer, ky :: integer, positions :: [[integer]]) ::
integer
def max_moves(kx, ky, positions) do

end
end
```

Erlang Solution:

```
-spec max_moves(Kx :: integer(), Ky :: integer(), Positions :: [[integer()]])
-> integer().
max_moves(Kx, Ky, Positions) ->
.
```

Racket Solution:

```
(define/contract (max-moves kx ky positions)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```