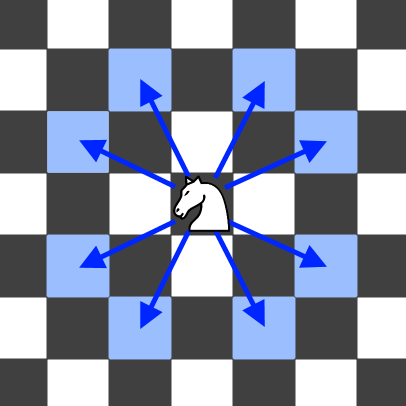
<https://leetcode.ca/2019-03-11-1197-Minimum-Knight-Moves/>

In an **infinite** chess board with coordinates from -infinity to +infinity, you have a **knight** at square [0, 0].

A knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.



Return *the minimum number of steps needed to move the knight to the square* [x, y]. It is guaranteed the answer exists.

**Example 1:**

**Input:** x = 2, y = 1

**Output:** 1

**Explanation:** [0, 0] → [2, 1]

**Example 2:**

**Input:** x = 5, y = 5

**Output:** 4

**Explanation:** [0, 0] → [2, 1] → [4, 2] → [3, 4] → [5, 5]

**Constraints:**

-300 <= x, y <= 300

0 <= |x| + |y| <= 300

**Attempt 1: 2025-04-30**

**Solution 1: BFS (30 min)**

**Refer to Deepseek**

**LeetCode 1197. Minimum Knight Moves - Java Solution**

This problem requires finding the minimum number of moves a knight needs to reach a target position on an infinite chessboard from the origin (0,0).

**Approach**

We can use a Breadth-First Search (BFS) approach since BFS naturally explores all positions level by level, which guarantees that the first time we reach the target position, we've done so with the minimum number of moves.

Key optimizations:

Symmetry: The problem is symmetric in all quadrants, so we can work with absolute values of coordinates

Boundary: We don't need to explore positions beyond |x| + 2 and |y| + 2 of the target

Visited tracking: To avoid revisiting the same positions

**Solution Code**

class Solution {

public int minKnightMoves(int x, int y) {

// Symmetry allows us to work in the first quadrant only

x = Math.abs(x);

y = Math.abs(y);

// Possible moves for a knight

int[][] moves = {{1,2},{2,1},{1,-2},{2,-1},{-1,2},{-2,1},{-1,-2},{-2,-1}};

Queue<int[]> queue = new LinkedList<>();

queue.add(new int[]{0, 0});

Set<String> visited = new HashSet<>();

visited.add("0,0");

int steps = 0;

while (!queue.isEmpty()) {

int size = queue.size();

for (int i = 0; i < size; i++) {

int[] current = queue.poll();

int currX = current[0];

int currY = current[1];

if (currX == x && currY == y) {

return steps;

}

for (int[] move : moves) {

int newX = currX + move[0];

int newY = currY + move[1];

// Check if the new position is within bounds and not visited

// The bounds check helps prevent infinite expansion

if (!visited.contains(newX + "," + newY) && newX >= -1 && newY >= -1) {

visited.add(newX + "," + newY);

queue.add(new int[]{newX, newY});

}

}

}

steps++;

}

return -1; // Shouldn't reach here for valid inputs

}

}

**Explanation**

Symmetry Handling: We convert the target coordinates to their absolute values since the problem is symmetric in all quadrants.

BFS Initialization: We start from the origin (0,0) and use a queue to explore all possible moves level by level.

Knight Moves: The knight has 8 possible moves from any position, which we represent as coordinate changes.

Visited Tracking: We maintain a set of visited positions to avoid cycles and redundant processing.

Level-wise Processing: For each level (representing a move count), we process all positions at that level before moving to the next.

Termination Condition: When we reach the target position, we return the current move count (steps).

This approach efficiently explores the chessboard using BFS, guaranteeing the minimum number of moves while avoiding unnecessary exploration through symmetry and bounds checking.

**Elaborate on "We convert the target coordinates to their absolute values since the problem is symmetric in all quadrants." ?**

**Understanding Symmetry in the Knight's Moves Problem**

When solving the "Minimum Knight Moves" problem, we can take advantage of the symmetry of a chessboard to simplify our solution. Here's a detailed explanation of why converting the target coordinates to their absolute values works and how it helps optimize the BFS approach.

**1. Symmetry in Chessboard Movement**

A chessboard is symmetric in all four quadrants (I, II, III, IV). This means:

The movement rules for a knight are identical in all directions.

The minimum number of moves required to reach (x, y) is the same as reaching (|x|, |y|) (the absolute values).

**Example:**

Moving from (0, 0) to (2, 1) takes the same number of moves as moving to (-2, 1), (2, -1), or (-2, -1) because the knight's possible moves are mirrored across the axes.

**2. Why Convert to Absolute Values?**

By converting (x, y) to (|x|, |y|), we:

Reduce the Problem Size: Instead of considering all four quadrants, we only need to work in the \*\*first quadrant (where both x and y are ≥ 0)\*\*.

Avoid Redundant Checks: Without symmetry, we would have to explore mirrored positions separately, increasing the BFS search space unnecessarily.

Simplify Boundary Conditions: Since we now work only in the first quadrant, we can set bounds more easily (e.g., ensuring newX ≥ -1 and newY ≥ -1).

**3. How Symmetry Optimizes BFS**

Original Problem: The knight can move in 8 directions, leading to an exponentially growing search space.

With Symmetry: By focusing only on (|x|, |y|), we effectively reduce the search space by a factor of 4 (since we ignore mirrored positions).

**Example:**

If the target is (3, 5), we convert it to (3, 5) (already in Q1).

If the target is (-3, 5), we convert it to (3, 5).

If the target is (-3, -5), we convert it to (3, 5).

The BFS will now compute moves only for (3, 5) instead of all four variants.

**4. Does This Affect Correctness?**

No, because:

The knight's moves are symmetric (e.g., moving (+2, +1) is the same as (-2, +1) in terms of distance).

The minimum path length remains the same regardless of direction.

**Example:**

Path to (2, 1):

(0,0) → (2,1) (2 right, 1 up) → 1 move.

Path to (-2, 1):

(0,0) → (-2,1) (2 left, 1 up) → 1 move (same distance).

Thus, working with (|x|, |y|) does not change the result but makes computation more efficient.

**5. Edge Case Handling**

(0, 0): The knight starts here, so moves needed = 0.

(1, 0) or (0, 1): A knight cannot reach these in one move (it must move in an L-shape), so the minimum moves are higher (e.g., (0,0) → (2,1) → (0,2) → (1,0) → 3 moves).

Negative Targets: Handled by symmetry (e.g., (-1, -1) becomes (1, 1)).

**6. Why Allow newX ≥ -1 and newY ≥ -1 in BFS?**

**The knight might temporarily move to (-1, -1) while trying to reach (1, 1) (e.g., (0,0) → (2,1) → (1,-1) → (-1,0) → (1,1)).**

**Restricting to newX ≥ 0 and newY ≥ 0 could miss valid paths, so we relax the bounds slightly.**

**Final Takeaway**

By exploiting symmetry and converting (x, y) to (|x|, |y|), we:

Reduce redundant computations (BFS explores fewer unique positions).

Maintain correctness (minimum moves remain the same).

Simplify implementation (only need to track one quadrant).

This optimization is crucial for handling large coordinates efficiently (e.g., x or y = 300). Without it, the BFS could time out due to excessive branching.

Would you like a step-by-step walkthrough of how BFS processes an example like (2, 1)?