Title

Reasoning

To solve the problem, we should first encode every possible state in a graph. Each node of the graph is a state of the things (x_1, y_1, x_2, y_2) and each edge represents a valid move to another valid state. Once we have made this graph, we can use breadth first search to find the exit path (or whether there is one).

While writing the psudocode, I realized we can also put the graph building inside the breadth first search, simplifying the code a lot, although the theory still stays the same.

Psudocode

```
def move (x, y, dx, dy):
    possible_x = x+dx
    possible_y = y + dy
    if (0 <= possible_x < a and 0 <= possible_y < b) and maze[possible_x]
[possible_y] is valid:
        return (possible_x, possible_y)
    return (x, y)

def try_exit(x, y, dx, dy):
    px = x+dx
    py = y+dy
    return !(0<=px<a and 0<=py<b) // Return if the new point is outside the
bounds of the maze

def things_escape_house(maze: 2d array of characters, start1: int[2],
start2: int[2]):
    queue = [ (start1[0], start1[1], start2[0], start2[1], 0) ]
    visited = [ (start1[0], start1[1], start2[0], start2[1]) ]

    while queue:
        x1, y1, x2, y2, dist = queue.popleft()

        for dx, dy in the cardinal directions:
            if is_exit(x1, y1, dx, dy) and is_exit(x2, y2, dx, dy):
                return dist + 1

            px1, py1 = move(x1, y1, dx, dy)
            px2, py2 = move(x2, y2, dx, dy)

            # handles both moving or colliding
            if (px1, py1) == (px2, py2) and not ((px1, py1) == (x2, y2) and
(px2, py2) == (x1, y1)):
                continue

            state = (px1, py1, px2, py2)
            if state not in visited:
                visited.add(state)
                queue.append((px1, py1, px2, py2, dist+1))

    return impossible #maybe use a negative number or make a custom return
class
```

Proof

This algorithm works because if you have made a graph of all possible game positions, you can then use breadth first search, which will either give you the shortest path to the exit by definition, or will fail, telling you that there is no way out of the house.

Running Time

Estimate

The running time is $O((ab)^2)$

Running Time Reasoning

Reasoning Here

```
def move (x, y, dx, dy): # O(1)
    possible_x = x+dx
    possible_y = y + dy
    if (0 <= possible_x < a and 0 <= possible_y < b) and maze[possible_x]
[possible_y] is valid:
        return (possible_x, possible_y)
    return (x, y)

def try_exit(x, y, dx, dy): # O(1)
    px = x+dx
    py = y+dy
    return !(0<=px<a and 0<=py<b) // Return if the new point is outside the
bounds of the maze

def things_escape_house(maze: 2d array of characters, start1: int[2],
start2: int[2]):
    queue = [ (start1[0], start1[1], start2[0], start2[1], 0) ] # O(1)
    visited = [ (start1[0], start1[1], start2[0], start2[1]) ] # O(1)

    while queue: # O(aabb)
        x1, y1, x2, y2, dist = queue.popleft() # O(1)

        for dx, dy in the cardinal directions: # O(4) => O(1)
            if is_exit(x1, y1, dx, dy) and is_exit(x2, y2, dx, dy): # O(1)
                return dist + 1 # O(1)

            px1, py1 = move(x1, y1, dx, dy) # O(1)
            px2, py2 = move(x2, y2, dx, dy) # O(1)

            # handles both moving or colliding
            if (px1, py1) == (px2, py2) and not ((px1, py1) == (x2, y2) and
(px2, py2) == (x1, y1)): # O(1)
                skip iteration

            state = (px1, py1, px2, py2) # O(1)
            if state not in visited: # O(1)
                visited.add(state) # O(1)
                queue.append((px1, py1, px2, py2, dist+1)) # O(1)

    return impossible # O(1)
```

So the final running time is $O((ab)^2)$