# 3015. Count the Number of Houses at a Certain Distance I



## Problem Description

In the given problem, we have a linear city with houses numbered from 1 to n. Each adjacent house is connected with a street, resulting in there being n - 1 such streets connecting houses from 1 to 2, 2 to 3, and so on, up to n - 1 to n. Additionally, there is another street that directly connects two specific houses, x and y, which makes it possible to travel between these two houses without having to pass through the ones in between.

The task is to find out, for every possible distance k (ranging from 1 to n), how many unique pairs of houses (house$_1$, house$_2$) require exactly k streets to travel from one to the other. It is important to note that the distance between the houses must be the minimum possible, taking into account the direct route between houses x and y if it provides a shortcut. The problem asks for the result to be in a list where the index (1-indexed) corresponds to the number of streets k, and the value at that index is the count of house pairs that are k streets apart.

## Intuition

To solve this problem, we need to consider all possible pairs of houses. For each pair of houses (i, j), there are three possible paths we need to evaluate:

1. The direct path from i to j via the connecting streets (the distance is `|i - j|`).
2. The path from i to house x, then using the direct street to y, and finally from y to j (the distance is `|i - x| + 1 + |j - y|`).
3. The path from i to house y, then using the direct street to x, and finally from x to j (the distance is `|i - y| + 1 + |j - x|`).

We then take the minimum of these three distances as the actual minimum distance between house i and house j. We increment the count for this distance twice since the pair (i, j) and the pair (j, i) are distinct according to the problem's condition.

The solution approach efficiently performs these calculations for every pair of houses and accumulates the number of pairs for each distance in an array. The final result gives the desired count of house pairs for each possible distance k from 1 to n.

## Solution Approach

The solution is implemented using a simple brute-force approach that considers all pairs of houses, and for each pair, it calculates the minimum number of streets needed to travel from one house to the other. This is done using two nested loops that iterate over all possible house combinations where the outer loop represents the starting house `i` and the inner loop represents the destination house `j`.

The code uses simple arithmetic calculations to determine the three possible distances for each pair `(i, j)`:

1. The direct distance `a`, which is simply the absolute difference between `i` and `j`: `|i - j|`.
2. The distance `b` that goes from `i` to `x`, then takes the direct road from `x` to `y`, and finally goes from `y` to `j`: `|i - x| + 1 + |j - y|`.
3. The distance `c` that goes from `i` to `y`, then takes the direct road from `y` to `x`, and finally goes from `x` to `j`: `|i - y| + 1 + |j - x|`.

After calculating these distances, the solution finds the minimum of the three distances using the built-in `min` function. This minimum represents the fewest number of streets needed to reach from one house to the other, considering the special direct connection between houses `x` and `y`.

The count for this minimum distance is updated by adding 2 to the corresponding index in the answer array `ans`, accounting for both `(i, j)` and `(j, i)` pairs. We subtract 1 from the minimum distance before using it as an index, because the array is 1-indexed as per the problem statement.

No special data structures or advanced algorithms are needed, as the approach relies on straightforward enumeration and counting based on distance calculations.

```
class Solution:
    def countOfPairs(self, n: int, x: int, y: int) -> List[int]:
        x, y = x - 1, y - 1  # Adjusting to 0-based index
        ans = [0] * n  # Initializing the answer array with zeros
        for i in range(n):
            for j in range(i + 1, n):
                a = j - i  # Direct distance
                b = abs(i - x) + 1 + abs(j - y)  # Distance via x to y
                c = abs(i - y) + 1 + abs(j - x)  # Distance via y to x
                ans[min(a, b, c) - 1] += 2  # Update the count for the minimum distance
        return ans
```

This approach gives us a solution that is easy to understand and implement. However, it has a time complexity of O(n^2) since we're iterating over each pair of houses, making it potentially inefficient for very large values of `n`.

## Example Walkthrough

Let's assume we have a linear city with `n = 5` houses and a direct street connecting houses `x = 2` and `y = 5`. We want to know how many unique pairs of houses require exactly `k` streets to travel from one to the other, for each `k` from `1` to `n`.

Using the brute force approach outlined, we would perform the following steps:

1. Initialize the array `ans` to `[0, 0, 0, 0, 0]`, which will hold the count of unique house pairs for each distance `k` from 1 to 5.
2. Begin iterating over all possible pairs of houses `(i, j)` where `i < j`, using two nested loops.

Here are the calculations for each house pair:

- Pair `(i=1, j=2)`: The direct distance is `1`, and since one of the houses is `x`, we don't need to consider other paths. Thus, we will increment the count at `ans[0]` by 2 as the pair `(1, 2)` and `(2, 1)` will both have 1 as their minimum distance.
- Pair `(i=1, j=3)`: The direct distance is 2. The distance via `x` and `y` (houses 2 and 5) is `1 + 1 + 2 = 4` and the distance via `y` to `x` is `4 + 1 + 1 = 6`, so the minimum is the direct distance 2. We increment `ans[1]` by 2.
- Pair `(i=1, j=4)`: The direct distance is 3. The distance via `x` to `y` is `1 + 1 + 1 = 3`, and the distance via `y` to `x` is `4 + 1 + 2 = 7`. Both direct distance and via `x` to `y` are 3, so `ans[2]` is incremented by 2.
- Pair `(i=1, j=5)`: The direct distance is 4. However, since `j` is `y`, and there is a direct shortcut from `i` to `y` (`x` to `y`), we consider the distance `1 + 1 = 2` using the direct road. Thus, `ans[1]` is incremented by 2 again.
- Pair `(i=2, j=3)`, `(i=2, j=4)`, and `(i=3, j=4)` follow similar calculations using either the direct distance or the special road.
- Pair `(i=2, j=5)` uses the direct road between `x` and `y`, resulting in the distance being `1`. So, `ans[0]` is incremented by 2.
- Pair `(i=3, j=5)` has a direct road distance of `2` (using the `x` to `y` direct connection), which is smaller than the direct distance of `3`, so `ans[1]` is incremented.
- Pair `(i=4, j=5)` is straightforward as house `4` is next to `y` (house `5`), resulting in a minimum distance of `1`. Hence, `ans[0]` is incremented.

After running these calculations, the `ans` array becomes `[4, 6, 2, 0, 0]`, which means:

- There are 4 unique house pairs that are exactly 1 street apart.
- There are 6 unique house pairs that are exactly 2 streets apart.
- There are 2 unique house pairs that are exactly 3 streets apart.
- There are no house pairs that are exactly 4 or 5 streets apart.

Thus, the result for this example would be `[4, 6, 2, 0, 0]`.

## Solution Implementation

### Python

```
from typing import List

class Solution:
    def count_of_pairs(self, n: int, x: int, y: int) -> List[int]:
        # Decrement x and y to switch from 1-based to 0-based indexing
        x, y = x - 1, y - 1

        # Initialize a list to store the counts of pairs
        count_list = [0] * n

        # Iterate over the range to find the distances for each pair (i, j)
        for i in range(n):
            for j in range(i + 1, n):  # Ensure j > i
                # Calculate the direct distance between i and j
                direct_distance = j - i

                # Calculate the distance when passing through x and y
                detour_x_y_distance = abs(i - x) + 1 + abs(j - y)
                detour_y_x_distance = abs(i - y) + 1 + abs(j - x)

                # Choose the minimum of the three distances
                min_distance = min(direct_distance, detour_x_y_distance, detour_y_x_distance)

                # Increment the count of pairs for the determined minimum distance
                # Since pairs are counted twice (i,j) and (j,i), we increase the count by 2.
                count_list[min_distance - 1] += 2

        # Return the final count list
        return count_list
```

### Java

```
class Solution {
    public int[] countOfPairs(int n, int x, int y) {
        // Initialize an array to hold the count of pairs for each distance
        int[] answer = new int[n];

        // Adjust the positions of 'x' and 'y' to be zero-indexed
        x--;
        y--;

        // Iterate over all possible pairs (i, j)
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                // Calculate the direct distance 'a' between points i and j
                int directDistance = j - i;

                // Calculate distance 'b' as the sum of the distance from i to x, then from x to y
                // with an additional step from x to y
                int indirectDistanceX = Math.abs(i - x) + 1 + Math.abs(j - y);

                // Calculate distance 'c' as the sum of the distance from i to y, then from y to j
                // with an additional step from y to x
                int indirectDistanceY = Math.abs(i - y) + 1 + Math.abs(j - x);

                // Determine the smallest of the three distances
                int minDistance = Math.min(directDistance, Math.min(indirectDistanceX, indirectDistanceY));

                // Increment the count of pairs for the determined smallest distance
                // Each pair contributes to two such distances, hence the increment by 2
                answer[minDistance - 1] += 2;
            }
        }

        // Return the array containing the count of pairs for each distance
        return answer;
    }
}
```

### C++

```
#include <vector>
#include <cmath>
#include <algorithm>

class Solution {
public:
    // Function to calculate the count of pairs with various minimum distances between two players on a linear board
    std::vector<int> countOfPairs(int n, int x, int y) {
        std::vector<int> answer(n); // Initialize a vector to store the counts for each distance
        x--; // Adjust x index to be zero based
        y--; // Adjust y index to be zero based
        // Iterate over all pairs to calculate minimum distances
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                // Scenario 1: Direct distance between positions i and j
                int directDistance = j - i;
                // Scenario 2: Distance when going through position x first, then to position y
                int viaXYDistance = std::abs(x - i) + std::abs(y - j) + 1;
                // Scenario 3: Distance when going through position y first, then to position x
                int viaYXDistance = std::abs(y - i) + std::abs(x - j) + 1;

                // Determine the minimum of the three calculated distances
                int minDistance = std::min({directDistance, viaXYDistance, viaYXDistance}) - 1;
                // Increment the count for this minimum distance by 2 since pair (i, j) and (j, i) are counted as two
                answer[minDistance] += 2;
            }
        }
        return answer; // Return the result vector with counts for each distance
    }
};
```

### TypeScript

```
function countOfPairs(n: number, x: number, y: number): number[] {
    // Initialize an array for the answer with size `n`, filled with 0s.
    const answerArray: number[] = new Array(n).fill(0);

    // Decrement `x` and `y` to convert them to zero-based indices.
    x--;
    y--;

    // Iterate through each possible pair of positions.
    for (let i = 0; i < n; ++i) {
        for (let j = i + 1; j < n; ++j) {
            // Calculate the direct distance between positions i and j.
            const directDistance = j - i;

            // Calculate the distance from position i to x, then from x to j (including x).
            const distanceViaX = Math.abs(x - i) + Math.abs(y - j) + 1;

            // Calculate the distance from position i to y, then from y to j (including y).
            const distanceViaY = Math.abs(y - i) + Math.abs(x - j) + 1;

            // Find the minimum distance and use it to update the answer array.
            // Since using one-based indexing for distances in answer array, subtract 1.
            answerArray[Math.min(directDistance, distanceViaX, distanceViaY) - 1] += 2;
        }
    }

    // Return the array containing the count of the minimum distances.
    return answerArray;
}


from typing import List

class Solution:
    def count_of_pairs(self, n: int, x: int, y: int) -> List[int]:
        # Decrement x and y to switch from 1-based to 0-based indexing
        x, y = x - 1, y - 1

        # Initialize a list to store the counts of pairs
        count_list = [0] * n

        # Iterate over the range to find the distances for each pair (i, j)
        for i in range(n):
            for j in range(i + 1, n):  # Ensure j > i
                # Calculate the direct distance between i and j
                direct_distance = j - i

                # Calculate the distance when passing through x and y
                detour_x_y_distance = abs(i - x) + 1 + abs(j - y)
                detour_y_x_distance = abs(i - y) + 1 + abs(j - x)

                # Choose the minimum of the three distances
                min_distance = min(direct_distance, detour_x_y_distance, detour_y_x_distance)

                # Increment the count of pairs for the determined minimum distance
                # Since pairs are counted twice (i,j) and (j,i), we increase the count by 2.
                count_list[min_distance - 1] += 2

        # Return the final count list
        return count_list
```

## Time and Space Complexity

The given Python code includes nested loops where the outer loop runs `n` times and the inner loop runs up to `n-1` times in the worst case. This setup leads to a time complexity of `O(n^2)` because each element is compared with each other element once. Specifically, for each `i` in the range `[0, n)`, `j` will iterate from `i+1` to `n`. Therefore, the total number of iterations is the sum of the series from `1` to `n-1`, which is `(n(n-1))/2`, representing a quadratic time complexity.

The space complexity of the code, excluding the output list `ans`, is `O(1)`. That is because the algorithm uses a constant amount of extra space regardless of the input size `n`. The variables `x`, `y`, `i`, `j`, `a`, `b`, and `c` are only a finite set of pointers and calculations occurring within the loops, and their memory usage does not scale with `n`. The answer array `ans`, although of size `n`, is typically not considered in the space complexity analysis as it is required for the output of the function. However, if we were to include `ans` in the space complexity, it would become `O(n)` as the space required would scale linearly with the input size `n`.