765. Couples Holding Hands

Hard · Greedy · Depth-First Search · Breadth-First Search · Union Find · Graph · Leetcode Link

Problem Description

In the given problem, we are presented with `n` couples (making a total of `2n` individuals) arranged in a row of `2n` seats. Each couple is represented by a pair of consecutive numbers, such as `(0, 1)`, `(2, 3)`, and so on up to `(2n - 2, 2n - 1)`. The arrangement of the individuals is provided in the form of an integer array called `row`, where `row[i]` is the ID number of the person sitting in the `i`th seat.

The goal is to make every couple sit next to each other with the least amount of swapping actions. A swap action is defined as taking any two individuals and having them switch seats with each other.

The challenge is to calculate the minimum number of swaps required to arrange all couples side by side.

Intuition

The intuition behind the solution is to treat the problem as a graph problem where each couple pairing is a connected component. We can build this conceptual graph by considering each couple as a node and each incorrect pair as an edge connecting the nodes. The size of a connected component then represents the number of couples in a mixed group that need to be resolved.

The key insight is to observe that within a mixed group, when you arrange one pair properly, it will automatically reduce the number of incorrect pairs by one since you reduce the group size. It's akin to untangling a set of connected strings; each time you properly untangle a string from the group, the complexity decreases.

We perform union-find operations to group the couples into connected components. The "find" operation is used to determine the representative (or the root) of a given couple, while the "union" (in this case, an assignment in the loop) combines two different couple nodes into the same connected component. This way, each swap connects two previously separate components, thereby reducing the total number of components. The answer to the minimum number of swaps is the initial number of couples `n` minus the number of connected components after the union-find process.

The reasoning is that for each connected component that is properly arranged, only (size of the component - 1) swaps are needed (consider that when you position the last couple correctly, no swap is needed for them).

The `minSwapsCouples` function thus iterates over the row array, pairing individuals with their partners by dividing their ID by 2 (since couples have consecutive numbers), and uses union-find to determine the minimum number of swaps required.

Solution Approach

The implementation of the solution approach for the problem of arranging couples uses the Union-Find algorithm, which is a classic algorithm in computer science used to keep track of elements which are split into one or more disjoint sets. Its primary operations are "find", which finds the representative of a set, and "union", which merges two sets into one.

Here's a step-by-step explanation of how the implementation works:

1. **Initialization**: First, we initialize a list `p` which will represent the parent (or root) of each set. Initially, each set has only one element, so the parent of each set is the element itself. In our case, each set is a couple, and there are `n` couples indexed from `0` to `n-1`. Hence, `p` is initialized with range `n`.

```python
n = len(row) >> 1
p = list(range(n))
```

2. **Finding the root**: The `find` function is a recursive function that finds the representative of a given element `x`. The representative (or parent) of a set is the common element shared by all elements in the set. The `find` function compresses the path, setting the `p[x]` to the root of `x` for faster future lookups.

```python
def find(x: int) -> int:
    if p[x] != x:
        p[x] = find(p[x])
    return p[x]
```

3. **Merging sets**: As we iterate through the row, we take two adjacent people `row[i]` and `row[i + 1]`. We find the set representative (root) of their couple indices, which are `row[i] >> 1` and `row[i + 1] >> 1` (bitwise right shift operator to divide by 2). We then merge these sets by assigning one representative to another. This is an application of the "union" operation.

```python
for i in range(0, len(row), 2):
    a, b = row[i] >> 1, row[i + 1] >> 1
    p[find(a)] = find(b)
```

4. **Count minimum swaps**: Finally, the number of swaps needed equals to the total number of couples `n` minus the number of sets that represent correctly paired couples. This calculation is done by counting the number of times `i` is the representative of the set it belongs to (`i == find(i)`). This line of code effectively counts the number of connected components after union-find.

```python
return n - sum(i == find(i) for i in range(n))
```

Mathematically, each set needs `size of the set - 1` swaps to arrange the couple properly. The sum hence gives us a total of how many sets were already properly paired initially, by subtracting this sum from `n` we get the total number of swaps.

This approach is efficient because Union-Find has nearly constant time (amortized) complexity for both union and find operations, which allows us to solve the problem in almost linear time relative to the number of people/couples.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `n = 2` couples, so a total of `4` individuals. Our `row` array representing the seating arrangement is: `[1, 0, 3, 2]`. This means the first person (1) is paired with the second person (0), and the third person (3) is paired with the fourth person (2). But we want to arrange them such that person `0` is next to `1` and `2` is next to `3`.

Using the mentioned approach, let's walk through the steps:

Step 1: Initialization We know `n = 2`. Initialize parent list `p` with the indices representing the couples (0-based).

```
n = 2
p = [0, 1]
```

Step 2: Finding the root The `find` function will determine the representative of each set. As we haven't made any unions yet, each set's representative is the set itself.

Step 3: Merging sets The iteration goes as follows:

- First pair is (1, 0). Their couple indices are `1 >> 1 = 0` and `0 >> 1 = 0`. Since both are the same, no union is needed.
- Second pair is (3, 2). Their couple indices are `3 >> 1 = 1` and `2 >> 1 = 1`. Again, no union is needed.

Now comes the merging by using the union operation:

- We set the representative of the first person's couple to be the same as the second person's couple: `p[find(0)] = find(1)`.
- However, both are already correct as `p[0]` is `0` and `find(1)` is `1`.
- No changes occur to `p`, it remains `[0, 1]`.

Step 4: Count minimum swaps We have `n = 2` couples, we go through the parent list `p` and count how many times `i` is equal to `find(i)`.

- For `i = 0`, `find(0)` returns `0`, so there's 1 correct pair.
- For `i = 1`, `find(1)` returns `1`, so there's another correct pair.

Thus, the sum of correct pairs is `2`. The minimum swaps needed are `n - sum` which is `2 - 2 = 0`. But in the given example, we see that they are not seated correctly. However, according to our merged sets, there should be no swaps because both pairs are already with their respective partners but only in the wrong order, so our process does include the final swap to make the order correct.

We now see there is a discrepancy. The example also illustrates that the merging of the representatives doesn't fully capture the physical seating arrangement - it just ensures elements are in the same set. In this example, a single swap between 0 and 3 is needed to arrange the couples correctly. So, the expected output for swaps is `1`, and the final array should look like this `[1, 0, 2, 3]`.

This walkthrough shows that while our parents' array after the `find` operation accurately identifies the sets, we still need to consider the physical swaps between these sets to achieve the proper arrangement. Thus, the minimum number of swaps calculated will be `1` and we get our desired seating `[1, 0, 2, 3]`.

Python Solution

```python
class Solution:
    def minSwapsCouples(self, row: List[int]) -> int:

        # Helper function to find the root parent in the disjoint set.
        def find_root_couple(couple: int) -> int:
            if parent[couple] != couple:
                # Recursively find the root couple and perform path compression.
                parent[couple] = find_root_couple(parent[couple])
            return parent[couple]

        # Number of couples is half the length of the row.
        num_couples = len(row) // 2
        # Initialize parent array for disjoint set union-find data structure.
        parent = list(range(num_couples))

        # Iterate through every second index (i.e., every couple)
        for i in range(0, len(row), 2):
            # Bitwise right shift by 1 to find the index of the couple
            first_couple, second_couple = row[i] // 2, row[i + 1] // 2

            # Union the two couples in the disjoint set.
            parent[find_root_couple(first_couple)] = find_root_couple(second_couple)

        # Calculate the number of swap operations needed.
        # It's equal to the number of couples - number of disjoint set roots.
        return num_couples - sum(i == find_root_couple(i) for i in range(num_couples))

# Explanation:
# The `minSwapsCouples` function determines the minimum number of swaps required
# to arrange a row of couples such that each pair sits together.
# "row" is an even-length list where couples are represented by consecutive integers.
# `find_root_couple` is a helper function that uses the union-find algorithm to manage
# disjoint sets of couples. This function helps in identifying the connected components
# (couples sitting together) by finding the root of each disjoint set.
# The main loop pairs up partners by union operations on the disjoint set.
# The final result is calculated by taking the total number of couples and subtracting
# the number of unique roots, which gives the number of swaps needed to sort the pairs.

```

Java Solution

```java
class Solution {
    private int[] parent;

    // Function that takes an array representing couples sitting in a row
    // and returns the minimum number of swaps to make all couples sit together.
    public int minSwapsCouples(int[] row) {
        int numOfCouples = row.length >> 1; // Determine the number of couples
        parent = new int[numOfCouples];
        // Initialize the union-find structure
        for (int i = 0; i < numOfCouples; ++i) {
            parent[i] = i;
        }
        // Join couples in the union-find structure
        for (int i = 0; i < numOfCouples << 1; i += 2) {
            int partner1 = row[i] >> 1;
            int partner2 = row[i + 1] >> 1;
            // union operation: join the two sets that contain partner1 and partner2
            parent[find(partner1)] = find(partner2);
        }
        int swaps = numOfCouples; // Start with a maximum possible swap count
        // Count the number of unique sets, which equals minimum number of swaps
        for (int i = 0; i < numOfCouples; ++i) {
            if (i == find(i)) {
                swaps--;
            }
        }
        return swaps;
    }

    // Helper function that finds the root of x using path compression
    private int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <numeric> // for std::iota
#include <functional> // for std::function

class Solution {
public:
    // Function to solve the couples holding hands problem with the minimum number of swaps.
    int minSwapsCouples(std::vector<int>& row) {
        // The number of couples is half the number of people in the row
        int numCouples = row.size() / 2;

        // Parent array for union-find data structure
        std::vector<int> parent(numCouples);

        // Initialize parent array to self indices
        std::iota(parent.begin(), parent.end(), 0);

        // Helper function to find root of the element in union-find data structure
        std::function<int(int)> findRoot = [&](int x) -> int {
            if (parent[x] != x) {
                parent[x] = findRoot(parent[x]);
            }
            return parent[x];
        };

        // Loop through each pair in the row and unify the pairs
        for (int i = 0; i < row.size(); i += 2) {
            // Each person belongs to couple numbered as person's index divided by 2
            int coupleA = row[i] / 2;
            int coupleB = row[i + 1] / 2;
            // Union the roots of the two couples
            parent[findRoot(coupleA)] = findRoot(coupleB);
        }

        // Count the number of groups (connected components) that are already seated correctly
        int correctGroups = 0;
        for (int i = 0; i < numCouples; ++i) {
            correctGroups += i == findRoot(i);
        }

        // Subtract the correctly seated groups from the total number of couples
        // to get the minimum necessary swaps
        int minSwaps = numCouples - correctGroups;

        return minSwaps;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the minimum number of swaps needed to arrange
// pairs of couples consecutively
function minSwapsCouples(row: number[]): number {
    const pairsCount = row.length >> 1;
    // Parent array for union-find, initialized to self references
    const parent: number[] = Array.from({ length: pairsCount }, (_, i) => i);

    // Find function for union-find with path compression
    const find = (x: number): number => {
        if (parent[x] !== x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    };

    // Iterate over pairs and apply union-find
    for (let i = 0; i < row.length; i += 2) {
        const firstPartner = row[i] >> 1;
        const secondPartner = row[i + 1] >> 1;
        // Union step: assign the parent of the first partner to the parent of the second
        parent[find(firstPartner)] = find(secondPartner);
    }

    let swapCount = pairsCount;
    // Count the number of connected components, one swap is needed for each component minus one
    for (let i = 0; i < pairsCount; i++) {
        if (i === find(i)) {
            swapCount--;
        }
    }

    // Return the total number of swaps required
    return swapCount;
}

```

Time and Space Complexity

The given code implements a function to count the minimum number of swaps required to arrange couples in a row. It uses a union-find algorithm to merge the couple pairs. Here's the analysis:

Time Complexity:

The time complexity of this function is dominated by two parts, the union-find operations (`find` function) and the loop that iterates over the row to perform unions.

- The `find` function has an amortized time complexity of `O(α(n))`, where `α` is the inverse Ackermann function, which grows extremely slowly and is considered practically constant for modest input sizes.
- The loop iterates `n/2` times since we're stepping by `2`. Inside the loop, we perform at most two `find` operations and one union operation per iteration. Since the `find` operation's amortized time complexity is `O(α(n))` and the union operation can be considered `O(1)` in this amortized context, the total time complexity inside the loop is `O(n * α(n))`.
- After the loop, we have a comprehension that iterates `n` times to count the number of unique parents, which is `O(n)`.

Combining these, the overall time complexity is `O(n * α(n)) + O(n)`, which simplifies to `O(n * α(n))` due to the dominance of the `find` operation within the loop.

Space Complexity:

The space complexity is determined by the space used to store the parent array `p` and any auxiliary space used by recursion in the `find` function.

- The parent array `p` has a size of `n`, which is half the number of elements in the initial row because we're considering couples.
- The `find` function's recursion depth is at most `O(α(n))`. However, due to the path compression (`p[x] = find(p[x])`), the depth does not expand with each call; thus, the recursion stack does not significantly increase the space used.

Hence, the space complexity is essentially `O(n)` for the parent array `p`, with no significant additional space used for recursive calls due to path compression. Overall the space complexity is `O(n)`.