1029. Two City Scheduling

Medium Greedy Array Sorting Leetcode Link

Problem Description

A company is conducting interviews and needs to fly interviewees to two different cities, city `a` and city `b`. They have a total of `2n` people to interview and need to send exactly `n` people to each city. The cost of flying each person to each city is different and is provided in a list known as `costs`. The `costs` list is structured such that `costs[i] = [aCost_i, bCost_i]`, with `aCost_i` representing the cost to fly the `i`th person to city `a` and `bCost_i` representing the cost to fly the `i`th person to city `b`. The goal is to determine the minimum possible total cost to fly exactly `n` people to city `a` and `n` people to city `b`.

Intuition

The intuition behind the solution is to use a greedy strategy. A greedy algorithm makes the locally optimal choice at each step with the hope of finding the global optimum. In the context of this problem, we want to minimize the total cost of flying people to the two cities. To do this, we need to find the `n` people for whom the difference between flying to city `a` and city `b` is the largest in favor of city `a`, and fly them there. This will ensure that we are saving as much money as possible on the first `n` flights.

By sorting the `costs` array based on the difference `aCost_i - bCost_i` in ascending order, we can easily find the `n` people to fly to city `a` (those at the start of the sorted list) and the `n` people to fly to city `b` (those at the end of the list). This sorting allows us to consider the relative costs of flying each person to each city and ensure that, overall, we are making the most cost-effective decisions.

The solution handles the sorting and then computes the total cost by summing up the costs for flying the first `n` people to city `a` and the last `n` people to city `b`.

Solution Approach

The solution approach for the given problem uses a greedy algorithm, which is a common pattern for optimization problems. The idea here is to make a sequence of choices that are locally optimal in order to reach a solution that is close to or equal to the optimal global solution. Here's a breakdown of how the approach is implemented:

Steps Involved:

1. **Sorting Costs**: First, we sort the `costs` array by the difference in cost between flying to city `a` and city `b` (`aCost_i - bCost_i`). By doing this, we cluster people who are cheaper to fly to city `a` at the beginning of the array and those cheaper to city `b` towards the end.

   ```
   costs.sort(key=lambda x: x[0] - x[1])
   ```

2. **Calculating Half Length**: Since we need to send `n` people to each city and we have `2n` people in total, we find `n` by dividing the length of the costs array by 2. Note that we're using a bit shift operation to divide by 2, which is equivalent but slightly faster in most programming languages.

   ```
   n = len(costs) >> 1
   ```

3. **Determining Minimum Cost**: Finally, we take the sum of the costs for flying the first `n` people to city `a` and the last `n` people to city `b`, which gives us the minimum cost.

   ```
   sum(costs[i][0] + costs[i + n][1] for i in range(n))
   ```

Data Structures Used:

- A list of lists is used to store the flying costs to both cities for each person.

Algorithms & Patterns Used:

- Greedy Algorithm: The algorithm sorts the array and then selects the first `n` elements for one city and the last `n` elements for the other city, based on the local optimality of the cost difference.
- Sorting: A key step in the greedy approach where the array is sorted based on a specific condition (cost difference here).
- List Comprehension: Python list comprehension is used to sum the costs which result in a concise and readable implementation.

Complexity Analysis:

- The time complexity of the solution is dominated by the sorting step, which is `O(n log n)` where `n` is the number of people.
- The space complexity is `O(1)` or constant space, aside from the input, since no additional data structures are used that grow with the size of the input.

The algorithm implemented here effectively balances the costs for both cities by understanding and utilizing the differences in individual costs to minimize the total expenditure.

Example Walkthrough

Let's consider an example where a company has `2n = 4` people (`n = 2`) to send to cities `a` and `b`. The costs for flying each person to cities `a` and `b` are provided as follows:

```
costs = [[400, 200], [300, 600], [100, 500], [200, 300]]
```

Now, let's walk through the solution approach step by step using this example:

1. **Sorting Costs** First, we sort the costs array based on the difference between flying to city `a` and city `b`, so we calculate `aCost_i - bCost_i` for each person:

   ```
   Person 1: 400 - 200 = 200
   Person 2: 300 - 600 = -300
   Person 3: 100 - 500 = -400
   Person 4: 200 - 300 = -100
   ```

   After sorting these based on the difference in ascending order, our `costs` array looks like this:

   ```
   costs = [[100, 500], [300, 600], [200, 300], [400, 200]]
   ```

   Now, the first two people on the list are the ones we will fly to city `a`, and the last two to city `b`.

2. **Calculating Half Length** Here, `n` is already given as 2 (since `2n = 4`). In a more general sense, we would compute `n` as `len(costs) >> 1`.

3. **Determining Minimum Cost** Lastly, we will add the cost for flying the first `n` people to city `a` and the last `n` people to city `b`:

   ```
   For city a: costs[0][0] + costs[1][0] = 100 + 300 = 400
   For city b: costs[2][1] + costs[3][1] = 300 + 200 = 500
   ```

   The total minimum cost to fly all these people is the sum of the costs for city `a` and city `b`: `400 + 500 = 900`.

Through this example, we can see how the company efficiently reduces the total cost for flying out interviewees to two different cities by implementing a greedy algorithm that focuses on sending each person to the city that results in the lowest additional cost.

Python Solution

```python
from typing import List

class Solution:
    def two_city_sched_cost(self, costs: List[List[int]]) -> int:
        # Sort the costs list based on the cost difference between the two cities (A and B).
        # This helps in figuring out the cheaper city for each person in a way that can minimize the total cost.
        costs.sort(key=lambda cost: cost[0] - cost[1])

        # Compute the number of people that should go to each city.
        # Since it's a two-city schedule, half the people will go to each city.
        num_people = len(costs) // 2  # Using // for floor division which is standard in Python 3.

        # Calculate the total minimum cost by adding the cost of sending the first half
        # of the people to city A and the remaining half to city B.
        # This works because the costs array is now sorted in a way that the first half
        # are the people who have a cheaper cost going to city A compared
        # to city B, and vice versa for the second half.
        total_cost = sum(costs[i][0] for i in range(num_people)) + \
                     sum(costs[i + num_people][1] for i in range(num_people))

        return total_cost

# Example usage:
# costs = [[10,20],[30,200],[400,50],[30,20]]
# solution = Solution()
# print(solution.two_city_sched_cost(costs))  # Should print the result of the cost calculation based on the provided costs array.

```

Java Solution

```java
import java.util.Arrays; // Import the Arrays class to use the sort method

class Solution {
    public int twoCitySchedCost(int[][] costs) {
        // Sort the array based on the cost difference between the two cities (a and b)
        // for each person. The comparator subtracts the cost of going to city B from the
        // cost of going to city A for each person, and sorts based on these differences.
        Arrays.sort(costs, (a, b) -> (a[0] - a[1]) - (b[0] - b[1]));

        int totalCost = 0; // Initialize total cost to 0
        int n = costs.length / 2; // Calculate n, where n is half the number of people (half will go to city A, half to city B)

        // Accumulate the minimum cost by sending the first n people to city A
        // and the remaining n people to city B based on the sorted order
        for (int i = 0; i < n; ++i) {
            totalCost += costs[i][0];   // Add cost for city A
            totalCost += costs[i + n][1]; // Add cost for city B
        }

        return totalCost; // Return the accumulated total cost
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include necessary headers

class Solution {
public:
    // Function to calculate the minimum cost to send n people to two cities.
    int twoCitySchedCost(vector<vector<int>>& costs) {
        // Sort the input vector based on the cost difference for attending city A and city B
        sort(costs.begin(), costs.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[0] - a[1] < b[0] - b[1];
        });

        // The total number of people is twice the number of people we need to send to one city
        int totalPeople = costs.size();
        // Calculate the number of people to send to each city
        int halfPeople = totalPeople / 2;
        // Initialize answer to store the total minimized cost
        int totalCost = 0;

        // Calculate the minimum total cost
        for (int i = 0; i < halfPeople; ++i) {
            // Add up the cost of sending the first half of people to city A
            totalCost += costs[i][0];
            // Add up the cost of sending the second half of people to city B
            totalCost += costs[i + halfPeople][1];
        }

        // Return the computed total cost
        return totalCost;
    }
};

```

Typescript Solution

```typescript
function twoCitySchedCost(costs: number[][]): number {
    // Sort the array of costs based on the difference in cost between sending
    // a person to city A and city B.
    costs.sort((firstPair, secondPair) => (firstPair[0] - firstPair[1]) - (secondPair[0] - secondPair[1]));

    // Calculate half the length of the costs array since we're splitting the group in two
    const halfLength = costs.length / 2;

    // Initialize an accumulator for the total cost
    let totalCost = 0;

    // Iterate over the first half of the sorted array for city A costs and the second
    // half for city B costs to compute the minimum total cost
    for (let i = 0; i < halfLength; ++i) {
        // Add the cost of sending the i-th person to city A from the first half
        totalCost += costs[i][0];

        // Add the cost of sending the i-th person to city B from the second half
        totalCost += costs[i + halfLength][1];
    }

    // Return the computed total cost
    return totalCost;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code consists of the following parts:

1. Sorting the `costs` list: Since the `sort()` function in Python uses the Timsort algorithm, which has a time complexity of `O(n log n)`, where `n` is the total number of elements in the list. Here `n` is `2N` since we're supposed to send `2N` people to two cities (`N` to each city).

2. Summing up the costs: The list comprehension iterates over the first half and the second half of the sorted list, resulting in `N` iterations (since `n = len(costs) >> 1` computes to `N`). The summing operation inside the list comprehension is `O(1)` for each element, so the overall time for this part is `O(N)`.

Combining these two parts, the total time complexity is `O(n log n) + O(N)`, which simplifies to `O(n log n)` since logarithmic factors are dominant for large `n`.

Hence, the time complexity of the code is `O(n log n)`.

Space Complexity

The space complexity of the code analysis involves:

1. The sorted list `costs`: Sorting is done in-place, so it does not consume additional space proportional to the input size.

2. The list comprehension for summing the costs and the range used within: both utilize only a small constant amount of additional space.

Therefore, the space complexity of the entire operation is `O(1)`, as no significant additional space is used that would increase with the size of the input.

Overall, the space complexity of the code is `O(1)`.