1057. Campus Bikes

Medium Greedy Array Sorting Leetcode Link

Problem Description

The task involves assigning bikes to workers on a campus that is laid out on an X-Y coordinate plane. There are `n` workers and `m` bikes, with the condition that `n <= m`, meaning there are at least as many bikes as there are workers. Each worker and bike has a unique position on the plane, represented by their coordinates.

The goal is to assign each worker exactly one bike based on the shortest Manhattan distance between them. The Manhattan distance between two points `p1` and `p2` is defined as `|p1.x - p2.x| + |p1.y - p2.y|`, which essentially measures the distance between the points if you could only move along the grid horizontally or vertically.

If multiple workers could be assigned to the same bike due to equal distances, the worker with the lower index gets priority. Similarly, if one worker has the option of multiple bikes at the same distance, the bike with the lower index is assigned.

The output should be an array where each element represents the index of the bike assigned to the worker at the corresponding index.

Intuition

The solution approach requires creating a list of all possible combinations of workers and bikes along with their Manhattan distances. These combinations are then sorted primarily by the distance, then by the worker index, and finally by the bike index. This guarantees that the shortest and prioritized combinations (based on worker and bike indices) are considered first in the sorted list.

Once sorted, the algorithm iterates through the list of combinations and assigns bikes to workers one by one. During this process, both workers and bikes are marked as 'visited' to ensure that each worker gets only one bike, and each bike is assigned only once.

By sorting the combination of distances and applying the rules of priority in indexing, we ensure an efficient distribution of bikes to workers based on the proximity and predefined rules. This approach guarantees that the problem constraints will be respected and the correct assignments will be made.

Solution Approach

The reference solution provided implements a straightforward but effective algorithm to solve the problem with a greedy approach. Here is a walkthrough of the solution step by step:

1. **Collect Pair Information**: Initially, the solution collects all possible combinations of worker and bike pairs along with their Manhattan distances. This is done using a nested loop to iterate over each worker and bike to calculate their distance: `dist = abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j][1])`.

2. **Distance-Pair Array**: These pairs, along with the distances, are stored in an array where each element is a tuple comprising the distance, the worker index `i`, and the bike index `j`: `(dist, i, j)`.

3. **Sorting Pair Array**: Next, the tuples are sorted based on the first, second, and third elements (distance, worker index, and bike index, respectively). This orders the pairings primarily by the shortest distance, then by the smallest worker index, and lastly by the smallest bike index if there's a tie on the distance.

4. **Initialize Visited Flags**: Two arrays `vis1` for workers and `vis2` for bikes are created to keep track of whether a worker has been assigned a bike and whether a bike has been taken by a worker.

5. **Assignment**: The solution iterates through the sorted array and checks the visited flags for both workers and bikes. If both are unvisited, (i.e., the worker has not yet been assigned a bike and the bike has not been assigned to any worker), the bike is assigned to the worker: `ans[i] = j` and the visited flags are updated to `True`.

6. **Return Assignment**: Once all workers have been assigned a bike, the `ans` array, which holds the mapping of workers to their assigned bike indices, is returned.

It is worth noting that this approach leverages the Python's tuple sorting behavior which automatically sorts the tuples based on their first element, and in case of ties, it uses the second element, and so on. The algorithm also makes efficient use of space by using boolean arrays to keep track of the workers and bikes that have already been assigned. This algorithm achieves a balance between performance and clarity, allowing for an effective solution to the problem.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with 2 workers and 3 bikes.

Initial Setup:

- Workers' coordinates: `[(0, 0), (2, 1)]` (Worker 0 at `(0, 0)` and Worker 1 at `(2, 1)`)
- Bikes' coordinates: `[(1, 0), (2, 2), (2, 3)]` (Bike 0 at `(1, 0)`, Bike 1 at `(2, 2)`, and Bike 2 at `(2, 3)`)

Step 1: Collect Pair Information We begin by calculating the Manhattan distance for all possible worker-bike pairs:

- Worker 0 & Bike 0: `abs(0 - 1) + abs(0 - 0) = 1`
- Worker 0 & Bike 1: `abs(0 - 2) + abs(0 - 2) = 4`
- Worker 0 & Bike 2: `abs(0 - 2) + abs(0 - 3) = 5`
- Worker 1 & Bike 0: `abs(2 - 1) + abs(1 - 0) = 2`
- Worker 1 & Bike 1: `abs(2 - 2) + abs(1 - 2) = 1`
- Worker 1 & Bike 2: `abs(2 - 2) + abs(1 - 3) = 2`

Step 2: Distance-Pair Array Construct an array with the pairs and their distances:

- `[(1, 0, 0), (4, 0, 1), (5, 0, 2), (2, 1, 0), (1, 1, 1), (2, 1, 2)]`

Step 3: Sorting Pair Array Sort the array of tuples by distance, then by worker index, and then by bike index:

- `[(1, 0, 0), (1, 1, 1), (2, 1, 0), (2, 1, 2), (4, 0, 1), (5, 0, 2)]`

Step 4: Initialize Visited Flags Create boolean arrays to track the visited workers and bikes:

- `vis1 = [False, False]` (for workers)
- `vis2 = [False, False, False]` (for bikes)

Step 5: Assignment Iterate through the sorted pairs and assign bikes to workers:

- Worker 0 is assigned Bike 0 because the pair `(1, 0, 0)` comes first and both Worker 0 and Bike 0 are unvisited. Mark Worker 0 and Bike 0 as visited.
- Skip to the next unvisited pair `(1, 1, 1)`. Assign Worker 1 to Bike 1 and mark them as visited.
- Remaining pairs either contain a worker or a bike that has already been assigned, so we skip them.

Step 6: Return Assignment The `ans` array after the assignments is `[0, 1]`, indicating that Worker 0 is assigned Bike 0 and Worker 1 is assigned Bike 1.

Now the assignment mapping of workers to bike indices is returned, completing the algorithm execution for the given example.

Python Solution

```python
from typing import List
from itertools import product

class Solution:
    def assignBikes(self, workers: List[List[int]], bikes: List[List[int]]) -> List[int]:
        num_workers = len(workers)
        num_bikes = len(bikes)
        distances = []  # List to hold the distance and associated worker and bike indices

        # Calculate the manhattan distances between each worker-bike pair and store them
        for worker_index, bike_index in product(range(num_workers), range(num_bikes)):
            distance = abs(workers[worker_index][0] - bikes[bike_index][0]) \
                       + abs(workers[worker_index][1] - bikes[bike_index][1])
            distances.append((distance, worker_index, bike_index))

        # Sort the distances list in ascending order
        distances.sort()

        # Initialize the visited lists for workers and bikes
        visited_workers = [False] * num_workers
        visited_bikes = [False] * num_bikes
        assignments = [-1] * num_workers  # -1 to indicate unassigned workers

        # Assign bikes to workers based on sorted distances
        for distance, worker_index, bike_index in distances:
            # Assign a bike to a worker if both are unvisited/unassigned
            if not visited_workers[worker_index] and not visited_bikes[bike_index]:
                visited_workers[worker_index] = True   # Mark the worker as visited
                visited_bikes[bike_index] = True       # Mark the bike as visited
                assignments[worker_index] = bike_index # Assign the bike to the worker

        return assignments

```

Java Solution

```java
class Solution {
    public int[] assignBikes(int[][] workers, int[][] bikes) {
        int numWorkers = workers.length;
        int numBikes = bikes.length;
        // Create an array to store distance, worker index, and bike index
        int[][] distances = new int[numWorkers * numBikes][3];

        // Calculate the Manhattan distance between each worker and bike pair
        int index = 0;
        for (int i = 0; i < numWorkers; ++i) {
            for (int j = 0; j < numBikes; ++j) {
                int dist = Math.abs(workers[i][0] - bikes[j][0]) +
                           Math.abs(workers[i][1] - bikes[j][1]);
                distances[index++] = new int[] {dist, i, j};
            }
        }

        // Sort the distances array by distance first, then worker index, then bike index
        Arrays.sort(distances, (a, b) -> {
            if (a[0] != b[0]) {
                return a[0] - b[0]; // Compare distances
            }
            if (a[1] != b[1]) {
                return a[1] - b[1]; // Compare worker indices
            }
            return a[2] - b[2]; // Compare bike indices
        });

        // Keep track of which workers and bikes have been visited/assigned
        boolean[] workersVisited = new boolean[numWorkers];
        boolean[] bikesVisited = new boolean[numBikes];
        int[] assignments = new int[numWorkers]; // Result array with assignments

        // Assign bikes to workers based on the sorted distances
        for (int[] arr : distances) {
            int workerIdx = arr[1];
            int bikeIdx = arr[2];
            // If the current worker and bike have not been visited yet, assign the bike to the worker
            if (!workersVisited[workerIdx] && !bikesVisited[bikeIdx]) {
                workersVisited[workerIdx] = true;
                bikesVisited[bikeIdx] = true;
                assignments[workerIdx] = bikeIdx;
            }
        }

        return assignments; // Return the final assignment of bikes to workers
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to assign bikes to workers
    vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>& bikes) {
        int numWorkers = workers.size();
        int numBikes = bikes.size();

        // Create an array to store the distances and indices
        vector<tuple<int, int, int>> allPairs(numWorkers * numBikes);

        // Iterate over each worker and each bike to calculate distances
        for (int workerIndex = 0, pairIndex = 0; workerIndex < numWorkers; ++workerIndex) {
            for (int bikeIndex = 0; bikeIndex < numBikes; ++bikeIndex) {
                int distance = abs(workers[workerIndex][0] - bikes[bikeIndex][0]) +
                               abs(workers[workerIndex][1] - bikes[bikeIndex][1]);
                // Store the tuple of distance, worker index, and bike index
                allPairs[pairIndex++] = {distance, workerIndex, bikeIndex};
            }
        }

        // Sort all pairs by distance, then by worker index, then by bike index
        sort(allPairs.begin(), allPairs.end());

        // Vectors to keep track of which workers and bikes have been visited
        vector<bool> workerVisited(numWorkers, false);
        vector<bool> bikeVisited(numBikes, false);

        // Initialize the answer vector to store the bike index for each worker
        vector<int> answer(numWorkers, -1);  // Default to -1 for unassigned

        // Iterate over each sorted pair
        for (auto& [distance, workerIndex, bikeIndex] : allPairs) {
            // If the current worker and bike have not been visited
            if (!workerVisited[workerIndex] && !bikeVisited[bikeIndex]) {
                // Assign the bike to the worker
                workerVisited[workerIndex] = true;
                bikeVisited[bikeIndex] = true;
                answer[workerIndex] = bikeIndex;
            }
        }

        // Return the final assignment of bikes to workers
        return answer;
    }
};

```

Typescript Solution

```typescript
// Import necessary elements from array functions
import { abs, sort } from 'math';

// Type definition for a pair representing distance, worker index, and bike index
type WorkerBikePair = [number, number, number];

// Function to calculate the Manhattan distance between a worker and a bike
function calculateManhattanDistance(worker: number[], bike: number[]): number {
    return abs(worker[0] - bike[0]) + abs(worker[1] - bike[1]);
}

// Function to assign bikes to workers based on their Manhattan distances
function assignBikes(workers: number[][], bikes: number[][]): number[] {
    const numWorkers: number = workers.length;
    const numBikes: number = bikes.length;

    // Array to store all worker-bike pairs with their corresponding distances
    const allPairs: WorkerBikePair[] = [];

    // Populate the array with distances and indices
    for (let workerIndex = 0; workerIndex < numWorkers; ++workerIndex) {
        for (let bikeIndex = 0; bikeIndex < numBikes; ++bikeIndex) {
            const distance: number = calculateManhattanDistance(workers[workerIndex], bikes[bikeIndex]);
            allPairs.push([distance, workerIndex, bikeIndex]);
        }
    }

    // Sort the pairs by distance, then by worker index, then by bike index
    allPairs.sort((a, b) => {
        // Compare by distance
        if (a[0] !== b[0]) {
            return a[0] - b[0];
        }
        // If distance is the same, compare by worker index
        if (a[1] !== b[1]) {
            return a[1] - b[1];
        }
        // If distance and worker index are the same, compare by bike index
        return a[2] - b[2];
    });

    // Arrays to keep track of whether workers or bikes are already matched
    const workerVisited: boolean[] = new Array(numWorkers).fill(false);
    const bikeVisited: boolean[] = new Array(numBikes).fill(false);

    // Array to keep the results of bike assignments for each worker
    const answer: number[] = new Array(numWorkers).fill(-1);

    // Assign bikes to workers based on the sorted allPairs
    for (const [distance, workerIndex, bikeIndex] of allPairs) {
        // Proceed if both the worker and the bike are not yet visited
        if (!workerVisited[workerIndex] && !bikeVisited[bikeIndex]) {
            // Mark as matched
            workerVisited[workerIndex] = true;
            bikeVisited[bikeIndex] = true;
            // Assign the bike to the worker in the answer array
            answer[workerIndex] = bikeIndex;
        }
    }

    // Return the final assignments
    return answer;
}

```

Time and Space Complexity

The time complexity and space complexity of the given code can be analyzed as follows:

Time Complexity

1. The nested loop with `product(range(n), range(m))` generates all possible pairs of workers and bikes. There are `n` workers and `m` bikes, so there are `n * m` pairs. This operation has a time complexity of `O(n * m)`.

2. Calculating the Manhattan distance for each worker-bike pair takes constant time, so this does not change the overall time complexity from `O(n * m)`.

3. Appending each tuple `(dist, i, j)` to the `arr` list is done `n * m` times, which is also `O(n * m)`.

4. The sort operation on `arr` has a time complexity of `O(n * m * log(n * m))` because sorting is typically implemented with an `O(n log n)` complexity, and here we are sorting `n * m` elements.

5. The final loop goes over the sorted array, which has `n * m` elements, and it assigns bikes to workers. Since each worker and bike is visited at most once, the loop will run at most `n + m` times - because once every worker has a bike, the process stops. The time complexity for this loop is thus `O(n + m)`.

Therefore, the overall time complexity of the code is dominated by the sorting step, which is `O(n * m * log(n * m))`.

Space Complexity

1. The `arr` list contains `n * m` tuples, so it has a space complexity of `O(n * m)`.

2. The `vis1` and `vis2` arrays each have a space complexity of `O(n)` and `O(m)` respectively. Adding them together gives `O(n + m)`.

3. The answer array `ans` has a space complexity of `O(n)` because it must hold an assignment for each worker.

The additional space required is for the tuples and the arrays to track the visited workers and bikes. Since these do not depend on the size of the input apart from the number of workers and bikes, we can consider them all in aggregate.

Hence, the overall space complexity is `O(n * m)` as it is the term that grows more significantly with the input size.