Problem 1057: Campus Bikes

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

On a campus represented on the X-Y plane, there are

n

workers and

m

bikes, with

n <= m

.

You are given an array

workers

of length

n

where

workers[i] = [x

i

, y

i

]

is the position of the

i

th

worker. You are also given an array

bikes

of length

m

where

bikes[j] = [x

j

, y

j

]

is the position of the

j

th

bike. All the given positions are

unique

.

Assign a bike to each worker. Among the available bikes and workers, we choose the

(worker

i

, bike

j

)

pair with the shortest

Manhattan distance

between each other and assign the bike to that worker.

If there are multiple

(worker

i

, bike

j

)

pairs with the same shortest

Manhattan distance

, we choose the pair with

the smallest worker index

. If there are multiple ways to do that, we choose the pair with

the smallest bike index

. Repeat this process until there are no available workers.

Return

an array

answer

of length

n

, where

answer[i]

is the index (

0-indexed

) of the bike that the

i

th

worker is assigned to

.

The

Manhattan distance

between two points

p1

and

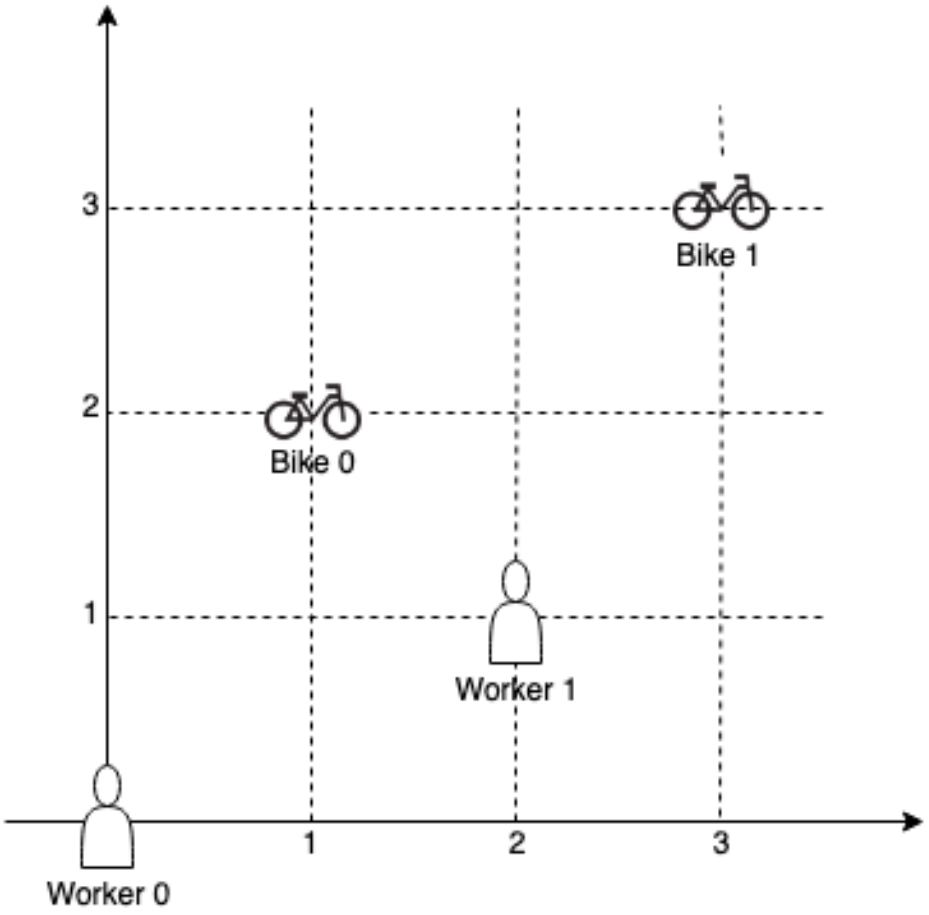
p2

is

Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|

.

Example 1:



Input:

workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

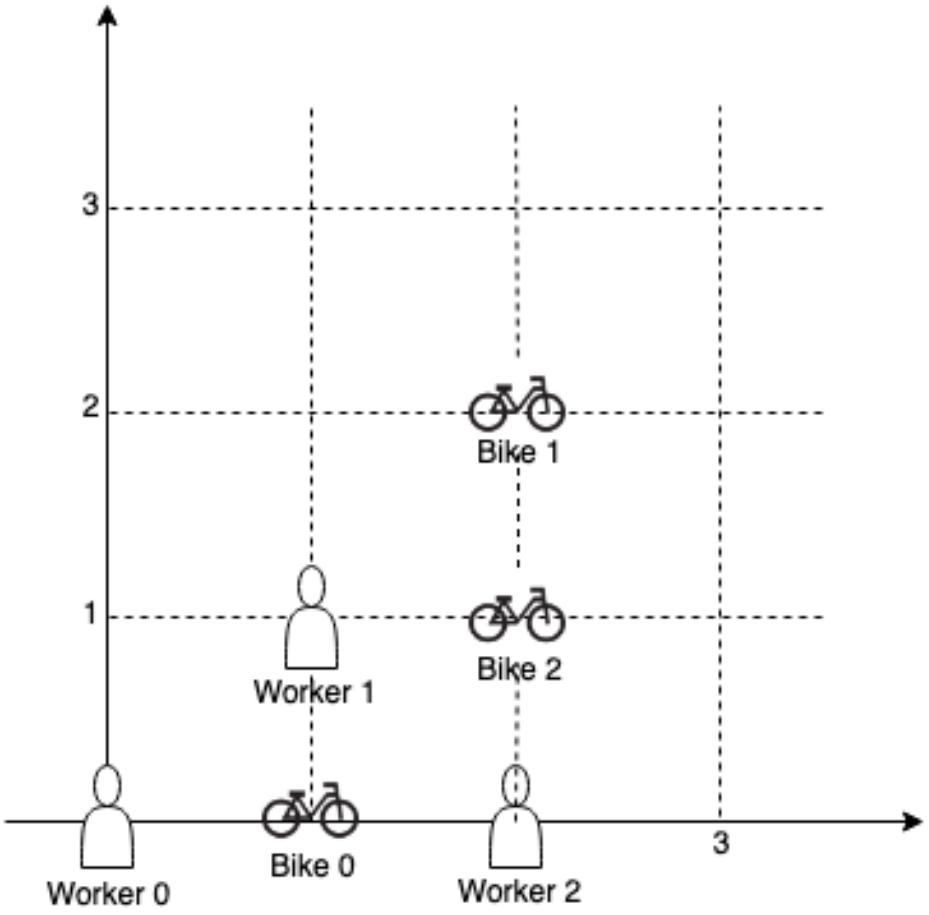
Output:

[1,0]

Explanation:

Worker 1 grabs Bike 0 as they are closest (without ties), and Worker 0 is assigned Bike 1. So the output is [1, 0].

Example 2:



Input:

workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

Output:

[0,2,1]

Explanation:

Worker 0 grabs Bike 0 at first. Worker 1 and Worker 2 share the same distance to Bike 2, thus Worker 1 is assigned to Bike 2, and Worker 2 will take Bike 1. So the output is [0,2,1].

Constraints:

n == workers.length

m == bikes.length

1 <= n <= m <= 1000

workers[i].length == bikes[j].length == 2

0 <= x

i

, y

i

< 1000

0 <= x

j

, y

j

< 1000

All worker and bike locations are

unique

.

Code Snippets

C++:

```
class Solution {
public:
vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {
```

```
}
};
```

Java:

```
class Solution {
public int[] assignBikes(int[][] workers, int[][] bikes) {

}
}
```

Python3:

```
class Solution:
def assignBikes(self, workers: List[List[int]], bikes: List[List[int]]) ->
List[int]:
```

Python:

```
class Solution(object):
def assignBikes(self, workers, bikes):
"""
:type workers: List[List[int]]
:type bikes: List[List[int]]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[][]} workers
* @param {number[][]} bikes
* @return {number[]}
*/
var assignBikes = function(workers, bikes) {

};
```

TypeScript:

```
function assignBikes(workers: number[][], bikes: number[][]): number[] {

};
```

C#:

```
public class Solution {
public int[] AssignBikes(int[][] workers, int[][] bikes) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* assignBikes(int** workers, int workersSize, int* workersColSize, int**
bikes, int bikesSize, int* bikesColSize, int* returnSize) {

}
```

Go:

```
func assignBikes(workers [][]int, bikes [][]int) []int {

}
```

Kotlin:

```
class Solution {
fun assignBikes(workers: Array<IntArray>, bikes: Array<IntArray>): IntArray {

}
}
```

Swift:

```
class Solution {
func assignBikes(_ workers: [[Int]], _ bikes: [[Int]]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn assign_bikes(workers: Vec<Vec<i32>>, bikes: Vec<Vec<i32>>) -> Vec<i32>
{

}
}
```

Ruby:

```
# @param {Integer[][]} workers
# @param {Integer[][]} bikes
# @return {Integer[]}
def assign_bikes(workers, bikes)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $workers
* @param Integer[][] $bikes
* @return Integer[]
*/
function assignBikes($workers, $bikes) {

}
}
```

Dart:

```
class Solution {
List<int> assignBikes(List<List<int>> workers, List<List<int>> bikes) {

}
}
```

Scala:

```
object Solution {
def assignBikes(workers: Array[Array[Int]], bikes: Array[Array[Int]]):
Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec assign_bikes(workers :: [[integer]], bikes :: [[integer]]) :: [integer]
def assign_bikes(workers, bikes) do

end
end
```

Erlang:

```
-spec assign_bikes(Workers :: [[integer()]], Bikes :: [[integer()]]) ->
[integer()].
assign_bikes(Workers, Bikes) ->
.
```

Racket:

```
(define/contract (assign-bikes workers bikes)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Campus Bikes
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {

}
};
```

Java Solution:

```
/**
* Problem: Campus Bikes
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int[] assignBikes(int[][] workers, int[][] bikes) {

}
}
```

Python3 Solution:

```
"""
Problem: Campus Bikes
Difficulty: Medium
Tags: array, sort, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""
```

```
class Solution:
def assignBikes(self, workers: List[List[int]], bikes: List[List[int]]) ->
List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def assignBikes(self, workers, bikes):
"""
:type workers: List[List[int]]
:type bikes: List[List[int]]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Campus Bikes
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} workers
* @param {number[][]} bikes
* @return {number[]}
*/
var assignBikes = function(workers, bikes) {

};
```

TypeScript Solution:

```
/**
* Problem: Campus Bikes
```

```
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function assignBikes(workers: number[][], bikes: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Campus Bikes
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[] AssignBikes(int[][] workers, int[][] bikes) {

}
}
```

C Solution:

```
/*
* Problem: Campus Bikes
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* assignBikes(int** workers, int workersSize, int* workersColSize, int**
bikes, int bikesSize, int* bikesColSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Campus Bikes
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func assignBikes(workers [][]int, bikes [][]int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun assignBikes(workers: Array<IntArray>, bikes: Array<IntArray>): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func assignBikes(_ workers: [[Int]], _ bikes: [[Int]]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Campus Bikes
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn assign_bikes(workers: Vec<Vec<i32>>, bikes: Vec<Vec<i32>>) -> Vec<i32>
{

}
}
```

Ruby Solution:

```
# @param {Integer[][]} workers
# @param {Integer[][]} bikes
# @return {Integer[]}
def assign_bikes(workers, bikes)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $workers
* @param Integer[][] $bikes
* @return Integer[]
*/
function assignBikes($workers, $bikes) {

}
}
```

Dart Solution:

```
class Solution {
List<int> assignBikes(List<List<int>> workers, List<List<int>> bikes) {
```

```
}
}
```

Scala Solution:

```
object Solution {
def assignBikes(workers: Array[Array[Int]], bikes: Array[Array[Int]]):
Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec assign_bikes(workers :: [[integer]], bikes :: [[integer]]) :: [integer]
def assign_bikes(workers, bikes) do

end
end
```

Erlang Solution:

```
-spec assign_bikes(Workers :: [[integer()]], Bikes :: [[integer()]]) ->
[integer()].
assign_bikes(Workers, Bikes) ->
.
```

Racket Solution:

```
(define/contract (assign-bikes workers bikes)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
exact-integer?))
)
```