Problem 3102: Minimize Manhattan Distances

Problem Information

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

Problem Description

You are given an array

points

representing integer coordinates of some points on a 2D plane, where

points[i] = [x

i

, y

i

]

.

The distance between two points is defined as their

Manhattan distance

.

Return

the

minimum

possible value for

maximum

distance between any two points by removing exactly one point

.

Example 1:

Input:

points = [[3,10],[5,15],[10,2],[4,4]]

Output:

12

Explanation:

The maximum distance after removing each point is the following:

After removing the 0

th

point the maximum distance is between points (5, 15) and (10, 2), which is

|5 - 10| + |15 - 2| = 18

.

After removing the 1

st

point the maximum distance is between points (3, 10) and (10, 2), which is

|3 - 10| + |10 - 2| = 15

.

After removing the 2

nd

point the maximum distance is between points (5, 15) and (4, 4), which is

|5 - 4| + |15 - 4| = 12

.

After removing the 3

rd

point the maximum distance is between points (5, 15) and (10, 2), which is

|5 - 10| + |15 - 2| = 18

.

12 is the minimum possible maximum distance between any two points after removing exactly one point.

Example 2:

Input:

points = [[1,1],[1,1],[1,1]]

Output:

0

Explanation:

Removing any of the points results in the maximum distance between any two points of 0.

Constraints:

3 <= points.length <= 10

5

points[i].length == 2

1 <= points[i][0], points[i][1] <= 10

8

Code Snippets

C++:

```
class Solution {
public:
int minimumDistance(vector<vector<int>>& points) {

}
};
```

Java:

```
class Solution {
public int minimumDistance(int[][] points) {

}
}
```

Python3:

```
class Solution:
def minimumDistance(self, points: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def minimumDistance(self, points):
"""
:type points: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} points
* @return {number}
*/
var minimumDistance = function(points) {

};
```

TypeScript:

```
function minimumDistance(points: number[][]): number {

};
```

C#:

```
public class Solution {
public int MinimumDistance(int[][] points) {

}
}
```

C:

```
int minimumDistance(int** points, int pointsSize, int* pointsColSize) {

}
```

Go:

```
func minimumDistance(points [][]int) int {
```

```
}
```

Kotlin:

```
class Solution {
fun minimumDistance(points: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func minimumDistance(_ points: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_distance(points: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} points
# @return {Integer}
def minimum_distance(points)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $points
* @return Integer
*/
```

```
function minimumDistance($points) {

}
}
```

Dart:

```
class Solution {
int minimumDistance(List<List<int>> points) {

}
}
```

Scala:

```
object Solution {
def minimumDistance(points: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_distance(points :: [[integer]]) :: integer
def minimum_distance(points) do

end
end
```

Erlang:

```
-spec minimum_distance(Points :: [[integer()]]) -> integer().
minimum_distance(Points) ->
.
```

Racket:

```
(define/contract (minimum-distance points)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
* Tags: array, math, sort
*
* 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 minimumDistance(vector<vector<int>>& points) {

}
};
```

Java Solution:

```
/**
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
* Tags: array, math, sort
*
* 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 minimumDistance(int[][] points) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimize Manhattan Distances
Difficulty: Hard
Tags: array, math, sort

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 minimumDistance(self, points: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumDistance(self, points):
"""
:type points: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
* Tags: array, math, sort
*
* 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[][]} points
* @return {number}
*/
var minimumDistance = function(points) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
* Tags: array, math, sort
*
* 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 minimumDistance(points: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
* Tags: array, math, sort
*
* 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 MinimumDistance(int[][] points) {

}
}
```

C Solution:

```
/*
* Problem: Minimize Manhattan Distances
* Difficulty: Hard
```

```
* Tags: array, math, sort
*
* 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
*/

int minimumDistance(int** points, int pointsSize, int* pointsColSize) {

}
```

Go Solution:

```
// Problem: Minimize Manhattan Distances
// Difficulty: Hard
// Tags: array, math, sort
//
// 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 minimumDistance(points [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun minimumDistance(points: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func minimumDistance(_ points: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimize Manhattan Distances
// Difficulty: Hard
// Tags: array, math, sort
//
// 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 minimum_distance(points: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} points
# @return {Integer}
def minimum_distance(points)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $points
* @return Integer
*/
function minimumDistance($points) {

}
}
```

Dart Solution:

```
class Solution {
int minimumDistance(List<List<int>> points) {
```

```
}
}
```

Scala Solution:

```
object Solution {
def minimumDistance(points: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_distance(points :: [[integer]]) :: integer
def minimum_distance(points) do

end
end
```

Erlang Solution:

```
-spec minimum_distance(Points :: [[integer()]]) -> integer().
minimum_distance(Points) ->
.
```

Racket Solution:

```
(define/contract (minimum-distance points)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```