Problem 1924: Erect the Fence II

Problem Information

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

Problem Description

You are given a 2D integer array

trees

where

trees[i] = [x

i

, y

i

]

represents the location of the

i

th

tree in the garden.

You are asked to fence the entire garden using the minimum length of rope possible. The garden is well-fenced only if all the trees are enclosed

and the rope used

forms a perfect circle

. A tree is considered enclosed if it is inside or on the border of the circle.

More formally, you must form a circle using the rope with a center

(x, y)

and radius

r

where all trees lie inside or on the circle and

r

is

minimum

.

Return

the center and radius of the circle as a length 3 array

[x, y, r]

.

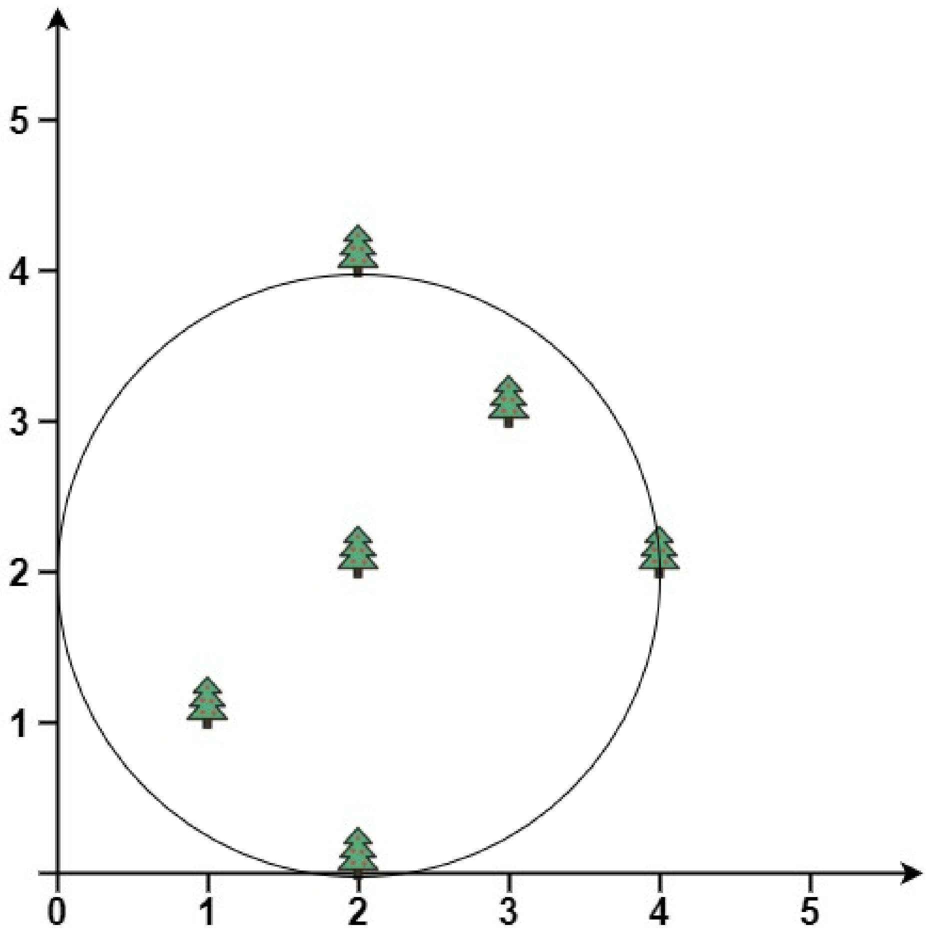
Answers within

10

-5

of the actual answer will be accepted.

Example 1:



Input:

trees = [[1,1],[2,2],[2,0],[2,4],[3,3],[4,2]]

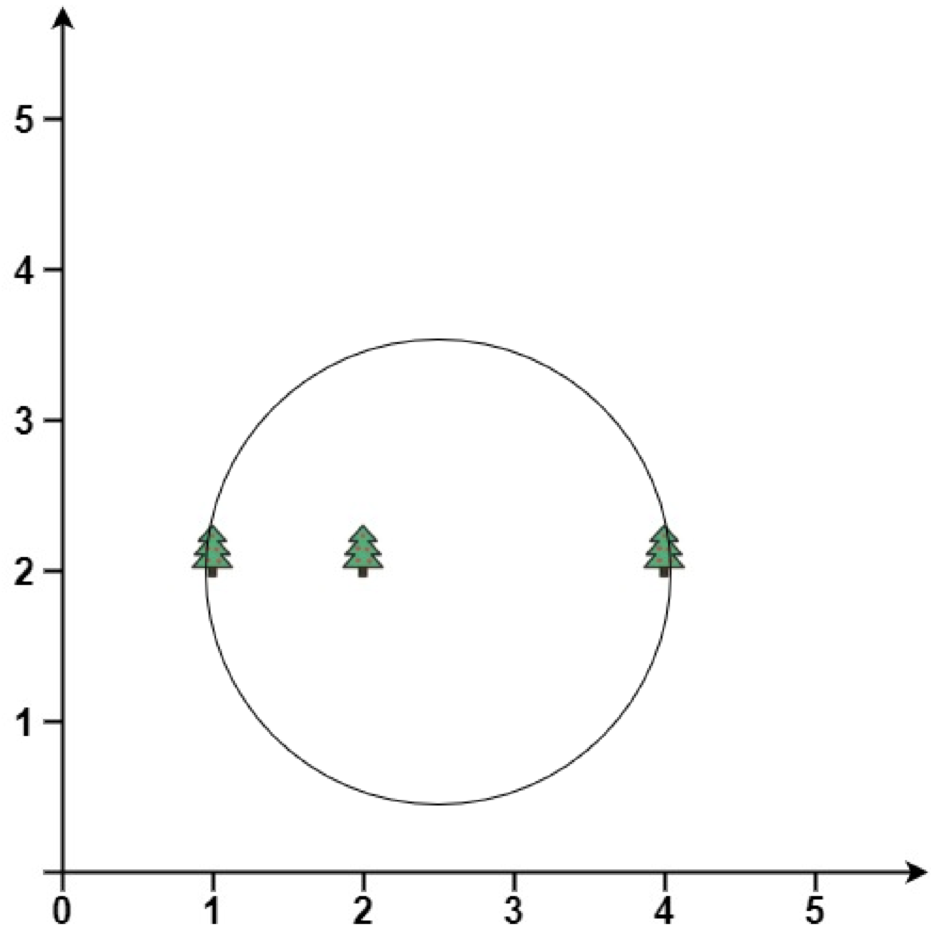
Output:

[2.00000,2.00000,2.00000]

Explanation:

The fence will have center = (2, 2) and radius = 2

Example 2:



Input:

trees = [[1,2],[2,2],[4,2]]

Output:

[2.50000,2.00000,1.50000]

Explanation:

The fence will have center = (2.5, 2) and radius = 1.5

Constraints:

1 <= trees.length <= 3000

trees[i].length == 2

0 <= x

i

, y

i

<= 3000

Code Snippets

C++:

```
class Solution {
public:
vector<double> outerTrees(vector<vector<int>>& trees) {

}
};
```

Java:

```
class Solution {
public double[] outerTrees(int[][] trees) {

}
}
```

Python3:

```
class Solution:
def outerTrees(self, trees: List[List[int]]) -> List[float]:
```

Python:

```
class Solution(object):
def outerTrees(self, trees):
```

```
"""
:type trees: List[List[int]]
:rtype: List[float]
"""
```

JavaScript:

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

};
```

TypeScript:

```
function outerTrees(trees: number[][]): number[] {

};
```

C#:

```
public class Solution {
public double[] OuterTrees(int[][] trees) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
double* outerTrees(int** trees, int treesSize, int* treesColSize, int*
returnSize) {

}
```

Go:

```
func outerTrees(trees [][]int) []float64 {

}
```

Kotlin:

```
class Solution {
fun outerTrees(trees: Array<IntArray>): DoubleArray {

}
}
```

Swift:

```
class Solution {
func outerTrees(_ trees: [[Int]]) -> [Double] {

}
}
```

Rust:

```
impl Solution {
pub fn outer_trees(trees: Vec<Vec<i32>>) -> Vec<f64> {

}
}
```

Ruby:

```
# @param {Integer[][]} trees
# @return {Float[]}
def outer_trees(trees)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $trees
* @return Float[]
```

```
*/
function outerTrees($trees) {

}
}
```

Dart:

```
class Solution {
List<double> outerTrees(List<List<int>> trees) {

}
}
```

Scala:

```
object Solution {
def outerTrees(trees: Array[Array[Int]]): Array[Double] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec outer_trees(trees :: [[integer]]) :: [float]
def outer_trees(trees) do

end
end
```

Erlang:

```
-spec outer_trees(Trees :: [[integer()]]) -> [float()].
outer_trees(Trees) ->
.
```

Racket:

```
(define/contract (outer-trees trees)
(-> (listof (listof exact-integer?)) (listof flonum?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Erect the Fence II
* Difficulty: Hard
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<double> outerTrees(vector<vector<int>>& trees) {

}
};
```

Java Solution:

```
/**
* Problem: Erect the Fence II
* Difficulty: Hard
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public double[] outerTrees(int[][] trees) {

}
}
```

Python3 Solution:

```
"""
Problem: Erect the Fence II
Difficulty: Hard
Tags: array, tree, math

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def outerTrees(self, trees: List[List[int]]) -> List[float]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def outerTrees(self, trees):
"""
:type trees: List[List[int]]
:rtype: List[float]
"""
```

JavaScript Solution:

```
/**
* Problem: Erect the Fence II
* Difficulty: Hard
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

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

```
};
```

TypeScript Solution:

```
/**
* Problem: Erect the Fence II
* Difficulty: Hard
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function outerTrees(trees: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Erect the Fence II
* Difficulty: Hard
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public double[] OuterTrees(int[][] trees) {

}
}
```

C Solution:

```
/*
* Problem: Erect the Fence II
* Difficulty: Hard
```

```
* Tags: array, tree, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
double* outerTrees(int** trees, int treesSize, int* treesColSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Erect the Fence II
// Difficulty: Hard
// Tags: array, tree, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func outerTrees(trees [][]int) []float64 {

}
```

Kotlin Solution:

```
class Solution {
fun outerTrees(trees: Array<IntArray>): DoubleArray {

}
}
```

Swift Solution:

```
class Solution {
func outerTrees(_ trees: [[Int]]) -> [Double] {
```

```
}
}
```

Rust Solution:

```
// Problem: Erect the Fence II
// Difficulty: Hard
// Tags: array, tree, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn outer_trees(trees: Vec<Vec<i32>>) -> Vec<f64> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} trees
# @return {Float[]}
def outer_trees(trees)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $trees
* @return Float[]
*/
function outerTrees($trees) {

}
}
```

Dart Solution:

```
class Solution {
List<double> outerTrees(List<List<int>> trees) {

}
}
```

Scala Solution:

```
object Solution {
def outerTrees(trees: Array[Array[Int]]): Array[Double] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec outer_trees(trees :: [[integer]]) :: [float]
def outer_trees(trees) do

end
end
```

Erlang Solution:

```
-spec outer_trees(Trees :: [[integer()]]) -> [float()].
outer_trees(Trees) ->
.
```

Racket Solution:

```
(define/contract (outer-trees trees)
(-> (listof (listof exact-integer?)) (listof flonum?))
)
```