Problem 1552: Magnetic Force Between Two Balls

Problem Information

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

Problem Description

In the universe Earth C-137, Rick discovered a special form of magnetic force between two balls if they are put in his new invented basket. Rick has

n

empty baskets, the

i

th

basket is at

position[i]

, Morty has

m

balls and needs to distribute the balls into the baskets such that the

minimum magnetic force

between any two balls is

maximum

.

Rick stated that magnetic force between two different balls at positions

x

and

y

is

|x - y|

.

Given the integer array

position

and the integer

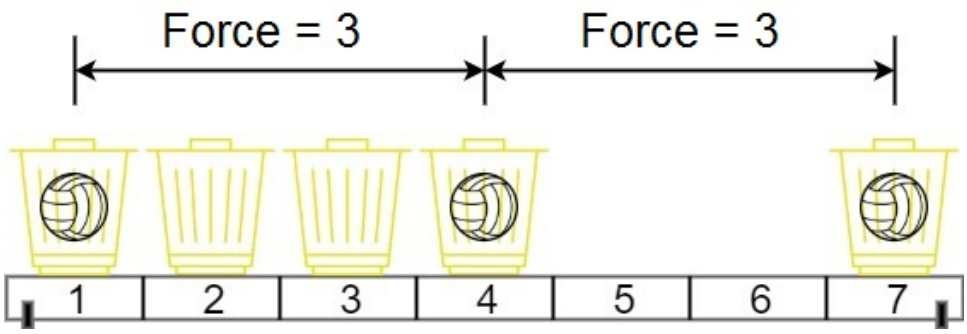
m

. Return

the required force

.

Example 1:



Input:

position = [1,2,3,4,7], m = 3

Output:

3

Explanation:

Distributing the 3 balls into baskets 1, 4 and 7 will make the magnetic force between ball pairs [3, 3, 6]. The minimum magnetic force is 3. We cannot achieve a larger minimum magnetic force than 3.

Example 2:

Input:

position = [5,4,3,2,1,1000000000], m = 2

Output:

999999999

Explanation:

We can use baskets 1 and 1000000000.

Constraints:

n == position.length

2 <= n <= 10

5

1 <= position[i] <= 10

9

All integers in

position

are

distinct

.

2 <= m <= position.length

Code Snippets

C++:

```
class Solution {
public:
int maxDistance(vector<int>& position, int m) {

}
};
```

Java:

```
class Solution {
public int maxDistance(int[] position, int m) {

}
}
```

Python3:

```
class Solution:
def maxDistance(self, position: List[int], m: int) -> int:
```

Python:

```
class Solution(object):
def maxDistance(self, position, m):
"""
:type position: List[int]
:type m: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} position
* @param {number} m
* @return {number}
*/
var maxDistance = function(position, m) {

};
```

TypeScript:

```
function maxDistance(position: number[], m: number): number {

};
```

C#:

```
public class Solution {
public int MaxDistance(int[] position, int m) {

}
}
```

C:

```
int maxDistance(int* position, int positionSize, int m) {

}
```

Go:

```
func maxDistance(position []int, m int) int {

}
```

Kotlin:

```
class Solution {
fun maxDistance(position: IntArray, m: Int): Int {

}
}
```

Swift:

```
class Solution {
func maxDistance(_ position: [Int], _ m: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn max_distance(position: Vec<i32>, m: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} position
# @param {Integer} m
# @return {Integer}
def max_distance(position, m)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $position
* @param Integer $m
* @return Integer
*/
function maxDistance($position, $m) {

}
}
```

Dart:

```
class Solution {
int maxDistance(List<int> position, int m) {

}
}
```

Scala:

```
object Solution {
def maxDistance(position: Array[Int], m: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_distance(position :: [integer], m :: integer) :: integer
def max_distance(position, m) do

end
end
```

Erlang:

```
-spec max_distance(Position :: [integer()], M :: integer()) -> integer().
max_distance(Position, M) ->
```

```
.
```

Racket:

```
(define/contract (max-distance position m)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 maxDistance(vector<int>& position, int m) {

}
};
```

Java Solution:

```
/**
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 maxDistance(int[] position, int m) {

}
}
```

Python3 Solution:

```
"""
Problem: Magnetic Force Between Two Balls
Difficulty: Medium
Tags: array, sort, search

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 maxDistance(self, position: List[int], m: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxDistance(self, position, m):
"""
:type position: List[int]
:type m: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
```

```
* 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[]} position
* @param {number} m
* @return {number}
*/
var maxDistance = function(position, m) {

};
```

TypeScript Solution:

```
/**
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 maxDistance(position: number[], m: number): number {

};
```

C# Solution:

```
/*
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 MaxDistance(int[] position, int m) {

}
}
```

C Solution:

```
/*
* Problem: Magnetic Force Between Two Balls
* Difficulty: Medium
* Tags: array, sort, search
*
* 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 maxDistance(int* position, int positionSize, int m) {

}
```

Go Solution:

```
// Problem: Magnetic Force Between Two Balls
// Difficulty: Medium
// Tags: array, sort, search
//
// 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 maxDistance(position []int, m int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxDistance(position: IntArray, m: Int): Int {
```

```
}
}
```

Swift Solution:

```
class Solution {
func maxDistance(_ position: [Int], _ m: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Magnetic Force Between Two Balls
// Difficulty: Medium
// Tags: array, sort, search
//
// 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 max_distance(position: Vec<i32>, m: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} position
# @param {Integer} m
# @return {Integer}
def max_distance(position, m)

end
```

PHP Solution:

```
class Solution {
```

```
/**
* @param Integer[] $position
* @param Integer $m
* @return Integer
*/
function maxDistance($position, $m) {

}
}
```

Dart Solution:

```
class Solution {
int maxDistance(List<int> position, int m) {

}
}
```

Scala Solution:

```
object Solution {
def maxDistance(position: Array[Int], m: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_distance(position :: [integer], m :: integer) :: integer
def max_distance(position, m) do

end
end
```

Erlang Solution:

```
-spec max_distance(Position :: [integer()], M :: integer()) -> integer().
max_distance(Position, M) ->
.
```

Racket Solution:

```
(define/contract (max-distance position m)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```