1237. Find Positive Integer Solution for a Given Equation

Interactive

Problem Description

Medium Math Two Pointers Binary Search

monotonically increasing in both its arguments x and y. This means f(x, y) is always less than f(x+1, y) and f(x, y+1). Our task is to identify all pairs of positive integers (x, y) such that when the function f is applied to these pairs, the result equals a given target value z. Note that x and y must be positive integers. The final answer can be returned in any order. Since we don't know the exact formula of f, we cannot directly calculate the values. Therefore, we need to use the properties of

In this problem, we are given a special callable function f(x, y) that has a hidden formula. The function f is known to be

the function to systematically find all pairs (x, y) that satisfy f(x, y) == z. Intuition

• The function f outputs positive integers, and we are looking for positive integer inputs (x, y) as well. This means there will be a distinct and

finite number of solutions.

The key to solving this problem lies in understanding two main points:

Given these properties, one efficient way to find all pairs (x, y) that satisfy f(x, y) == z is by using a two-pointer technique. We start with the smallest possible value for x (which is 1) and the largest possible value for y within a reasonable range (for

• The function f is monotonically increasing which means as we increase x or y, the value of f(x, y) increases.

example, 1000) and evaluate f(x, y): • If f(x, y) < z, it means that with the current pair (x, y), the result is too small. Since f is monotonically increasing and we cannot decrease y

further (as we started with the maximum possible value), the only way to increase f(x, y) is by increasing x. Hence, we increment x. • If f(x, y) > z, the result is too large. We decrease y since decreasing y is certain to decrease the value of f(x, y). • If f(x, y) == z, we found a valid pair and we add it to our answer list. We then increment x and decrement y to find other possible pairs.

The algorithm continues this process until we exhaust all combinations up to the maximum value for both x and y. This approach leverages the function's monotonicity and avoids testing impossible combinations, leading to an efficient solution. Solution Approach

The implementation is based on the two-pointer technique, as mentioned in the intuition. Here's a detailed walkthrough: Initialize two pointers, x and y. Set x to 1, as we want to start with the smallest positive integer for x, and set y to 1000,

Use a while loop to iterate as long as x is less than or equal to 1000 and y is greater than 0. This ensures we consider all

assuming that the function f doesn't have x or y values greater than 1000. This is a reasonable assumption for these types of

Example Walkthrough

problems unless specified otherwise.

reasonable x and y combinations.

possible combinations of x and y within the given bounds.

value of f(x, y) due to the monotonic nature of the function).

Inside the loop, call customfunction f(x, y) with the current x and y variables to get the output of f. Check if the value of f(x, y) is less than the target value z. If it is, we need to increase x (since increasing x will increase the

decreasing y will decrease f(x, y). If f(x, y) equals z, we've found a valid pair, and we add [x, y] to the list ans. We then move both pointers—incrementing x

If f(x, y) is greater than z, then our current output is too high, and we need to reduce it. We do this by reducing y, as

- and decrementing y—to continue searching for other possible solutions. The loop exits once we either exceed the maximum value for x or y reaches 0. At this point, we would have tested all
- The algorithm effectively navigates the search space using the two-pointer technique, which exploits the monotonically increasing property of the function f. In terms of data structures, the implementation uses a list ans to store the pairs that fulfill the condition. No advanced data structures are needed since the problem is tackled primarily with algorithmic logic.

This approach is quite efficient, as it avoids unnecessary calculations for x and y pairs that obviously do not meet the condition.

By carefully increasing and decreasing x and y, it homes in on the correct pairs without checking every possible combination.

Note: The approach described as "Binary search" in the provided reference solution approach appears to be a mislabel, as the

actual implemented technique is, in fact, the two-pointer technique. Binary search would involve repeatedly halving the search space to find a solution, which isn't the case here.

to find all positive integer pairs (x, y) that satisfy the equation f(x, y) == z.

We enter our while loop and start evaluating f(x, y) to see if it matches z.

increasing x would increase the output of f. Hence, we move to x = 2.

Return the list ans containing all pairs [x, y] that satisfy f(x, y) == z.

Following our solution approach: We initialize x as 1 and y as 1000 (assuming f's range is within these bounds).

In our first iteration, we call f(1, 1000). If we find that f(1, 1000) < 14, we can conclude that we need to increase x because

We call f(2, 1000) and suppose we find f(2, 1000) > 14. The result is too high, meaning y must be decreased. Let's say we

We call f(2, 999) and find that f(2, 999) == 14. We've found a matching pair (2, 999), so we add it to our answer list.

We continue iterating, incrementing x to 3 and decrementing y to 998, and we call f(3, 998). If f(3, 998) < 14, we must

increase x. If f(3, 998) > 14, we must decrease y. If f(3, 998) = 14, we've found another valid pair.

The process repeats, traversing various (x, y) pairs and responding accordingly, until x > 1000 or y <= 0.

Through this example, it becomes clear how the two-pointer technique effectively pinpoints the valid (x, y) combinations

Let's illustrate the solution approach with an example. Suppose we have the special function f(x, y) and our target value z is 14.

We don't know the hidden formula within f, but we are given that f is monotonically increasing in both x and y, and we are tasked

We finish iterating and return the list ans that contains all our valid pairs [x, y].

guaranteed to miss the target z.

def f(self, x: int, y: int) -> int:

while x <= 1000 and y > 0:

y -= 1

x += 1

solutions.append([x, y])

// Import required package for using List and ArrayList

// Definition for a point (x, y) in the custom function

// Returns f(x, y) for any given positive integers x and y.

// Finds all the pairs (x, y) where the result of customfunction f(x, y) equals z

vector<vector<int>> solutions; // To store all the solution pairs

// Loop until x is less than or equal to 1000 and y is positive

// Compare the result with the target value

} else { // If the result is equal to targetValue

// You should not implement or assume the implementation of this class/interface.

* Finds all positive integer solutions to the equation customfunction f(x, y) = z.

* Note: The CustomFunction API is used to determine the result for the current x and y values.

* @param customfunction - An instance of CustomFunction that contains the definition for f(x, y).

++x; // If the result is less than z, increment x to increase the result.

// If the result is equal to z, we have found a valid solution.

x++; // Increment x to search for the next potential solution.

y--; // Decrement y to search for the next potential solution.

--y; // If the result is greater than z, decrement y to decrease the result.

solutions.push([x, y]); // Add the solution [x, y] to the solutions array.

vector<vector<int>> findSolution(CustomFunction& customFunction, int targetValue) {

// Start with the smallest possible value of x

x++; // If the result is less than the target, increment x to increase result

solutions.push_back($\{x, y\}$); // Add the current pair (x, y) to the solutions

y--; // If the result is greater than the target, decrement y to decrease result

// Method f should be defined in CustomFunction and it accepts two numbers x and y, and returns a number.

int result = customFunction.f(x, y); // Compute the custom function result

x++; // Increment x and decrement y to avoid repeating a solution

// Not to be implemented, just for reference

else:

Python

pass

solutions = []

x, y = 1, 1000

class Solution:

choose y = 999.

without redundant checks, by taking advantage of f's monotonicity. As such, assuming our y never needed to decrease below 995 and our x never rose above 10 before we finished, we might end up with a list ans that could look something like [[2, 999],

function f is unknown and merely illustrative to demonstrate the search process.)

which means f(x, y) < f(x + 1, y) and f(x, y) < f(x, y + 1).

Initialize an empty list to store the solutions.

def findSolution(self, custom_function: CustomFunction, target: int) -> List[List[int]]:

Start with the smallest possible value of x and the largest possible value of y.

We can increment x if the current function value is smaller than the target,

and decrement y if the current function value is greater than the target.

current_val = custom_function.f(x, y) # Compute the current value.

Perform a 2-pointer approach from both ends of the possible values.

Since the function is monotonic (increasing in both variables),

Proceed to the next potential solutions.

Solution Implementation

(Note: The actual results of f(2, 1000), f(2, 999), and f(3, 998) are arbitrary for the purposes of this example since the

[4, 997], [8, 995]]. The solution is efficient because it avoids unnecessarily checking combinations where f(x, y) would be

Define the interface for the CustomFunction, don't implement it. class CustomFunction: # The CustomFunction interface includes a method f(x, y) that takes two integers. # It is known that the function is increasing with respect to both x and y,

if current_val < target:</pre> # If the current value is less than the target, increase x. x += 1elif current_val > target: # If the current value is greater than the target, decrease y.

If the current value is equal to the target, we found a valid (x, y) pair.

```
# Return the list of valid (x, y) pairs that match the target value when plugged into custom_function.
        return solutions
Java
```

C++

public:

};

/**

TypeScript

class Solution {

int x = 1;

int y = 1000;

while $(x \le 1000 \&\& y > 0)$ {

// The CustomFunction API is predefined.

f(x: number, y: number): number;

declare class CustomFunction {

* Constraints: 1 <= x, y <= 1000

if (result < z) {</pre>

} else {

class CustomFunction:

} else if (result > z) {

while x <= 1000 and y > 0:

x += 1

y -= 1

x += 1

y -= 1

Time and Space Complexity

return solutions

else:

if current_val < target:</pre>

elif current_val > target:

solutions.append([x, y])

if (result < targetValue) {</pre>

} else if (result > targetValue) {

return solutions; // Return all solution pairs

import java.util.List;

class CustomFunction {

import java.util.Arrays;

import java.util.ArrayList;

```
// Note that f(x, y) is increasing with respect to both x and y.
   public int f(int x, int y) {
       // Implementation provided by system
       return -1; // Placeholder for the compiler
public class Solution {
    public List<List<Integer>> findSolution(CustomFunction customFunction, int target) {
       // Initialize a list to store pairs (x, y) that meet the condition f(x, y) == z
       List<List<Integer>> solutions = new ArrayList<>();
       // Set initial values for x and y to start at opposite ends of the range [1, 1000]
       int x = 1;
        int y = 1000;
       // Use a two-pointer approach to find all solution pairs
       while (x <= 1000 && y > 0) {
           // Evaluate the custom function for the current pair (x, y)
            int result = customFunction.f(x, y);
            if (result < target) {</pre>
                // If the result is less than target, increment x to increase the result
                X++;
           } else if (result > target) {
                // If the result is greater than target, decrement y to decrease the result
            } else {
                // If the result is equal to the target, add the pair (x, y) to the solutions list
                solutions.add(Arrays.asList(x, y));
                // Move both pointers to get closer to the next potential solution
                X++;
                y--;
       return solutions;
```

// Start with the largest possible value of y due to the property of the custom function

```
function findSolution(customfunction: CustomFunction, z: number): number[][] {
   let x = 1; // Initialize x to start from 1.
   let y = 1000; // Initialize y to start from 1000.
   const solutions: number[][] = []; // Array to store the pairs [x, y] satisfying the equation.
   // Iterate as long as x and y remain within their respective bounds.
   while (x \le 1000 \&\& y >= 1) {
       const result = customfunction.f(x, y); // Evaluate f(x, y) using the custom function.
```

return solutions; // Return the array of solutions.

Define the interface for the CustomFunction, don't implement it.

which means f(x, y) < f(x + 1, y) and f(x, y) < f(x, y + 1).

* @param z - The result for which solutions are being searched.

* @returns An array of integer pairs (arrays) [x, y] where f(x, y) = z.

```
def f(self, x: int, y: int) -> int:
        pass
class Solution:
   def findSolution(self, custom_function: CustomFunction, target: int) -> List[List[int]]:
       # Initialize an empty list to store the solutions.
        solutions = []
       # Start with the smallest possible value of x and the largest possible value of y.
       x, y = 1, 1000
```

Perform a 2-pointer approach from both ends of the possible values.

We can increment x if the current function value is smaller than the target,

and decrement y if the current function value is greater than the target.

current_val = custom_function.f(x, y) # Compute the current value.

If the current value is less than the target, increase x.

If the current value is greater than the target, decrease y.

If the current value is equal to the target, we found a valid (x, y) pair.

Return the list of valid (x, y) pairs that match the target value when plugged into custom_function.

Since the function is monotonic (increasing in both variables),

The CustomFunction interface includes a method f(x, y) that takes two integers.

It is known that the function is increasing with respect to both x and y,

Time Complexity The given code has a while loop that iterates at most min(1000, z) times because:

Proceed to the next potential solutions.

 For every iteration, either x is incremented or y is decremented. • The loop stops when x exceeds 1000 or y reaches below 1.

• The maximum possible value for x and y is 1000 (as per the constraints).

consider the worst-case scenario where it takes a constant time), the time complexity of the entire operation is 0(min(1000, z)).

Space Complexity The space complexity mainly depends on the number of valid (x, y) pairs that satisfy the equation f(x, y) = z. In the worst

case, all pairs (x, y) where $1 \le x, y \le 1000$ could be solutions, so there could be as many as 1000 solutions.

Therefore, the space complexity of the code is 0(1).

Assuming that the custom function f(x, y) is O(1) (since we don't have information about its internal implementation, we

However, since the space used by ans depends on the output, which we do not count towards the space complexity in the analysis, the additional space used by the algorithm (i.e., for variables x, y, t) is constant.