1362. Closest Divisors

Medium Math

Problem Description

The problem presents a scenario where you are given a single integer `num`. Your task is to find two integers whose product is either `num + 1` or `num + 2`. These two integers should be as close to each other as possible in terms of their absolute difference – meaning the difference between the two numbers without considering whether it is positive or negative should be minimal. The final solution does not require the integers to be in any specific order, thus either of the integers can come first.

Intuition

In solving this problem, the concept of factors of a number is key. For any given number, its factors are the numbers that divide it without leaving a remainder. In this case, our target numbers are `num + 1` and `num + 2`. We look for the pair of factors, for each of these numbers, that are closest to each other. The intuition here is that the pair of factors that are closest to each other will have the smallest absolute difference.

To efficiently find such a pair for a given target number, we can start checking from the square root of the target number and move downwards. The square root gives us a good starting point since it is the largest number that can multiply by itself to not exceed the target. Therefore, any factor larger than the square root would result in a product larger than our target number, disqualifying it from being a correct answer.

Once we have the factors for both `num + 1` and `num + 2`, we compare their absolute differences. The pair with the smaller absolute difference is the closest pair – this represents our final answer. The reason for checking both `num + 1` and `num + 2` is to fulfill the task's requirement of finding the closest integers in absolute difference.

Solution Approach

The solution uses a straightforward approach by defining a helper function `f(x)` which takes a number `x` (which will either be `num + 1` or `num + 2`) and finds the closest divisors of `x`.

Here's how the `f(x)` function works in detail:

1. Start by calculating the integer square root of `x` using `int(sqrt(x))`. This is the starting point for finding our factors.
2. Iterate downwards from this square root to 1 (inclusive), as any factor greater than the square root of `x` would result in a product larger than `x` when multiplied with another integer.
3. In each iteration, check if `x % i == 0` which is the condition to confirm if `i` is a factor of `x`.
4. Once a factor is found, calculate the pair by dividing `x` by the factor `i`. The pair will be `[i, x // i]`.
5. Return the pair of factors found.

The main function `closestDivisors(num: int) -> List[int]` calls this helper function twice: once with `num + 1` and once with `num + 2`.

1. The function stores the pairs returned by `f(num + 1)` and `f(num + 2)` in variables `a` and `b` respectively.
2. It then compares the absolute differences of the two integers in each pair.
3. The pair with the smaller absolute difference is chosen as the result.
4. The result pair is returned.

Algorithmically, this is an efficient approach because it only searches up to the square root of the target numbers rather than iterating through all possible divisors, which significantly reduces the number of operations, especially for very large numbers.

To summarize, the solution algorithm involves:

- Finding pairs of factors (divisors) for two numbers `num + 1` and `num + 2`.
- Starting the search from the square root of these numbers and iterating downwards to find these pairs.
- Returning the pair with the smallest absolute difference between the two factors.

Data structures used in this solution are basic and include primarily lists to store the pairs of factors. The pattern utilized here is an optimization over brute force where only necessary divisors are considered, which is made possible by the mathematical property that a number's divisors are symmetrical around the square root.

Example Walkthrough

Let's assume `num = 8`. We want to find two integers whose product is `num + 1` or `num + 2`, which here would be `9` and `10`. These two integers should have the smallest possible absolute difference. Let's apply our solution approach:

1. We first calculate the square root of `num + 1` which is `sqrt(9) = 3`.
2. Then, we check if there's any number from 3 down to 1 that evenly divides 9:
   - 3 is a divisor of 9 since `9 % 3 == 0`.
   - The pair is formed by `3` (the divisor) and `9 // 3 = 3`. So the factor pair for `num + 1` is `[3, 3]`.
3. Next, we calculate the square root of `num + 2`, which is slightly over `sqrt(10)` but we use `int(sqrt(10)) = 3` as our starting point.
4. We check divisors from 3 to 1 for the number 10:
   - 3 is not a divisor of 10 since `10 % 3 != 0`.
   - 2 is a divisor since `10 % 2 == 0`.
   - The pair for `num + 2` is `[2, 10 // 2]`, which is `[2, 5]`.
5. Now we have two pairs: `[3, 3]` for `num + 1` and `[2, 5]` for `num + 2`. We compare their absolute differences:
   - The absolute difference for `[3, 3]` is `3 - 3 = 0`.
   - The absolute difference for `[2, 5]` is `5 - 2 = 3`.
6. We choose the pair with the smallest absolute difference, which is `[3, 3]`, with an absolute difference of 0.
7. The final result is `[3, 3]`, indicating that these are the two integers (which in this case happen to be identical) that, when multiplied together, yield `num + 1` (which is 9) and have the smallest absolute difference.

This example walk-through demonstrates the solution approach. By checking only the divisors from the square root and below, we quickly identify the closest pair of numbers satisfying the condition without unnecessary computation.

Solution Implementation

Python

```python
from typing import List
from math import sqrt

class Solution:
    def closestDivisors(self, num: int) -> List[int]:
        # Define a helper function to find the pair of divisors
        # of a number 'x' that are closest to each other.
        def find_closest_divisors(x):
            # Start by finding the square root of 'x' and iterate backwards
            for i in range(int(sqrt(x)), 0, -1):
                # If 'i' is a divisor of 'x'
                if x % i == 0:
                    # Return the divisor pair [i, x // i]
                    return [i, x // i]

        # Find the closest divisors for 'num + 1'
        closest_divisors_num_plus_one = find_closest_divisors(num + 1)
        # Find the closest divisors for 'num + 2'
        closest_divisors_num_plus_two = find_closest_divisors(num + 2)

        # Compare which pair of divisors has the smallest difference
        # and return that pair.
        if abs(closest_divisors_num_plus_one[0] - closest_divisors_num_plus_one[1]) < abs(closest_divisors_num_plus_two[0] - closest_
            return closest_divisors_num_plus_one
        else:
            return closest_divisors_num_plus_two
```

Java

```java
class Solution {

    // This function finds two closest divisors of the input number 'num'
    public int[] closestDivisors(int num) {
        // Find the closest divisors for the number 'num + 1'
        int[] divisorsNumPlusOne = findClosestDivisors(num + 1);
        // Find the closest divisors for the number 'num + 2'
        int[] divisorsNumPlusTwo = findClosestDivisors(num + 2);

        // Compare abs difference of divisors pairs and return the pair with the smallest difference
        if (Math.abs(divisorsNumPlusOne[0] - divisorsNumPlusOne[1]) <
            Math.abs(divisorsNumPlusTwo[0] - divisorsNumPlusTwo[1])) {
            return divisorsNumPlusOne;
        } else {
            return divisorsNumPlusTwo;
        }
    }

    // Helper function that calculates the two closest divisors of 'x'
    private int[] findClosestDivisors(int x) {
        // Start from the square root of 'x' and check for the closest divisors by moving downwards
        for (int i = (int) Math.sqrt(x); true; --i) { // the condition is always true, it breaks inside if a divisor pair is found
            // If 'i' divides 'x' with no remainder, 'i' and 'x / i' are divisors of 'x'
            if (x % i == 0) {
                // Found the closest divisors, return them in an array
                return new int[] {i, x / i};
            }
        }
    }
}
```

C++

```cpp
#include <vector>
#include <cmath> // Include cmath for sqrt function

class Solution {
public:
    // Function to find the closest divisors of an integer 'num'
    vector<int> closestDivisors(int num) {
        // Lambda function to find the divisors closest to the square root of 'x'
        auto findClosestDivisors = [](int x) -> vector<int> {
            // Start from the largest possible factor that could be closest to sqrt of x
            for (int i = sqrt(x); i > 0; --i) { // Ensure i is always positive
                if (x % i == 0) {
                    // If i is a divisor, return the pair (i, x/i)
                    return vector<int>{i, x / i};
                }
            }
            // This code should never reach here since every number has at least one pair of divisors
        };

        // Find the closest divisors for both num+1 and num+2
        vector<int> divisorsForNumPlusOne = findClosestDivisors(num + 1);
        vector<int> divisorsForNumPlusTwo = findClosestDivisors(num + 2);

        // Determine which pair of divisors has the smallest difference
        // Return the pair with the smallest difference
        return abs(divisorsForNumPlusOne[0] - divisorsForNumPlusOne[1]) < abs(divisorsForNumPlusTwo[0] - divisorsForNumPlusTwo[1]) ?
    }
};

// Remember to include necessary headers before using this code.
```

TypeScript

```typescript
// Import sqrt function from Math module
import { sqrt, abs } from 'math';

// Function to find closest divisors of an integer 'num'
function closestDivisors(num: number): number[] {
    // Lambda function to find the divisors closest to the square root of 'x'
    const findClosestDivisors = (x: number): number[] => {
        // Start from the largest possible factor that could be closest to sqrt of x
        for (let i = Math.floor(sqrt(x)); i > 0; --i) { // Ensure i is always positive
            if (x % i === 0) {
                // If i is a divisor, return the pair [i, x/i]
                return [i, x / i];
            }
        }
        // Since every number has at least one pair of divisors, this line should not be reached.
        throw new Error("No divisors found"); // To handle edge cases theoretically unreachable
    };

    // Find the closest divisors for both num+1 and num+2
    const divisorsForNumPlusOne: number[] = findClosestDivisors(num + 1);
    const divisorsForNumPlusTwo: number[] = findClosestDivisors(num + 2);

    // Determine which pair of divisors has the smallest difference, and return the pair with the smallest difference
    return abs(divisorsForNumPlusOne[0] - divisorsForNumPlusOne[1]) < abs(divisorsForNumPlusTwo[0] - divisorsForNumPlusTwo[1])
        ? divisorsForNumPlusOne
        : divisorsForNumPlusTwo;
}
```


```python
from typing import List
from math import sqrt

class Solution:
    def closestDivisors(self, num: int) -> List[int]:
        # Define a helper function to find the pair of divisors
        # of a number 'x' that are closest to each other.
        def find_closest_divisors(x):
            # Start by finding the square root of 'x' and iterate backwards
            for i in range(int(sqrt(x)), 0, -1):
                # If 'i' is a divisor of 'x'
                if x % i == 0:
                    # Return the divisor pair [i, x // i]
                    return [i, x // i]

        # Find the closest divisors for 'num + 1'
        closest_divisors_num_plus_one = find_closest_divisors(num + 1)
        # Find the closest divisors for 'num + 2'
        closest_divisors_num_plus_two = find_closest_divisors(num + 2)

        # Compare which pair of divisors has the smallest difference
        # and return that pair.
        if abs(closest_divisors_num_plus_one[0] - closest_divisors_num_plus_one[1]) < abs(closest_divisors_num_plus_two[0] - closest_
            return closest_divisors_num_plus_one
        else:
            return closest_divisors_num_plus_two
```


Time and Space Complexity

Time Complexity

The time complexity of the function primarily depends on the `for` loop within the nested function `f(x)`, which iterates over numbers starting from `int(sqrt(x))` to `1`.

Since the square root function essentially reduces the number of iterations to the square root of `x`, the time complexity for finding the divisors would be `O(sqrt(x))`.

Given that the function `f(x)` is called twice—once for `num + 1` and once for `num + 2`—the overall time complexity is then `O(sqrt(num + 1) + sqrt(num + 2))`, which simplifies to `O(sqrt(num))`, as the higher order term dominates and adding 1 or 2 to `num` has a negligible effect for large numbers.

Space Complexity

The space complexity of this algorithm is `O(1)`, as the space used does not grow with the input size `num`. The only extra space used is for a handful of variables that store the divisor pairs and their differences.