1387. Sort Integers by The Power Value

Medium Memoization Dynamic Programming Sorting Leetcode Link

Problem Description

In this problem, we are introduced to a special mathematical transformation of an integer `x`, which ultimately reduces `x` to `1` through a series of steps. These steps consist of halving `x` if it is even, or replacing `x` with `3 * x + 1` if it is odd. The "power" of an integer `x` is defined as the number of these steps required to reduce `x` to `1`.

Given three integers `lo`, `hi`, and `k`, the goal is to find the `kth` smallest integer in the range `[lo, hi]` when all the integers within this range are sorted by their power values first (ascending), and by their natural values next (also ascending) if they have the same power value.

Consider an example where `lo = 3`, `hi = 5`, and `k = 2`. Power values for integers 3, 4 and 5 are 7, 2, and 5 respectively. After sorting them by their power values, we get the sequence 4, 5, 3. The second integer in this sequence is 5, which would be the output for these input values.

It's guaranteed in this problem that the power value for any integer `x` within the range will fit within a 32-bit signed integer, and that `x` will eventually transform to `1` through the given steps.

Intuition

To solve this problem, we need a function that computes the power of any given number `x`. The provided solution encapsulates this in a function `f(x)`, which is decorated with `@cache` to store the results of computation for each unique value of `x`. This prevents repeated computation and improves efficiency if the same number appears multiple times during the sorting process.

Once we have the function to compute the power of an integer, we can generate all integers within the range `[lo, hi]` and sort them primarily by the computed power values and secondarily by their natural values in case of ties in power values. This is achieved by passing the custom sorting key `f` to Python's built-in `sorted` function. After sorting, we simply return the `k-1` indexed element from this sorted list (since lists are zero-indexed in Python).

The solution is quite straightforward:

1. Compute the power for each number in the range using the `f(x)` function.
2. Cache the result each time it's computed to improve efficiency for repeated calculations.
3. Sort all numbers by their power values, and if there's a tie, by their natural values.
4. Return the `kth` number in the sorted sequence.

Solution Approach

The implementation of the solution breaks down into the creation of a helper function `f(x)` and using built-in Python features to sort and retrieve the desired `kth` element.

First, let's discuss the helper function `f(x)`:

Helper Function `f(x)`

The helper function `f(x)` computes the power of a number `x` according to the rules given in the problem statement. It initiates a counter (`ans`) that tracks the number of steps taken to reduce `x` to `1`. Inside a `while` loop, as long as `x` is not `1`, the function checks if `x` is even or odd. If it's even, `x` is divided by `2`, otherwise it's replaced with `3 * x + 1`, as described in the problem. The counter is incremented after each operation.

A key optimization implemented here is the use of the `@cache` decorator. This decorator from the `functools` module caches the result of the power calculation for each unique input `x`. Later calls to the `f(x)` function with the same `x` value fetch the result from the cache rather than recomputing it, significantly saving time especially for ranges with repeated power sequences.

Sort and Retrieve `kth` Element

The `Solution` class has a method `getKth(lo, hi, k)` which leverages the `f(x)` function to accomplish the task. Here is the step by step approach:

1. Generate a list of integers from `lo` to `hi`, inclusive. This list represents all the candidates for sorting.

2. Use the `sorted` function to sort this range, passing `f` as the key argument. By doing this, Python will call `f(x)` on each element of the range and sort the numbers according to the values returned by `f(x)`. In case of a tie in power values, the `sorted` function defaults to sorting by the numbers themselves, maintaining them in ascending order.

   The `sorted` function has a time complexity of O(n log n), where `n` represents the length of the list being sorted. Given that the cache is in place, each power computation (beyond the first instance) takes O(1) time thanks to the memoization.

3. Once the list is sorted, we are interested in the `kth` value of this sorted sequence. We simply index into the sorted list with `k - 1` (to account for 0-based indexing) and return this value.

4. The final returned value is the `kth` smallest element by power value, and by numerical order in case of identical powers.

The algorithm's main time complexity is driven by the sorting step which is O(n log n). The space complexity includes O(n) for the list of integers and the additional space required by the cache, which at most will be O(n) for a range of different numbers requiring distinct power calculations.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we are given the input `lo = 6`, `hi = 10`, and `k = 3`. We aim to find the 3rd smallest integer by power value in the range [6, 10]. We will calculate the power for each integer within the range and sort them accordingly.

First, we need to compute the power for each integer:

- The power of 6: 6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1 (total 8 steps).
- The power of 7: 7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → ... (we have already calculated this from 5: 5 steps). Hence, the total is 12 + 5 = 17 steps.
- The power of 8: 8 → 4 → 2 → 1 (total 3 steps).
- The power of 9: 9 → 28 → 14 → 7 → ... (we have already calculated this from 7: 17 steps). So the total is 3 + 17 = 20 steps.
- The power of 10: 10 → 5 → ... (we have already calculated from 5: 5 steps). Hence, the total is 1 + 5 = 6 steps.

Next, we sort these integers by their power values:

- Power values in ascending order: 3 (8), 6 (8), 10 (6), 7 (17), 9 (20).
- Since 3 and 6 have the same power value (8), they will be sub-sorted by their natural values.

So the sorted list by power and natural values is: 6, 8, 10, 7, 9. We want the 3rd smallest integer by power value, which is the integer at index 2 in the sorted list (keeping in mind 0-based indexing). Therefore, the answer is 10.

We achieved this using the following steps:

1. Create the helper function `f(x)` with the `@cache` decorator to calculate power values and make subsequent calculations faster by using cached results.

2. Generate the list of integers from `lo` to `hi` inclusive, which in our case is [6, 7, 8, 9, 10].

3. Use Python's `sorted` function with `f` as the key argument to sort the integers by their power values and by their natural values if there are ties.

4. Index into the sorted array with `k - 1`, which is `3 - 1 = 2` in our example, to get the 3rd smallest item according to the defined criteria. Our final result is the integer `10`.

This walkthrough exemplifies how the given solution approach effectively finds the `kth` smallest integer by its "power" value within a given range.

Python Solution

```python
from functools import lru_cache

# Decorator to cache the results of the function to avoid recalculation
@lru_cache(maxsize=None)
def compute_steps(x: int) -> int:
    # Initialize the number of steps taken to reach 1
    steps = 0
    # Loop until x becomes 1
    while x != 1:
        # If x is even, halve it
        if x % 2 == 0:
            x //= 2
        # If x is odd, apply 3x + 1
        else:
            x = 3 * x + 1
        # Increment the step count for each iteration
        steps += 1
    # Return the total number of steps taken
    return steps

class Solution:
    def getKth(self, lo: int, hi: int, k: int) -> int:
        # Sort the range [lo, hi] by the number of steps to reach 1
        # for each number, as determined by the compute_steps function
        # Then, retrieve the k-th element in this sorted list.
        return sorted(range(lo, hi + 1), key=compute_steps)[k - 1]

```

Java Solution

```java
class Solution {

    // Method to get the k-th integer in the range [lo, hi] after sorting by a custom function
    public int getKth(int lo, int hi, int k) {
        // Create an array of Integers to store the range [lo, hi]
        Integer[] numbers = new Integer[hi - lo + 1];

        // Fill the array with numbers from lo to hi
        for (int i = lo; i <= hi; ++i) {
            numbers[i - lo] = i;
        }

        // Sort the array using a custom comparator based on the transformation count
        Arrays.sort(numbers, (a, b) -> {
            int transformCountA = computeTransformCount(a);
            int transformCountB = computeTransformCount(b);
            if (transformCountA == transformCountB) {
                return a - b; // If counts are equal, sort by natural order
            } else {
                return transformCountA - transformCountB; // Otherwise, sort by the count
            }
        });

        // Return the k-th element after sorting
        return numbers[k - 1];
    }

    // Helper method computing the number of steps to reach 1 based on the Collatz conjecture
    private int computeTransformCount(int x) {
        int count = 0; // Initialize step count

        // While x is not 1, apply the Collatz function and increase the count
        while (x != 1) {
            if (x % 2 == 0) {
                x /= 2; // If x is even, divide it by 2
            } else {
                x = x * 3 + 1; // If x is odd, multiply by 3 and add 1
            }
            count++;
        }

        return count;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function computes the k-th integer in the range [lo, hi] such that
    // an integer x's power value is defined as the minimum number of steps
    // to reach 1 when we start from x and iteratively apply the following operations:
    // If x is even, replace it with x / 2.
    // If x is odd, replace it with 3 * x + 1.
    int getKth(int lo, int hi, int k) {
        // Lambda function to calculate the power value of a number
        auto calculatePower = [](int x) {
            int power = 0;
            while (x != 1) {
                x = (x % 2 == 0) ? x / 2 : 3 * x + 1;
                ++power;
            }
            return power;
        };

        // Vector to store the range of numbers from lo to hi
        vector<int> numbers;
        for (int i = lo; i <= hi; ++i) {
            numbers.push_back(i);
        }

        // Sort the range of numbers based on their power value
        // If two numbers have the same power value, the smaller number comes first
        sort(numbers.begin(), numbers.end(), [&](int x, int y) {
            int powerX = calculatePower(x);
            int powerY = calculatePower(y);
            return (powerX != powerY) ? powerX < powerY : x < y;
        });

        // Return the k-th element in the sorted list
        return numbers[k - 1];
    }
};

```

Typescript Solution

```typescript
// Define a helper function to calculate the power value for a given integer.
// The power value is the number of steps to transform the integer to 1 using the following process:
// If x is even, divide it by 2.
// If x is odd, multiply by 3 and add 1.
function calculatePowerValue(x: number): number {
    let steps = 0;
    while (x !== 1) {
        steps++;
        if (x % 2 === 0) {
            x >>= 1; // If x is even, right shift to divide by 2.
        } else {
            x = x * 3 + 1; // If x is odd, multiply by 3 and then add 1.
        }
    }
    return steps;
}

// Function to get the k-th integer in the range [lo, hi] after sorting by their power value.
// If two integers have the same power value, then sort them by numerical value.
function getKth(lo: number, hi: number, k: number): number {
    // Create an array of all integers from lo to hi, inclusive.
    const nums: number[] = new Array(hi - lo + 1).fill(0).map((_, index) => lo + index);

    // Sort the array based on the power value calculated via the helper function.
    // If two integers have the same power value, they will be sorted numerically instead.
    nums.sort((a, b) => {
        const powerA = calculatePowerValue(a);
        const powerB = calculatePowerValue(b);
        if (powerA === powerB) {
            return a - b; // Sort by numerical value if power values are equal.
        }
        return powerA - powerB; // Sort by power value.
    });

    // Return the k-th element in the sorted array, considering array indices start at 0.
    return nums[k - 1];
}

```

Time and Space Complexity

The time complexity and space complexity of the method `getKth` in the `Solution` class can be understood in parts.

Firstly, the function `f` is decorated with `@cache`, which uses memoization to store the results of inputs to avoid redundant calculations. The complexity of function `f` without memoization is tough to quantify precisely, as it requires understanding the behavior of the Collatz sequence, which is a longstanding unsolved problem in mathematics. Nevertheless, each call to the uncached function `f` performs a series of operations proportional to the length of the sequence required to reach 1, starting from `x`. During this sequence, the "halving" operation (`x //= 2`) may occur `O(log x)` times, while the "tripling plus one" operation (`x = 3 * x + 1`) is more unpredictable. Despite this, for our complexity analysis, we can assume a worst-case scenario where these operations would result in a complexity of `O(log x)` on average per function call due to memoization, as previously computed results would be reused.

The method `getKth` sorts the range `[lo, hi]`, inclusive. Let `n = hi - lo + 1` be the number of elements in this range. The sorted function applies the `f` function as a key, and because of the caching of `f`, each unique call is `O(log x)`. However, because we are sorting `n` elements, the sorting operation has a time complexity of `O(n log n)`. Since each element requires calling function `f`, and assuming `f` costs `O(log hi)` at most given its upper bound, the overall time complexity is `O(n log n * log hi)`.

The space complexity primarily comes from two sources: the memoization cache and the sorted list. The cache's size is unbounded and depends on the range of numbers and the length of the Collatz sequences encountered—it can potentially reach `O(hi)` in space due to the range of unique values passed to `f`. However, if many values share similar Collatz paths, which is quite possible, the actual memory usage may be significantly less. The sorted list has a space complexity of `O(n)`. Thus, the overall space complexity may be approximated as `O(hi)` given the potentially large cache size.

In summary:

- Time Complexity: `O(n log n * log hi)`
- Space Complexity: `O(hi)`