786. K-th Smallest Prime Fraction

Medium Array Binary Search Sorting Heap (Priority Queue) Leetcode Link

Problem Description

You are provided with a sorted array named `arr`, which consists of the number 1 and other prime numbers. All elements in this array are unique. Alongside this, you are given an integer `k`. Taking every combination of elements `i` and `j` from the array—where the index `i` is less than the index `j`—you form fractions of the form `arr[i] / arr[j]`. Your task is to find the `k`th smallest fraction among all the possible fractions you can create this way. The result should be returned as an array with two elements: `[arr[i], arr[j]]`, representing the numerator and denominator of the determined fraction, respectively.

Intuition

To solve this problem, we leverage a min-heap to efficiently track and sort fractions according to their value. The min-heap is a data structure that allows us to always have access to the smallest element. By pushing all possible fractions formed by the first element of the array as the numerator and all other elements as denominators into the heap, we get all the smallest possible fractions with `1 / arr[j]`.

The min-heap is initialized with tuples containing the fraction `arr[i] / arr[j]`, as well as the indices `i` and `j`. This initial population of the heap starts with `i` fixed at `0` and `j` ranging over all valid indices, which ensures that all the smallest fractions are considered first.

Once the heap is populated, the following steps are taken:

1. We pop out the smallest element from the heap.
2. Since each fraction `arr[i] / arr[j]` is formed by considering all the possible `j` for a particular `i`, we need to consider the next possible fraction for the current smallest `i`. This means if `arr[i] / arr[j]` was the smallest, we now need to consider `arr[i + 1] / arr[j]`, ensuring `i + 1 < j` to maintain our fraction condition `i < j`.
3. We then push the new fraction `arr[i + 1] / arr[j]` into the heap.
4. This process is repeated `k - 1` times because every pop operation retrieves the smallest fraction at the moment, and we want the `k`th smallest.

After repeating this process `k - 1` times, the top of the heap contains the `k`th smallest fraction. We then return this fraction as `[numerator, denominator]` using the indices stored in the heap tuple to access elements from the `arr`.

The reason we don't initialize the heap with all possible fractions is because it would be inefficient. Since the array is sorted, the smallest fractions are formed with the smallest denominator. Hence, we initially consider these and incrementally add fractions with larger numerators, leveraging the heap's sorting property to efficiently find the `k`th smallest fraction.

Solution Approach

The solution involves multiple concepts, primarily heap data structure operations and basic arithmetic. Let's walk through the implementation:

- **Initialize the Heap**: The first step involves initializing a min-heap with tuples. Each tuple contains three elements:
  1. The value of the fraction `arr[i] / arr[j]`.
  2. The index `i` of the numerator.
  3. The index `j` of the denominator.

  Using Python's `heapq` module, we create a min-heap because it allows us to easily push and pop the smallest elements. The heap is populated with the reciprocal of all elements of `arr` starting from the second element because the list of primes is sorted and fractions with 1 as a numerator will be the smallest.

- **Heapify**: The `heapify` function converts the list into a heap structure. This step is essential to maintain the heap properties after the initial insertion of elements.

- **Iteration and Heap Operations**: The core logic of the heap manipulation happens inside a loop that runs `k-1` times. This loop represents the iteration to find the `k`th smallest element:

```python
for _ in range(k - 1):
    frac, i, j = heappop(h)
    if i + 1 < j:
        heappush(h, (arr[i + 1] / arr[j], i + 1, j))
```

  In each iteration, we:

  - Extract (`heappop`) the smallest element from the heap. The smallest element corresponds to the currently smallest fraction.
  - Check if we can form a new fraction by incrementing the numerator's index `i`. If `i + 1 < j`, it means there is another fraction to consider.
  - If the new fraction can be formed, push (`heappush`) the new fraction `arr[i + 1] / arr[j]` along with its indices back into the heap.

- **Extracting the Result**: After the loop has completed `k-1` iterations, the smallest element remaining on top of the heap is the `k`th smallest fraction. We extract the indices `i` and `j` from the top of the heap and use them to return the result as `[arr[i], arr[j]]`.

- **Return the `k`th Smallest Fraction**: Finally, the indices from the tuple that's at the top of the heap after `k-1` pops represent the `k`th smallest fraction, and the final return statement `return [arr[h[0][1]], arr[h[0][2]]]` fetches the `numerator` and `denominator` from the `arr`.

In summary, the algorithm efficiently keeps track of the potential `k`th smallest fraction at each step by using a min-heap to ensure that the smallest possible fraction is always available for comparison, without the need to calculate and store all possible fractions at the beginning. It's an elegant solution that combines heap operations with the sorted property of the input array to provide an efficient answer.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Assume we have the sorted array `arr` containing prime numbers, which looks like this: `[1, 2, 3, 5]`, and we want to find the `k`th smallest fraction where `k=3`.

The initial step is to initialize a min-heap and populate it with fractions having `1` as the numerator. So initially, our heap looks like this, containing the value of the fraction and the indices (numerator index, denominator index):

- `[(0.5, 0, 1), (0.333..., 0, 2), (0.2, 0, 3)]`

We run the heapify process to ensure it's a valid min-heap (although in this case, it's already a min-heap because we started with the smallest possible fractions):

After the heapify process, our heap remains the same (it's already in heap order):

- `[(0.5, 0, 1), (0.333..., 0, 2), (0.2, 0, 3)]`

Now we want to find the 3rd smallest fraction. We need to perform `k-1` operations on the heap.

1. **First iteration**:
   - We pop out the smallest fraction, which is `(0.2, 0, 3)` corresponding to fraction `1/5`.
   - Next, we check whether we can form a new fraction by incrementing the numerator's index. However, since there is not an index `i + 1` for `i=0` that is less than `j=3`, we do not push a new fraction to the heap.
   - The heap after the first pop is: `[(0.333..., 0, 2), (0.5, 0, 1)]`.
2. **Second iteration**:
   - We pop the next smallest fraction, which is `(0.333..., 0, 2)` corresponding to the fraction `1/3`.
   - We verify if a new fraction can be formed with `i+1` where `i=0` before `j=2`, but since `1 < 2`, we cannot increment `i` to get a valid new fraction while keeping `i < j`.
   - The heap after the second pop is: `[(0.5, 0, 1)]`.

Since we've done `k-1` operations (here k=3, so we did 2 operations), the top of the heap now contains the 3rd smallest fraction. We now have the smallest fraction on top of the heap as `(0.5, 0, 1)` which corresponds to the fraction `1/2`.

The result for `k=3` would be `[arr[0], arr[1]]` which translates to `[1, 2]`. Thus, the `3rd smallest fraction` formed by elements from `arr` is `1/2`.

By following this process, we efficiently find the `k`th smallest fraction without creating a full list of fractions at the beginning and instead only maintaining a heap of the smallest fractions at any given time, minimizing memory usage and computation.

Python Solution

```python
from heapq import heapify, heappop, heappush
from typing import List

class Solution:
    def kthSmallestPrimeFraction(self, primes: List[int], k: int) -> List[int]:
        # Create a min-heap of tuples, with each tuple containing the fraction,
        # the index of the numerator, and the index of the denominator.
        min_heap = [(primes[0] / primes[j], 0, j) for j in range(1, len(primes))]

        # Convert the list into a heap in-place.
        heapify(min_heap)

        # Pop the smallest fraction from the heap 'k - 1' times,
        # since we need to find the kth smallest fraction.
        for _ in range(k - 1):
            # Pop the smallest element (fraction) from the heap.
            smallest_fraction, i, j = heappop(min_heap)

            # If we can move the numerator to the right in the array to get
            # another fraction with the same denominator, push that fraction to the heap.
            if i + 1 < j:
                new_numerator_index = i + 1
                new_fraction = (primes[new_numerator_index] / primes[j], new_numerator_index, j)
                heappush(min_heap, new_fraction)

        # After popping k-1 elements, the smallest fraction in the min-heap
        # is the kth smallest fraction. Return this fraction as [numerator, denominator].
        smallest_fraction, numerator_index, denominator_index = min_heap[0]
        return [primes[numerator_index], primes[denominator_index]]

```

Java Solution

```java
import java.util.PriorityQueue; // Import PriorityQueue from Java's utility library

class Solution {
    // Method to find the kth smallest prime fraction within an array
    public int[] kthSmallestPrimeFraction(int[] arr, int k) {
        int n = arr.length; // Get the length of the array

        // Create a PriorityQueue to hold Frac (Fraction) objects, ordered by their fraction value
        PriorityQueue<Fraction> priorityQueue = new PriorityQueue<>();

        // Initialize the priority queue with the smallest prime fractions
        for (int i = 1; i < n; i++) {
            priorityQueue.offer(new Fraction(arr[0], arr[i], 0, i));
        }

        // Poll the queue k-1 times to get the kth smallest prime fraction
        for (int count = 1; count < k; count++) {
            Fraction fraction = priorityQueue.poll();
            if (fraction.numeratorIndex + 1 < fraction.denominatorIndex) {
                // Insert the next fraction with the same denominator and the next greater numerator
                priorityQueue.offer(new Fraction(arr[fraction.numeratorIndex + 1], arr[fraction.denominatorIndex],
                                                 fraction.numeratorIndex + 1, fraction.denominatorIndex));
            }
        }
        Fraction kthSmallestFraction = priorityQueue.peek(); // Get the kth smallest prime fraction

        // Return the numerator and denominator of the kth smallest fraction
        return new int[] {kthSmallestFraction.numerator, kthSmallestFraction.denominator};
    }

    // Inner class to represent a fraction, implement Comparable to sort in PriorityQueue
    static class Fraction implements Comparable<Fraction> {
        int numerator, denominator; // Numerator and denominator of the fraction
        int numeratorIndex, denominatorIndex; // Indices of the numerator and denominator in the array

        // Constructor for Fraction class
        public Fraction(int numerator, int denominator, int numeratorIndex, int denominatorIndex) {
            this.numerator = numerator;
            this.denominator = denominator;
            this.numeratorIndex = numeratorIndex;
            this.denominatorIndex = denominatorIndex;
        }

        // Override the compareTo method to define the natural ordering of Fraction objects
        @Override
        public int compareTo(Fraction other) {
            // Fraction comparison by cross multiplication to avoid floating point operations
            return this.numerator * other.denominator - other.numerator * this.denominator;
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    std::vector<int> kthSmallestPrimeFraction(std::vector<int>& arr, int k) {
        // Alias for pair of ints for easier readability
        using Pair = std::pair<int, int>;

        // Custom comparator for the priority queue that will compare fractions
        auto compare = [&](const Pair& a, const Pair& b) {
            return arr[a.first] * arr[b.second] > arr[a.second] * arr[b.first];
        };

        // Define a priority queue with the custom comparator
        std::priority_queue<Pair, std::vector<Pair>, decltype(compare)> pq(compare);

        // Initialize the priority queue with fractions {0, i} (0 < i)
        for (int i = 1; i < arr.size(); ++i) {
            pq.push({0, i});
        }

        // Pop k-1 elements from the priority queue to reach the k-th smallest fraction
        for (int i = 1; i < k; ++i) {
            Pair fraction = pq.top();
            pq.pop();
            if (fraction.first + 1 < fraction.second) {
                // If we can construct a new fraction with a bigger numerator
                pq.push({fraction.first + 1, fraction.second});
            }
        }

        // The top of the priority queue is now our k-th smallest fraction
        // Return the values from `arr` corresponding to the indices of this fraction.
        return {arr[pq.top().first], arr[pq.top().second]};
    }
};

```

Typescript Solution

```typescript
// Import array and priority queue utilities (the default JavaScript/TypeScript
// environment might not support priority queues, so assume a library like 'pq' exists)
import { PriorityQueue } from 'pq';

// Define a custom comparator for the priority queue that will compare fractions
const compareFractions = (a: [number, number], b: [number, number], arr: number[]): boolean => {
    return arr[a[0]] * arr[b[1]] > arr[a[1]] * arr[b[0]];
};

// Declare an alias for a pair of numbers for easier readability
type Pair = [number, number];

let priorityQueue: PriorityQueue<Pair>;

// Function to find the k-th smallest prime fraction
const kthSmallestPrimeFraction = (arr: number[], k: number): [number, number] => {
    // Initialize the priority queue with the custom comparator
    priorityQueue = new PriorityQueue<Pair>((a, b) => compareFractions(a, b, arr));

    // Initialize the priority queue with fractions [0, i] (where 0 < i)
    for (let i = 1; i < arr.length; i++) {
        priorityQueue.add([0, i]);
    }

    // Pop k-1 elements from the priority queue to reach the k-th smallest fraction
    for (let i = 1; i < k; i++) {
        const fraction = priorityQueue.peek();
        priorityQueue.remove();

        // If we can construct a new fraction with a larger numerator, add it to the queue
        if (fraction[0] + 1 < fraction[1]) {
            priorityQueue.add([fraction[0] + 1, fraction[1]]);
        }
    }

    // The top of the priority queue is now our k-th smallest fraction
    // Return the values from `arr` corresponding to the indices of this fraction
    const kthFraction = priorityQueue.peek();
    return [arr[kthFraction[0]], arr[kthFraction[1]]];
};

// Example usage:
// let arr = [1, 2, 3, 5];
// let k = 3;
// let result = kthSmallestPrimeFraction(arr, k);
// console.log(result); // Should output the k-th smallest prime fraction

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is governed by the following factors:

1. **Heap Construction**: The list comprehension creates a heap with an initial size of `n-1`, where `n` is the length of the input array `arr`. The `heapify` function has a time complexity of `O(n)`.

2. **Heap Operations**: The main loop runs `(k - 1)` times because it pops the smallest element from the heap and potentially pushes a new element onto the heap. Each `heappop` and `heappush` operation has a time complexity of `O(log n)`.

Thus, the total time complexity is given by the initial heapification, `O(n)`, plus the `k` iterations of heap operations, each of which is `O(log n)`, resulting in `O(n + klog n)`.

Space Complexity

The space complexity of the given code depends on:

1. **Heap Space**: The heap size is at most `n-1`, where `n` is the length of the input array `arr`.
2. **No Additional Space**: No additional space other than the heap is used that grows with input size.

Hence, the space complexity is `O(n)` since that is the space used by the heap.