1099. Two Sum Less Than K

Easy Array Two Pointers Binary Search Sorting

Problem Description

Given an array `nums` of integers and an integer `k`, the challenge is to find the maximum possible sum of any two distinct elements (i.e., `nums[i]` and `nums[j]` where `i < j`) that is less than `k`. To clarify:

- The sum should be less than `k`.
- You need to find two different elements in the array (no element should be used twice).
- The maximum sum needs to be returned.
- If it is not possible to find such a pair, the function should return `-1`.

Intuition

The intuition behind the solution involves sorting the array first. When the array is sorted, we can use a two-pointer approach to efficiently find two numbers with the desired property. The first pointer `i` starts at the beginning of the list, and the second pointer `j` starts at the end. The sorting allows us to make decisions based on the sum of the elements at the two pointers:

- If the sum of the two pointed-to numbers is less than `k`, then we have a valid pair, and we can try to find a bigger sum by moving pointer `i` one step to the right (increasing the sum).
- If the sum is greater or equal to `k`, then we must reduce the sum by moving pointer `j` one step to the left.

Throughout the process, we maintain the highest sum obtained that is less than `k`, and this becomes our answer. The approach works because sorting the array allows us to know that, as `i` moves to the right, the sum will increase, and as `j` moves to the left, the sum will decrease. This helps us avoid considering every possible pair, which reduces the time complexity significantly compared to a brute-force approach.

Solution Approach

The solution makes effective use of a two-pointer technique with a sorted array to optimize the process of finding the maximum sum of a pair of integers that is less than `k`. Here's a step-by-step breakdown of the implementation:

1. **Sort the Array**: The first step is to sort the `nums` array which will arrange the integers in non-decreasing order. Sorting is crucial because it allows us to explore potential sums from the smallest and largest values and move our pointers based on the comparison with `k`.
2. **Initialize Two Pointers**: We set up two pointers, `i` and `j`, where `i` starts from the beginning of the array (`0` index) and `j` starts from the end (`len(nums) - 1`). These pointers will traverse the array from opposite ends towards each other.
3. **Initialize the Answer Variable**: An `ans` variable is initialized with a value of `-1`. This variable will hold the maximum sum encountered that is still less than `k`.
4. **Iterate with Two Pointers**: We then enter a while loop that will continue as long as `i` is less than `j`. Inside the loop, we do the following:
   - Calculate the sum (`s`) of the elements pointed to by `i` and `j`.
   - If this sum is less than `k`, compare it with the current `ans` variable and update `ans` to be the maximum of the two. Since we want to maximize our sum, we then move `i` one step to the right (i.e., `i += 1`) to potentially increase our sum.
   - If the sum is greater than or equal to `k`, we must reduce it. To do this, we move `j` one step to the left (i.e., `j -= 1`), as this will use a smaller number for the sum.
5. **Return the Answer**: After the loop terminates, the variable `ans` will contain the maximum sum less than `k` that we have found, or `-1` if no such sum exists. We return `ans`.

By following these steps, the solution ensures that every move of either `i` or `j` is purposeful and brings us closer to the maximum sum we can achieve under the constraint that it must be less than `k`. Through this optimization, the solution effectively manages the potential combinations of pairs, resulting in a more efficient algorithm.

Example Walkthrough

Suppose we have an array `nums = [34, 23, 1, 24, 75, 33, 54, 8]` and an integer `k = 60`.

According to our solution approach:

1. **Sort the Array**: First, we sort the array: `nums = [1, 8, 23, 24, 33, 34, 54, 75]`.
2. **Initialize Two Pointers**: We set two pointers, `i = 0` (pointing to `1`) and `j = 7` (pointing to `75`).
3. **Initialize the Answer Variable**: We initialize an answer variable with `ans = -1`.
4. **Iterate with Two Pointers**: We enter the while loop with the condition `i < j`.
   - During the first iteration, the sum `s = nums[i] + nums[j] = 1 + 75 = 76` which is greater than `k`. Hence, we decrease `j` to point to the second last element (`j = 6` pointing to `54`).
   - In the next iteration, the sum `s = 1 + 54 = 55`. This sum is less than `k`, so we update `ans = 55`. We want to see if we can get closer to `k`, so we increase `i` to point to the next element (`i = 1` pointing to `8`).
   - Now, the sum `s = 8 + 54 = 62`, which is greater than `k`. So we decrease `j` to point to `j = 5` (pointing to `34`).
   - The sum `s = 8 + 34 = 42`. This is less than `k` and greater than the current `ans`, so we update `ans = 42`. We try increasing `i` again (`i = 2` pointing to `23`).
   - The sum `s = 23 + 34 = 57`. It is less than `k` and greater than `ans`, so we update `ans = 57`. Since we are still trying to maximize the sum, we increment `i` again (`i = 3`, pointing to `24`).
   - Continuing the iteration, we keep moving `i` and `j` to find larger sums until `i` is no longer less than `j`.
5. **Return the Answer**: Once `i >= j`, the loop terminates. Our `ans` is the largest sum found which is less than `k`, which in this case is `57`. Hence, we would return `57`.

Solution Implementation

Python

```python
class Solution:
    def twoSumLessThanK(self, nums: List[int], k: int) -> int:
        # Sort the list to apply the two-pointer technique
        nums.sort()

        # Initialize two pointers, one at the start and one at the end of the list
        left_pointer, right_pointer = 0, len(nums) - 1

        # Initialize the answer variable with -1 which will be returned if no suitable pair is found
        max_sum = -1

        # Iterate until the two pointers meet
        while left_pointer < right_pointer:
            # Calculate the current sum of the values pointed by the two pointers
            current_sum = nums[left_pointer] + nums[right_pointer]

            # If the sum is less than k, it's a potential candidate for our answer
            if current_sum < k:
                # Update the answer with the maximum sum encountered which is less than k
                max_sum = max(max_sum, current_sum)
                # Move the left pointer to the right to potentially increase the sum
                left_pointer += 1
            else:
                # If the sum is greater than or equal to k, move the right pointer to the left
                # This is to try and find a smaller sum that is less than k
                right_pointer -= 1

        # Return the maximum sum that we have found which is less than k, or -1 if there is no such sum
        return max_sum
```

Java

```java
class Solution {

    /**
     * Finds the maximum sum of any pair of numbers in the array that is less than K
     *
     * @param numbers Array of integers
     * @param targetSum Target sum K we are trying not to exceed
     * @return Maximum sum of a pair of numbers less than K, or -1 if such a pair does not exist
     */
    public int twoSumLessThanK(int[] numbers, int targetSum) {
        // Sort the array in ascending order
        Arrays.sort(numbers);

        // Initialize the answer as -1 assuming there might be no valid pairs
        int maximumSum = -1;

        // Initialize two pointers, one at the beginning (left) and one at the end (right) of the array
        int left = 0, right = numbers.length - 1;

        // Iterate over the array using the two-pointer approach
        while (left < right) {
            // Calculate the sum of values at the left and right pointers
            int sum = numbers[left] + numbers[right];

            // If the sum is less than the targetSum, update maximumSum and move the left pointer forward
            if (sum < targetSum) {
                maximumSum = Math.max(maximumSum, sum);
                left++;
            } else {
                // If the sum is greater than or equal to targetSum, move the right pointer backward
                right--;
            }
        }

        // Return the maximum sum found, or -1 if no suitable pair was found
        return maximumSum;
    }
}
```

C++

```cpp
class Solution {
public:
    int twoSumLessThanK(vector<int>& nums, int k) {
        // Sort the array to use the two-pointer technique
        sort(nums.begin(), nums.end());

        // Initialize answer to -1 since we need a sum less than k
        int maxSum = -1;

        // Initialize two pointers, one at the beginning and one at the end of the sorted array
        int left = 0;
        int right = nums.size() - 1;

        // Use the two-pointer technique to find the two numbers with a sum less than k
        while (left < right) {
            // Calculate the sum of the two pointed elements
            int sum = nums[left] + nums[right];

            // If the current sum is less than k, it's a potential answer
            if (sum < k) {
                // Update maxSum to the maximum of itself and the current sum
                maxSum = max(maxSum, sum);

                // Move the left pointer to the right to increase the sum
                ++left;
            } else {
                // If the sum is greater than or equal to k, decrease the sum by moving the right pointer to the left
                --right;
            }
        }

        // Return the maximum sum found that is less than k, or -1 if no such sum exists
        return maxSum;
    }
};
```

TypeScript

```typescript
function twoSumLessThanK(nums: number[], k: number): number {
    // Sort the array in ascending order
    nums.sort((a, b) => a - b);

    // Initialize the variable to store the maximum sum found that is less than k
    let maxSum = -1;

    // Initialize two pointers, one starting from the beginning of the array (left) and
    // the other from the end of the array (right)
    let left = 0;
    let right = nums.length - 1;

    // Iterate through the array while the left pointer is less than the right pointer
    while (left < right) {
        // Calculate the sum of the elements pointed by the left and right pointers
        const sum = nums[left] + nums[right];

        // If the sum is less than k, check if it is greater than the current maxSum
        // and update maxSum if necessary. Then move the left pointer one step to the right.
        if (sum < k) {
            maxSum = Math.max(maxSum, sum);
            ++left;
        } else {
            // If the sum is greater than or equal to k, move the right pointer one step to the left
            --right;
        }
    }

    // Return the maximum sum found or -1 if no such sum exists
    return maxSum;
}
```

```python
class Solution:
    def twoSumLessThanK(self, nums: List[int], k: int) -> int:
        # Sort the list to apply the two-pointer technique
        nums.sort()

        # Initialize two pointers, one at the start and one at the end of the list
        left_pointer, right_pointer = 0, len(nums) - 1

        # Initialize the answer variable with -1 which will be returned if no suitable pair is found
        max_sum = -1

        # Iterate until the two pointers meet
        while left_pointer < right_pointer:
            # Calculate the current sum of the values pointed by the two pointers
            current_sum = nums[left_pointer] + nums[right_pointer]

            # If the sum is less than k, it's a potential candidate for our answer
            if current_sum < k:
                # Update the answer with the maximum sum encountered which is less than k
                max_sum = max(max_sum, current_sum)
                # Move the left pointer to the right to potentially increase the sum
                left_pointer += 1
            else:
                # If the sum is greater than or equal to k, move the right pointer to the left
                # This is to try and find a smaller sum that is less than k
                right_pointer -= 1

        # Return the maximum sum that we have found which is less than k, or -1 if there is no such sum
        return max_sum
```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is `O(N log N)` because of the sort operation. The subsequent while loop has at most `N` iterations (where `N` is the length of `nums`), since each iteration either increments `i` or decrements `j`, ensuring that each element is considered at most once. However, the sort operation dominates the time complexity.

Therefore, the overall time complexity is associated with the sorting step, which for most sorting algorithms like Timsort (used in Python's `.sort()` method), is `O(N log N)`.

Space Complexity

The space complexity of the function is `O(1)` or constant space complexity, assuming that the sort operation is done in-place. Aside from sorting, only a fixed number of integer variables are introduced (`i`, `j`, `s`, and `ans`), which do not depend on the size of the input array `nums`.

If the sorting algorithm used is not in-place, the space complexity would be `O(N)` due to the space required to create a separate sorted array. However, since Python's default `.sort()` method sorts the array in place, we consider the space complexity to be `O(1)`.