Documentation

The problem of counting range sums within a specified range [lower, upper] requires us to determine how many subarrays have sums that lie within the given bounds. The range sum S(i,j) is the sum of elements between indices i and j (inclusive). Given constraints such as $1 \leq$ nums.length $\leq 10^5$, a brute-force approach of iterating through all possible subarrays would result in $O(n^2)$ time complexity, which is inefficient for large inputs. Therefore, we need a more optimized solution to handle large datasets effectively. By leveraging **prefix sums** and a modified **merge sort** approach, we can achieve a time complexity of O(nlogn), ensuring scalability and efficiency.

To reformulate the problem, we use the concept of prefix sums. A prefix sum array allows us to express the range sum S(i, j) as the difference between two prefix sums: S(i,j)=prefixSum[j+1]−prefixSum[i]. This reduces the problem to counting the number of prefix sum pairs where the difference lies within [lower, upper]. Specifically, for a given prefix sum at index j+1j+1, we need to count how many previous prefix sums satisfy the condition prefixSum[j+1]−upper≤prefixSum[i]≤prefixSum[j+1]−lower. This observation transforms the problem into one of efficient range counting for prefix sums, which can be handled using a divide-and-conquer approach.

The modified merge sort algorithm forms the core of the solution. During the merge step, the algorithm counts the number of valid prefix sum differences while maintaining sorted order. By dividing the prefix sum array into two halves, sorting them, and merging them back together, we can efficiently count the valid ranges using two pointers. The two-pointer technique is beneficial during the merging process, as it allows us to identify how many prefix sums in the right half satisfy the range condition relative to each prefix sum in the left half. The counting operation runs in linear O(n) time for each merge step, and since merge sort operates in O(logn) levels, the overall complexity becomes O(n logn).

The approach offers significant advantages over naive methods. First, it operates efficiently on large inputs, thanks to the O(n logn)time complexity. Second, the use of prefix sums eliminates the need for repeatedly recalculating subarray sums, thereby optimizing the solution further. Third, the merge sort's divide-and-conquer structure ensures correctness and enables efficient range comparisons during the merging process. As a result, the solution is both scalable and robust, making it suitable for handling arrays with up to 10510^5 elements as required by the problem constraints.

This problem has several practical applications across different domains. In financial analysis, for instance, it can be used to identify periods where cumulative stock returns or price changes fall within specific bounds. In data analytics, the algorithm can detect patterns or anomalies by analyzing cumulative metrics over time. Similarly, in simulations or gaming scenarios, it can identify subranges of scores or other cumulative values that meet predefined conditions. The ability to efficiently count valid subranges makes this approach highly applicable to real-world problems where range-based analysis is necessary.

The algorithm also handles edge cases effectively. For single-element arrays, the solution verifies whether the lone prefix sum satisfies the condition. For arrays containing both negative and positive values, the prefix sum approach works uniformly without requiring special handling. Additionally, the constraints of the problem guarantee that the final result fits within a 32-bit integer, ensuring correctness across all input cases. The careful use of prefix sums, combined with the divide-and-conquer methodology, allows the algorithm to handle extreme cases seamlessly while maintaining high performance.

In conclusion, the problem of counting range sums highlights the importance of optimizing range-based computations through techniques like prefix sums and merge sort. By transforming the problem into one of efficient range counting and leveraging the power of divide-and-conquer, we achieve a solution that is both elegant and efficient. This method not only satisfies the constraints of the problem but also offers a scalable approach for analyzing large datasets, making it a valuable tool for solving similar problems in various applications.