APPROACH

Intuition

The main idea behind this approach is to use two pointers to form pairs of indices (left, right) such that the sum of the minimum and maximum element in the subsequence formed by using the elements from nums[left] to nums[right] is less than or equal to the given target. To count all the subsequences that can be formed using the elements from nums[left+1] to nums[right], we can use the formula 2^(right-left), because for each element in this range, we can either choose to include it or exclude it from the subsequence, thus giving us 2 choices, and we have (right-left) such elements, hence the total number of subsequences that can be formed is 2^(right-left).

Approach

- Sort the given array in non-decreasing order.
- Traverse through the array using two pointers, one pointing to the left end and the other pointing to the right end.
- For each pair of indices (left, right), if nums[left] + nums[right] <= target, then count all the subsequences of nums that can be formed using the elements from nums[left+1] to nums[right], and add it to the final result.
- To count all the subsequences that can be formed using the elements from nums[left+1] to nums[right], we can use the formula 2^(right-left), because for each element in this range, we can either choose to include it or exclude it from the subsequence, thus giving us 2 choices, and we have (right-left) such elements, hence the total number of subsequences that can be formed is 2^(right-left).
- Return the final result.

Complexity

- Time complexity: O(nlogn), where n is the length of the given array. This is because we first sort the given array in non-decreasing order, which takes O(nlogn) time, and then we traverse through the array using two pointers, which takes O(n) time in the worst case.
- Space complexity: O(1), since we only use a constant amount of extra space to store the two pointers and the final result.