GA1 Report

Lianghui Wang, Chih Hsuan Huang

Part (A):

Description:

1. Sorting: Sort array A and array B (takes O(m log m + n log n) time.)
2. Using Two Pointers Technique: We use two pointers, one (i) starting at the beginning of array A and the other (j) starting at the end of array B. So at first A[i] is the smallest element in A and B[j] is the biggest element in B. This way, we can find pairs that sum up to a value within the target range by moving the two pointers.

Algorithm:

1. Sort array A and array B.
2. Initialize: count for the number of valid pairs, initialize to 0. i for indexing array A starting from 0, initialize to 0. j for indexing array B starting from n-1,initialize to n-1.
3. While i < m and j >= 0:

        Calculate sum = A[i] + B[j].

        If sum is within the range [t_min, t_max]:

            increment count: count += 1

            move i to the next element: i += 1

        If sum is less than t_min:

            increment i to increase the sum: i += 1

        If sum is more than t_max:

            decrement j to decrease the sum: j -= 1

    Return count.

Running time analysis:

This algorithm has a time complexity of O(mlogm+nlogn) because sorting is the most expensive operation.

Part (B):

Description:

1. $O(n^3)$ approach: Check the sum of every possible subarray. Use two points, one (left) starting at the beginning of array A and the other (right) starting at the left pointer, check sum between A[left] and A[right], if the sum is in the range of [t_min, t_max], count increase 1.
2. O(n·polylog(n)) approach: Use prefix sum and binary search idea to decrease the running time:
    a. We create a list that contains all prefix sum (pre_sum) and use the 'Sortlist' function to sort them. By using prefix sum, we can easily find the sum of each sublist: sum from A[i] to A[j] is equal to pre_sum[j] - pre_sum[i].
    b. We need to find the subarray sum in the range of [t_min, t_max], which means pre_sum[j] - pre_sum[i] is in the range of [t_min, t_max]. That is: pre_sum[j] - t_min > pre_sum[i] and pre_sum[j] - t_max < pre_sum[i]
    c. When we add a new pre_sum to the sorted prefix sum list, we can use bisect_right(pre_sum - t_min) to find how many prefix sum are less or equal than (pre_sum - t_min), and use bisect_left(pre_sum - t_max) to find how many prefix sum are less than (pre_sum - t_max). The difference between these two numbers tells us how many sum of the subarrays in the range of [t_min, t_max].

Algorithm:

1. Initialize: Initialize prefix sum: pre_sum = 0, count for valid subarray: valid_subarr_count = 0, sored_sums = Sortlist[0]
2. For i in A, do:

    pre_sum += i

    valid_subarr_count += sorted_sums.bisect_right(pre_sum - t_min) - sorted_sums.bisect_left(pre_sum - t_max) sorted_sums.add(pre_sum)

3. Return valid_subarr_count

Running time analysis:

1. For n elements in array A, the total time complexity of calculating and updating the pre_sums is O(n)
2. For n elements in array A, using Sortlist, bisect_right and bisect_left function: O(nlogn)
3. Insert every pre_sum to sorted_sums: O(nlogn)
4. So the total time complexity is O(n)+O(nlogn)+O(nlogn)=O(nlogn)