CS 325: Analysis of Algorithms
Group Assignment 1

Hyun Taek Oh
Busra Dedeoglu
Yu-Hao Shih

February 1, 2024

1 Report Of Problem A

There are two arrays A[1,...,m] and B[1,...,n], which include integers between t_{min} and t_{max}. We wrote a code, which finds when a∈A and b∈B and a+b in [t_{min}, t_{max}], it finds valid pairs' count.

Step 1: The code reads inputs, which are given in input.txt. We have vecA and vecB, which represent array A and Array B. We used merge_sort function, which sorts an input array in non-decreasing order. Merge sort function sorts a vector in $O(n \log n)$ time complexity.

Step 2: To find the number of valid pairs, we needed to find number of pairs, T^{-}_{max} whose sum is less than t_{max}, which contains pairs that also sums to less than t_{min}, T^{-}_{min}. Therefore, we also find the number of pairs that sum to less than t_{min} which are out of the desired interval $[t_{min}, t_{max}]$. The difference $T^{-}_{max} - T^{-}_{min}$ results with the number of pairs that sum to a value in $[t_{min}, t_{max}]$. For a given element $a \in A$, it only takes $O(\log n)$ complexity to find the maximum element $b \in B$ such that $a + b \leq t_{max}$ or $a + b < t_{min}$ using binary search algorithm. Binary search algorithm uses division and creates a tree for search with $\log n$ stages. It first finds the mid index in A and checks if this value is less/higher than t_{max}. If it is higher, then it only searches upper portion of A recursively using binary search again.

Step 3: The get_sum_of_pairs function goes through each number in the first array vecA. For each of these numbers, it modifies the range based on that specific number. Then, it counts how many pairs can be formed by combining this number with elements from the second array vecB within the adjusted range. This counting is done using binary search functions. Therefore, the total complexity becomes $O(n \log m)$.

Step 4: The print_to_output function saves the final result to output.txt file. It takes the calculated value and writes it into the output.txt file. The code uses specific instructions given when starting the program from the command line. It takes the names of input and output files as inputs. Then, it reads the numbers from the input file, figures out how many pairs meet certain conditions, shows that number on the screen, and saves it in the output file.

2 Calculating Running Time Of Problem A

The code uses a method called merge sort to arrange numbers in a certain order. This method takes a reasonable amount of time, specifically $O(n \cdot \log n)$, where n is the number of elements. Mergesort uses a divide-and-conquer methodology, where it recursively splits input array into 2 pieces and applies mergesort again to each parts. This results with at most *logn* many stages in the recursion tree. At every stage it takes n operations to merge two split parts in sorted order by using 2 pointers, resulting with a total complexity of $O(n \log n)$.

In addition to this, another for loop is used to span vector A. For every element $a \in A$, we need to search for $b \in B$ that sum to values less than t_{min} and t_{max}. Binary search algorithm uses division and creates a tree for search with $\log n$ stages (similar to splits from half as in merge sort). It first finds the mid index in A and checks if this value is less/higher than t_{max}. If it is higher, then it only searches upper portion of A recursively using binary search again, resulting with complexity $O(\log m)$

Since these two sections are run one-after-another, the total complexity is always less than $O((n + m)\log(n + m))$ depending on the values of n, m.

3 Proof of Correctness Of Problem A

To prove that the proposed algorithm solves the problem, we need to show that mergesort always successfully sorts vectors and binary search is guaranteed to find the max element in B less than t_{min}, t_{max}. For any given vector mergesort will process every value in an vector during divide stage. During merge stage, it can be shown that a vector X and a vector Y will be merges in a sorted manner, i.e., there will be 2 pointers and the smallest two elements $x \in X, y \in Y$ at any given time will be compared. Therefore, there cannot be any $x \in X, y \in Y$ such that $x < y$ and sorted$= [y, x]$, because otherwise x must be compared to an $y^* < y$ and $x > y^*$, which is a contradiction. Therefore, at every merge, the output of the recursion must be sorted. Since same happens with every recursion, mergesort is guaranteed to provide a sorted output.

Similarly, binary search will search for a value z by dividing the input vector into 2 halves. Then one half H will be selected if $z \in H^1$. Since this will be done recursively, $z \in H^k, \forall k = 1, ..., \log n$. Therefore, binary search is also guaranteed to find z. Since binary search will work for any selected $a \in A$ and hence z, the proposed algorithm finds all possible number of valid pairs.

4 Report Of Problem B

In this problem, we are given a single array $A = [a_1, a_2, ..., a_n]$ that may contain negative or positive integers. We are tasked to find the number of contiguous subarrays whose elements add up to a value between $[t_{min}, t_{max}]$. The algorithm implemented to solve this problem leverages the contiguous property. The brute-force approach, in which we check all 1-sized, 2-sized, ..., k-sized subsets will result with a $O(n^3)$ complexity. The reason is that we repeat the summation operations over and over again.

A smarter approach is to compute the subarray summations in the following order for a given array $A = [a, b, c, d]$:

$$\{a\}, \{a, b\}, \{a, b, c\}, \{a, b, c, d\} \tag{1}$$
$$\{b\}, \{b, c\}, \{b, c, d\} \tag{2}$$
$$\{c\}, \{c, d\} \tag{3}$$
$$\{d\} \tag{4}$$

We here notice that we only need to perform 1 additional summation operation whenever we move to the subsets on the right. For an input array of size 4, the number of all possible contiguous subsets is $\frac{n(n+1)}{2} = \frac{4*5}{2} = 10$, which results with $O(n^2)$ complexity.

To further improve on this, we need to leverage divide-and-conquer methodology. Therefore, we implemented the following algorithm. We start with an index variable i set to 0 and a current summation variable *sum* set to 0. At each iteration, we add the ith value from A to *sum*. Notice that at iteration 0, *sum* stores the value $sum = a$, at iteration 1, *sum* stores the value $sum = a + b$ and at iteration 2, *sum* stores the value $sum = a + b + c$ and so on. While i increases, at every iteration we also store the sum value in a sorted list *prevsum*. Therefore, at the end of iteration *3*, *prevsum* will contain the sum values of $\{a\}, \{a, b\}, \{a, b, c\}$. At every iteration, we also compute a gap variable *gap* that is $gap = sum - t$, where t could be t_{min} or t_{max}. For instance, at iteration 4, we check if $sum = a+b+c+d \leq t_{max}$ first. If this is true, we increase the number of subsets whose elements sum up to a value less than t_{max}. Similarly, we check if $sum = a + b + c + d < t_{min}$. If this is true, we increase the number of subsets whose elements sum up to a value less than t_{min}. Please note that the final count will be number of subsets add up to a value less than t_{max} minus number of subsets add up to a value less than t_{min}. Then, we also compute $gap_{min} = sum - t_{min}$.

Since there might be other subsets that satisfy $sum - \sum A_k \in prevsum < t_{min}$, we will search for all subset sums $sum - t_{min} = gap_{min} < \sum A_k \in prevsum$. It is always true that $a, a+b, a+b+c \in prevsum$ at iteration 4. However, the order in prevsum might be different since prevsum is a sorted list. Let's assume $a < gap_{min} = sum - t_{min} < a+b+c < a+b$. Then, it is sufficient to find the index of $a+b+c$, i_{abc} and deduce the length of the sorted list slice $[i_{abc}:]$ without a need to check other conditions. Then we will increase the valid subset count by 2 (in this specific example). It is important here to note that this operation corresponds to checking for subsets $\{b,c,d\}, \{c,d\}, \{d\}$. In this particular example, $\{c,d\}, \{d\}$ resulted with sum values less than t_{min}. We do the same procedure for t_{max} including equality. Please note here that finding the index of value satisfying $a < gap_{min} = sum - t_{min} < a+b+c < a+b$ is $O(\log n)$ since binary search leverages divide-and-conquer. Since we only perform a single loop from 0 to n and binary search is $O(\log n)$, the final algorithm complexity is $O(n \log n)$.

5 Calculating Running Time Of Problem B

Proposed algorithm relies on 3 sub-modules. We perform a single loop over all values of $A = [a_1, ..., a_n]$. Inside the loop there 3 operations: 1. a single addition: at every iteration we find $\sum_{i=0}^{k-1} a_i + a_k$, where $\sum_{i=0}^{k-1} a_i$ is computed in previous iterations; 2. binary search in a sorted list; 3. insertion into a sorted list. The number 1 is an $O(1)$ operation, number 2 is an $O(\log n)$ operations, and number 3 is also an $O(\log n)$ operation. To achieve $O(\log n)$ complexity, a sorted list library (that is corresponding to standard array in C++) is used, which utilizes a binary tree. Since binary tree also leverages divide-and-conquer approach (its height is approximately $\log n$), insertion to a binary tree has complexity $O(\log n)$. Then the overall complexity of the algorithm becomes $O(n \log n)$ since insertion and binary search are sequential operations. The final subtraction of number of subarrays whose elements add up to a value less than t_{max} or t_{min} is $O(1)$ and does not impact the overall complexity.

6 Proof of Correctness Of Problem B

Proposed algorithm needs to check every possible contiguous subarrays to ensure the correctness of the results. An array of size n can have at most $\frac{n \times (n+1)}{2}$ contiguous subarrays. The loop in the proposed algorithm alongside summation checks exactly n contiguous subarrays. The sum of elements for each of these subarrays are stored in *prevsum* sorted list that is an n-length list, which grows at each iteration. At iteration k, *prevsum* contains the summations corresponding to the following subarrays: $\{a_1\}, \{a_1, a_2\}, ..., \{a_1, ..., a_{k-1}\}$. At the same time the current sum corresponds to the subarray $\{a_1, ..., a_k\}$. Now we can show that these subarrays are enough to check all contiguous subarrays of $A_k = \{a_1, ..., a_k\}$:

$$\text{All Right Contiguous Subarrays of } A_k = \text{All Right Contiguous Subarrays of } A_{k-1} \bigcup (A_k \setminus \{a_1\}) \cup (A_k \setminus \{a_1, a_2\}) \cup ... \cup (A_k \setminus \{a_1, ..., a_{k-1}\}) \quad (5)$$

which contains $k-1$ additional subarrays. If we can show that for $k = 2$, then by induction we can conclude the proof.

$$\text{All Right Contiguous Subarrays of } A_2 = \text{All Right Contiguous Subarrays of } A_1 \bigcup (A_2 \setminus \{a_1\}) \cup (A_2 \setminus \{a_1, a_2\}) \quad (6)$$

$$= \{\} \bigcup \{a_2\} \quad (7)$$

and it has 1 right contiguous subarray for $k = 2$, which confirms at every iteration we have $k-1$ right contiguous subarrays. Then, we have $k - 1 + 1 = k$ contiguous subarrays being checked at every iteration by running the proposed algorithm. Then, $\sum_{k=1}^{k=n} k = \frac{n \times (n+1)}{2}$ contiguous subarrays are checked, which completes the proof.