Week 1 CSES

First problem is trivial. The missing number is the difference between the AP sum and the sum of all presented numbers.

The second problem is more interesting. We could naively, for each integer in the list, sweep the list for a matching pair. This is of course too slow.

A suitable way to do this is to use an increasing and a decreasing pointer on the sorted list, we could decrease the right pointer and increase the left pointer independently: that is increase the left pointer until we find a sum or go over it. If latter then we decrease the right pointer, then decrease the left pointer until we either find one or go under the sum, then we decrease the right pointer again and starts increasing the left pointer...

The above method would still be, in amortized time be O(n log n), bounded by the sort time. The active pointer in the above could also be replaced with a binary search pointer. This does not affect the overall complexity.

Another suitable way to do this is to map the existence of a number, and new number `a` will check for an instance of `x-a` in the map. If it exists then return their indices, if not continue until either we exhaust the list or find something. `<map>` in c++ is implemented as a BST, therefore O(n log n) total time to insert and search for all n instances.

An interesting problem I ran into is when I tried to use `<unordered_set>` to find such an instance, which is implemented with a hash table rather than a BST. This reduces the insertion and search time for all instances to be O(n) expected. However (as posted on the forum), two of the tests were meticulously engineered to break the hash function to maximum hash collisions, slowing the algorithm to O(nk) where `k < n` being the number of collisions presented for each insertion. Although I could add a filter specifically for the cases that have such collisions (since they all yield negative value on `x - a`), but this raises a solid point on not to use `<unordered>` structs on large cases.