Problem 1. You’re an announcer at the hottest rock-paper-scissors competition in North America. The biggest match of the evening is coming up, Rocky Rick vs. Paper Pete, and you need make sure the audience stays engaged during the event. For past announcers, the biggest obstacle is keeping the audience engaged during the halftime show (the players need time to rest their hands).

In order to step things up, you’re planning to do something that has never been done before and want to be able to say that this is the \( n \)th time that the score between Rocky Rick and and Paper Pete has been \( i \) to \( j \) at halftime. The main challenge that has prevented previous announcers from doing this is that you won’t know what \( i \) and \( j \) are, as the game has not yet happened. You also cannot afford to scan through the entire list of previous halftime scores between Rick and Pete and count the number of \( i \) vs \( j \) appearances as the audience would surely revolt.

1. Devise an efficient method of processing the list of previous Rick-Pete halftime scores before their match begins, so that you can quickly say, right at the start of half-time, how many times the pair \((i, j)\) has occurred at similar moments in the past. Your pre-match processing should take time proportional to the number of previous games and the querying task should take constant time.

2. Justify the runtime and correctness of your scheme.

Solution. 1. We can maintain the frequency of each half time score \((i, j)\) into a hash map implementing chaining, using some hash function that depends on values \(i\) and \(j\). Assuming we have a hash function that disperses the data in the table with sufficient evenness, operations `insert` and `get` would have an expected time of \(O(1)\).

   i. When inserting a new score into the hash map, we can set its value to 1. When inserting a score that has already appeared into the hash map, we can increment its value.

   ii. When querying a score that is not in the hash map, we can return null or throw an error, indicating that this a score that has not been witnessed in previous games.

Note that this scheme is expected time \(O(1)\) because in a hash table, we have a worst case scenario where all numbers are mapped to a single slot in the map, in which case all elements would be in one chain. The run time of our algorithm would then no longer be constant.

2. i. By inserting the half time score of each previous game into the hash table, we are executing a constant operation with every previous game, meaning the processing phase takes time proportional to the number of previous games.

   ii. Because we assume we have an adequate hash function such that `get` is constant, querying should be a constant time operation.
Problem 2. You are given two integer lists sorted in ascending order representing the scores that Alice received on her tests and the scores that Bob received on his tests. You are also given an integer \( m \). We want to find the \( m \) pairs with the smallest sum of scores. A pair \((x, y)\) is defined as having one score from Alice and one score from Bob. For example, given the input Alice = \([1,7,11]\), Bob = \([2,4,6]\), \( m = 3 \), we want to return \([ (1,2), (1,4), (1,6) ]\).

1. Devise an efficient algorithm to find the \( m \) smallest pairs of scores and give the pseudocode.
2. Justify the runtime and correctness of your scheme.

Solution. 1. Pseudocode:

```
Algorithm 1 kSmallestPairs

1: procedure kSmallestPairs(nums1, nums2, k)
2:     if len(nums1) == 0 or len(nums2) == 0 then
3:         return empty list
4:     visited ← empty map
5:     pq ← empty min heap
6:     pairs ← empty list
7:     pq.push(nums1[0] + nums2[0], [0, 0])
8:     while pairs.size() < m and pq not empty do
9:         (val, [idx1, idx2]) ← pq.pop()
10:        pairs.append([(nums1[idx1], nums2[idx2])]
11:        if idx1 + 1 < nums1.size() and [idx1 + 1, idx2] not in visited then
12:            visited[[idx1 + 1, idx2]] ← true
13:            pq.push(nums1[idx1 + 1] + nums2[idx2], [idx1 + 1, idx2])
14:        if idx2 + 1 < nums2.size() and [idx1, idx2 + 1] not in visited then
15:            visited[[idx1, idx2 + 1]] ← true
16:            pq.push(nums1[idx1] + nums2[idx2 + 1], [idx1, idx2 + 1])
17:     return pairs
```

2. Runtime:

The time complexity is \( O(m \log m) \). The while loop runs \( m \) times. At each iteration, a constant number of heap additions and removals are made - since the heap has max size \( m \) (since at most two heappush, and one heappop for each iteration), these operations take \( O(\log m) \). Therefore, overall time complexity is \( O(m \log m) \).

Correctness:

(a) We always return the minimum pair from the heap
(b) The minimum pair not yet returned cannot be outside the heap, because of the way that we add elements to the heap. This can be showed by contradiction. (Sketch of contradiction: Assume \((i, j)\) are the indices of the minimum element, but it’s not in the heap. Let \(a, j\) (or \(i, b\)) be indices (where \(a/b\) is maximized) of a pair in the heap (so the corresponding pair
shares an element with \((i, j)\), the minimal element. Since \(a < i\), or \(b < j\) by construction, the pair \((a, j)\) or \((b, i)\) must be less than or equal to \((i, j)\). Therefore, there exists a minimal element in the heap - contradiction reached).