Top K Frequent Elements in a Data Stream

1. Multi hashmap + heap
   We will split the data into n shardings and move a record to an instance repectively. Instance0 only handles the data with its hash code equals hash(item) % n == 0, Instance1 only handles the data with hash(item) % n == 1. For each instance, we have a HashMap and Heap to store k most frequent most frequent items. Then we will use a centric server to gather all heaps from all instances and run a merge sort. Merge k sorted lists. Even though we distribute the work load to multiple machines to reduce the data size to 1/n, it’s still not the optimal way to handle large data set in daily work.

2. Count-Min Sketch + Heap
   There are many ways to perform approximate calculation, Count-Min Sketch is one of them. Assume that we have d hash functions, create a hash table T with d rows and m cols. For each item read from data stream, get its hash values from d hash functions and perform mod operations respectively by m. Increase the value by one for each position T[hash_func][hash_value], we call it sketch.
   When we want to query the frequency of a particular item, we get its d sketches, return the smallest sketch. As a matter of fact, each of sketch can be used as its approximate frequency, here we use the minimum sketch.

3. Lossy Counting
   Lossy Counting Algorithm is another approximate algorithm to identify elements in a data stream whose frequency count exceed a user-given threshold.
   Step1: Build a HashMap to store the mapping from element to its frequency.
   Step2: Build a data frame(window).
   Step3: Read Data from stream and put them in the data frame, get all their frequencies f and minus 1.
   Step4: Update the item frequencies to HashMap, remove the items whose frequency equals to 0 from the HashMap
   Step5: Repeat Step 3
   The basic idea is, it is less possible for a high frequent item to get removed from the map even though all of them have to be decreased by one for each round. As we read more data, low frequent items will be removed from HashMap and high frequent items stay.

35. Search Insert Position

We will use bisect_left to search elements less or equal to given target’s index. TC is O(logn)

```
from bisect import *
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        return bisect_left(nums, target)
```

47. Permutations II
We will take advantage of itertools to get all permutations and use set to get rid of duplication.
TC is O(n * n)

```
import itertools
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        return list(set(itertools.permutations(nums, r=len(nums))))
```