Daily Coding Problem #300

Problem

This problem was asked by Uber.

On election day, a voting machine writes data in the form `(voter_id, candidate_id)` to a text file. Write a program that reads this file as a stream and returns the top `3` candidates at any given time. If you find a voter voting more than once, report this as fraud.

Solution

To handle the requirements above, it will be helpful to create a class, `VoteReporter`, that has methods for reading the data from a stream, reporting frauds, and finding the top candidates.

When we read the data, we want to update two things: each voter's choice, and the running tally of each candidate's vote counts. Each of these can be represented as a dictionary. For the voters, each voter will be a key, whose value will be their chosen candidate. Whenever we come across a key that already exists in this dictionary, we can report a fraud, which for our purposes can mean appending that person to a list.

For the tally, each candidate will be a key, whose value will be the number of votes he or she has received so far. Finally, we can implement a `get_top_candidates` function that pushes all the items in this dictionary into a max heap, and pops the top three candidates.

```
import heapq

from collections import defaultdict

class VoteReporter:
    def __init__(self, data, k=3):
        self.data = data
        self.k = k
        self.voters = {}
        self.tally = defaultdict(int)
        self.frauds = []
        self.run()

    def run(self):

        self.read_data()
        self.get_top_candidates()

    def read_data(self):
        for voter, candidate in self.data:
            if voter not in self.voters:
                self.voters[voter] = candidate
                self.tally[candidate] += 1
            else:
                self.report_fraud(voter)

    def report_fraud(self, voter):
        self.frauds.append(voter)

    def get_top_candidates(self):
        heap = []
        for candidate, votes in self.tally.items():
            heapq.heappush(heap, (-votes, candidate))

        for i in range(1, self.k + 1):
            candidate = heapq.heappop(heap)[1]
            print("#{} candidate:".format(i), candidate)
```

For our complexity calculations, let V be the number of voters, and C be the number of candidates. Furthermore, let us assume that fraud cases are negligible overall. Then reading the file will take `O(V)` time, and building the heap of top candidates will take `O(C log C)`. If we suppose that the number of voters is far greater than the number of candidates, the overall time and space complexity will therefore be `O(V)`.

One problem with this implementation is that it prevents us from querying the top candidates while the stream of data is being read. To fix this, we can use Python's `threading` module.

The idea here is that there will be two independent threads: one that reads the stream of data, and one that polls to find the top candidates. We will initially set `get_top_candidates` to run repeatedly after some interval of time, giving us a real-time estimate on how the race is progressing. Once all the stream data has been read, we terminate this polling process.

```
import heapq
import threading
import time

from collections import defaultdict

class VoteReporter:
    def __init__(self, data, k=3, poll_interval=10):
        self.data = data
        self.k = k
        self.poll_interval = poll_interval
        self.voters = {}
        self.tally = defaultdict(int)
        self.frauds = []
        self.stream_done = False
        self.run()

    def run(self):
        t1 = threading.Thread(target=self.read_data)
        t2 = threading.Thread(target=self.get_top_candidates)

        for thread in (t1, t2):
            thread.start()

    def read_data(self):
        for voter, candidate in self.data:
            if voter not in self.voters:
                self.voters[voter] = candidate
                self.tally[candidate] += 1
            else:
                self.report_fraud(voter)

        self.stream_done = True

    def report_fraud(self, voter):
        self.frauds.append(voter)

    def get_top_candidates(self):
        while not self.stream_done:
            time.sleep(self.poll_interval)


            heap = []
            for candidate, votes in self.tally.items():
                heapq.heappush(heap, (-votes, candidate))

            for i in range(1, self.k + 1):
                candidate = heapq.heappop(heap)[1]
                print("#{} candidate:".format(i), candidate)
```

Complexity evaluation is independent of threading operations. That is, making a process run in parallel does not change the overall amount of work that must be done. However, in this implementation we have increased the potential number of polls that must be carried out, which may be significant. If each of `P` polls takes `O(C log C)` time to run, then we can consider the new time complexity to be `O(V + P * C log C)`.

Terms of Service Press