Write-up of Algorithm & Summary of Results

1. Start Zookeeper Server
   ```
   zookeeper-server-start /usr/local/etc/kafka/zookeeper.properties
   ```
2. Start Kafka Server:
   ```
   kafka-server-start /usr/local/etc/kafka/server.properties
   ```
3. Now open twitter_to_kafka.ipynb and run the cells one by one. This will fetch data from twitter and push it to kafka. The topic under which the tweets will be stored is called `got`.
4. Now to check if the data is landing in Kafka, use this command:
   ```
   kafka-console-consumer --bootstrap-server localhost:9092 --topic got --from-beginning
   ```
5. Start ElasticSearch Server
   ```
   elasticsearch
   ```
   Simultaneously, start Kibana:
   ```
   kibana
   ```
6. Now go to the logstash directory:
   ```
   cd /usr/local/Cellar/logstash/6.6.1
   ```
   First check if the configuration file is okay using this command:
   ```
   bin/logstash -f twitterstream.conf --config.test_and_exit
   ```
7. Use the following logstash command for transferring data from Kafka to ElasticSearch
8. `bin/logstash -f twitterstream.conf --config.reload.automatic`
9. This will start indexing the tweets into elastic search. Go to kibana on `https://localhost:5601` in a web browser and check that our data is present there. Go to dev tools and run a match all query. See that we've got our data.
   ```
   GET /got/_search { "query": { "match_all": {} } }
   ```
10. Run the DRPC_ElasticSearch.ipynb, execute each cell to see the results.

    We support five types of DRPC queries:

    i. Match Query

    ii. Match Query with AND operation

    iii. Aggregation Query

    iv. Term Query

    v. Range Query

11. Bloom Filter & Count Min Sketch

    a. Since our use case was about determining the top 5 trending Game of Thrones characters, we first defined this list of characters:
    ["ned", "eddard", "robert", "jamie", "catelyn", "cersei", "daenerys", "jorah", "viserys", "jon", "sansa", "arya", "theon", "bran", "joffrey", "tyrion", "baelish", "davos", "samwell", "stannis",

"melisandre", "jeor", "bronn", "varys", "tywin", "gendry", "brienne", "ramsay", "gilly", "daario", "missandei"]

b. These 31 characters were then used to generate a Bloom Filter and consequently a Count Min Sketch, designed in the following fashion:

Given that n = 31 and that we want an error rate of 1% i.e. 0.01, we use the two equations

$m = \frac{-n \ln ln\, p}{2)^2}$ and $k = \frac{m \ln ln\, 2}{n}$

We found m = 297.13 ~ 298 and we approximated it to 300

k = 6.666 ~ 7

So, we needed 7 hash functions and 300 bits for the bloom filter and another 31 rows for the CMS.

c. The hash functions we used were a linear combination of SHA1 and SHA224, a method advocated by [reference I can't find rn] that generated independent hash values.

12. Top K Heavy Hitters
    a. A python dictionary defined to store min counts of all characters maintains track of all its keys and presents a list of five most frequently occurring keys and their min frequency after each iteration.
13. Results
    a. As a result of this project, it was found that popularity of characters of the TV show does not vary much with time unless there has been a disturbance in its routine. Activity in the form of invocation of character names on Twitter was found to be more prominent during weekends of release of the TV show's episodes. At these times, not only was the fluctuation in characters' placement in the leaderboard more profound, but their total min counts were also found to be larger. This can be attributed to the release of new episodes coupled with more free time available on weekends.