02807: Challenge 2 - Reddit

Code with output can be found in this notebook (use this when reading report):
https://gist.github.com/anonymous/3ee11a4a3626a2ca629793dd57d37476
Scripts for the exercises can be found (if you want to test for yourself):
https://www.dropbox.com/sh/oy9prbtnqdaore1/AAAY9GNiOoqRyRKtqjyJq9RKa?dl=0

Problem 1 - Vocabulary:

For this problem, we need to figure out the size of the vocabulary for each subreddit. We have around 47000 subreddits, and there simply is not enough room in my RAM to store 47000 vocabularies at the same time.

The comment table of the SQL database is not ordered by subreddit_id, so in my query I have to order by it:

```
query = "SELECT subreddit_id, body FROM comments order by subreddit_id"
```

This query makes the operation of retrieving elements from the cursor-object very slow since it has to fetch its elements at random locations on the SSD and not in a sequential manner as it is, when we just select from the table without ordering.

Since correctness is judged upon the output of David's script, I decided to just use that as my tool to get the vocabulary of the subreddit even though there probably is a faster solution

The overall idea with the script is:

- Extract a list of id's for the subreddits and also a dictionary containing {"**subreddit_id**" : "**name**"}, so that we can later match the output of the algorithm with an actual name of the subreddit. This query is very fast.
- Next we want to figure out the size of each vocabulary so a dict is made, which will hold the size of the vocabulary for each subreddit {"**subreddit_id**" : "**len_voc**"}.
    - A query is made to get the subreddit_id and the body field for each comment, ordered by subreddit_id
    - We can now iterate over the cursor, the first *N1* elements will contain all the comments of **subreddit_1,** the next *N2* elements all the comments of **subreddit_2** and so on. So whenever the subreddit_id changes in the cursor, we know that we have seen all the comments of the previous subreddit and thus we can find the total length of the vocabulary, accumulated by Davids script.
    - Note that Davids script takes in a reference to the word-set, thus it doesn't have to return anything since it has access to its location in memory from within the function scope.
- After this process is done, we want to find the 10 largest values of our dictionary so we construct a list with the *sorted*-function that returns a sorted list of subreddits based on the size of the vocabularies
- Then it's just some printing and matching the subreddit_id with the name of the subreddit

Result:

Having done 3 trails throughout the project, my best time was 3420 seconds or 57 minutes. I forgot to take a screenshot of that so I can't document it, but the latest run that I just did can be seen in the notebook and the time together with the output of that run is:

```
Time: 79.88343771298727 min
Size:   id:       Name:
556635  t5_2qh1i  AskReddit
221424  t5_2qh33  funny
206683  t5_2qh0u  pics
193694  t5_2qh13  worldnews
189606  t5_2qqjc  todayilearned
188477  t5_30uy0  subredditreports
187362  t5_2qh1e  videos
162499  t5_2rfxx  leagueoflegends
153071  t5_2sgp1  pcmasterrace
138567  t5_2s7tt  AdviceAnimals
```

Problem 2 - Common Authors:

In this problem we should investigate, which pairs of subreddits that shares the most authors.

Since we have 47000 different subreddits we have an extreme amount of different 2-combinations between these and it would not be very efficient to go through them all since we just want to know top 10. It makes sense that there should be a correlation between the number of authors in a subreddit and the number of common authors, therefore I only want to investigate a subset of all possible pairs of subreddits, namely a subset of combinations between the most popular subreddits.

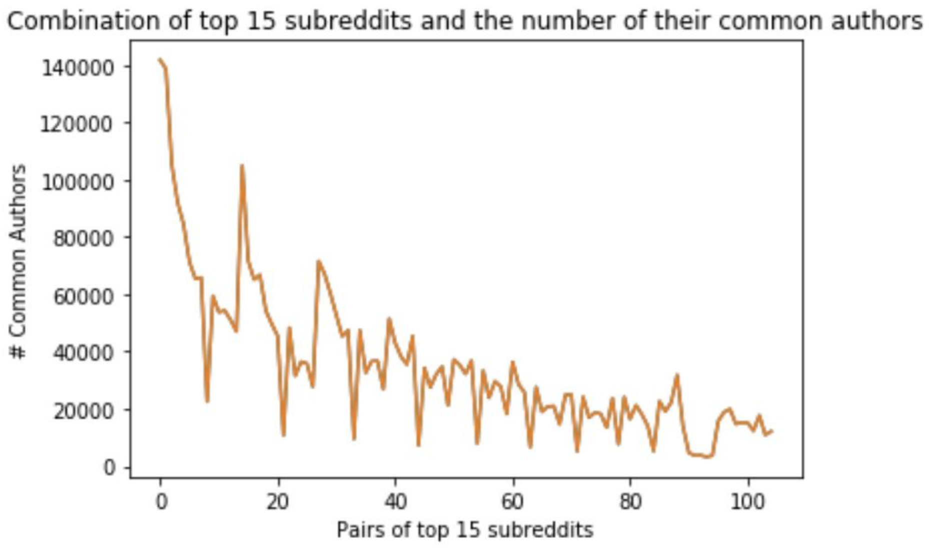
The overall idea for the script is:

- Find out which subreddits have the most authors:
    - Get the id and names of subreddit like in previous problem
    - Make a dict **sub_red** which is a dictionary of sets, where each key is a subreddit and the corresponding value is a set of authors for that subreddit.
      Since each comment is made in a subreddit, I just add the author of that comment to the set for the corresponding subreddit in the dictionary.
    - This can be sorted like before and we now have a list of the subreddits with most authors in descending order.
- Find all combinations of the 15 subreddits with most authors using itertools
- For each combination, do a intersection of the two sets of authors for the subreddits to find the set of authors they have in common.
- Take the length of this set to find the number of authors in common.
- Append it to our running list and find the indexes of the list for our top 10, print and done.

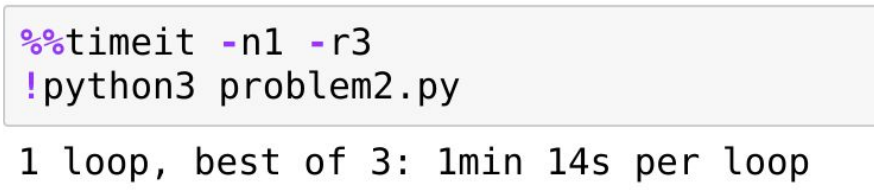
Results:

```
Final Time: 107.97922873497009 sec
Pairs    Frequency
141788   AskReddit-funny
138672   AskReddit-pics
105138   AskReddit-todayilearned
104903   funny-pics
91975   AskReddit-videos
84457   AskReddit-AdviceAnimals
71664   funny-todayilearned
71557   pics-todayilearned
71247   AskReddit-WTF
67090   pics-videos
```

If we take a look below we see why this method seems valid: The plot shows the number of common authors between all the combinations of the top 15 subreddits. Notice the periodicity of 15 for the graph and the general trend. If we were to include more subreddits (extend the graph), there is no reason to believe we should find a pair of subreddits that has a higher number of common authors than the top 10 for these combinations. I have tried to extend to 200+ most popular subreddits, and the top 10 combinations doesn't change.



Running the script directly in python, disabling output and closing chrome I get a best runtime of 74 seconds



```
%%timeit -n1 -r3
!python3 problem2.py
```

```
1 loop, best of 3: 1min 14s per loop
```

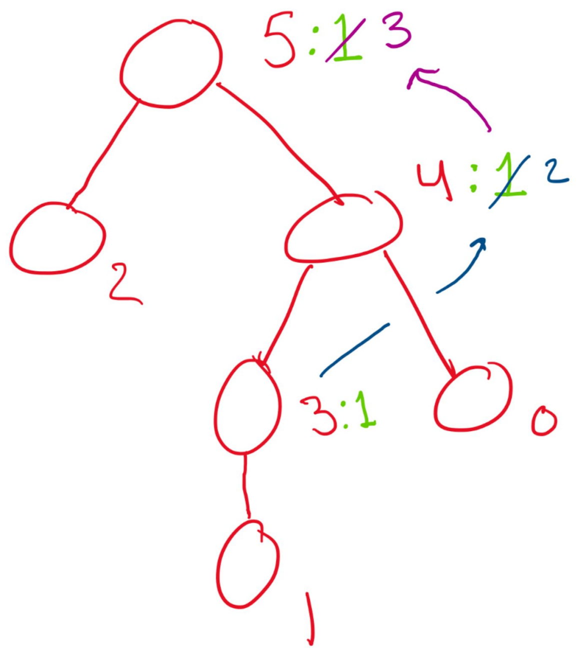
Problem 3 - Deepest average comment threads:
For this problem we should find the average depth from the top comments for all the subreddits.

This problem is a bit tricky, but something finally clicked:
First I realised (from the json structure), that the database is ordered in regards to time. I.e the first entry in the comments are the first comment, and the last entry in the last comment from a time perspective.

The goal is to create a tree structure for each "thread" and find the depth of the tree, but creating such a structure dynamically is hard if we just iterate over the DB since we have to keep track of everything. For instance, the query will give me a comment, and I know in which subreddit it belongs and also who its parent are, but when building up my tree I will have to search through the whole subreddit to find the parent before I know where to put it.

Instead I decided to combine the two tasks, by iterating BACKWARDS through time, i.e starting from the last row in the database. The idea is then to track back a leaf node to it's parents and keep doing this until we are at the top of the thread. Illustrated with an example:

Consider the below tree as a thread with nodes as comments. The red numbers beside the nodes indicate the time-step when looking from the back of the database. I.e 5 is the first observed comment, then 4, 3, 2, 1, 0. Note that a parent always comes before its children from a time perspective, makes good sense - you can't make a comment to a comment that hasn't been written yet. After the creation of the tree we see that it has a depth of 3.



My technique to solve the problem is then:

- When we observe a comment with **id, parent_id**, we know that the parent node will have at least one comment.
- Set the parent node to: **nodes[parent_id] = max(nodes[parent_id], nodes[id] + 1)**

I.e if the current comment comes from a longer chain than the one registered by the parent, update it else don't.

The code for this example:

```
dic = defaultdict(int)
indi = [(0,4), (1, 3), (2, 5), (3, 4), (4, 5)]

for idd, parent in indi:
    dic[parent] = max(dic[parent], dic[idd] + 1)

dic
```

```
defaultdict(int, {0: 0, 1: 0, 2: 0, 3: 1, 4: 2, 5: 3})
```

Note that we are actually not interested in all the intermediate outputs, so when a comment has been detected and its parent updated we can delete it

```
dic = defaultdict(int)
indi = [(0,4), (1, 3), (2, 5), (3, 4), (4, 5)]

for idd, parent in indi:
    dic[parent] = max(dic[parent], dic[idd] + 1)
    dic.pop(idd)
dic
```

```
defaultdict(int, {5: 3})
```

With that explanation out of the way, let's go on to outline the idea behind the script in the context of the theory above:

- Fetch the subreddit indices and the names as before
- Construct a dictionary containing a key for all the subreddits and dictionary as value. The default value for this dictionary is set to an integer, so if we look up an entry in the dictionary that doesn't exist, it will be created and receive a value of 0.
- Now we iterate backwards through the database using the ORDER BY syntax.
- For each comment we read, go into the subreddit dictionary and update its parent with the **max-**operation as described above. Note that the default value is 0 if we look up an entry in this dictionary that hasn't been seen yet. So essentially if the parent doesn't exist we create it, else we update it.
- When the parent has been updated, we no longer care about this comment as no other comments back in time can be a comment to this. This just means that since no more comments in the database will have this comment as parent, we can just delete it.
- After this operation, each subreddit dictionary will have a structure as follows:

```
{'t1_cl59b2j': 1,
 't1_cn0yo1t': 1,
 't1_cn45ptk': 1,
 't3_2it1zw': 1,
 't3_2iwf9y': 1,
 't3_2krji5': 1,
 't3_2l8pg9': 1,
 't3_2ldxa7': 2,
 't3_2s1wp2': 1,
 't3_2stxdc': 5,
 't3_2tpq7k': 1,
 't3_2two4x': 1})
```

- We just have the top-level comments left, plus some comments who has a parent that does not exist as an individual entry in the database. This has been discussed on Aula, maybe they are sub comments to a deleted comment and thus without a parent, but since they can't be placed anywhere we should not count them.
- Note that all the values are one too large because of the structure of our top comment, so when we compute the depth we will have to subtract one from all of them.
- For all the dictionaries representing the individual subreddit, extract the values for keys starting with "t3", subtract one and take the mean.

Result:

```
Final Time: 508.0327637195587 sec
Avg Depth     id          name
381.9         t5_2u9jq    counting
343.9         t5_2zlk4    EroticRolePlay
339.2         t5_33wg4    SburbRP
149.5         t5_366p4    TheTwoCenturions
112.0         t5_2x15g    ExploreFiction
95.0         t5_3272o    RidersOfBerk
90.2         t5_323vp    GalacticGuardians
89.4         t5_33lo3    YamakuHighSchool
88.6         t5_2z5u0    randomsuperpowers
83.0         t5_31ryx    ksrp
```