Ministry of Education of Republic of Moldova

Technical University of Moldova CIM Faculty

Anglophone Department

Laboratory work Nr.2



|  |  |  |
| --- | --- | --- |
| **Performed by** |  | |
| **St. gr.FAF-212:** |  | **Nastas C.** |
| **Verifed by asis. univ.:** |  | **Fiștic C.** |

Chişinău 2021

**Contents**

[**1.** **Random vulnerability** 3](#_Toc92144218)

[**1.1** **Solution:** 4](#_Toc92144219)

[**1.2** **Result:** 4](#_Toc92144220)

[**2.** **RSA** 5](#_Toc92144221)

[**2.1** **Solution** 5](#_Toc92144222)

[**2.2** **Result** 6](#_Toc92144223)

[**3**. **Matrix** 7](#_Toc92144224)

[**3.1 Friends** 8](#_Toc92144225)

[**3.2 Sort** 9](#_Toc92144226)

[**3.3 Let's do ratings** 10](#_Toc92144227)

[**3.4 Influential people** 13](#_Toc92144228)

[**3.5 Analyze your content** 15](#_Toc92144229)

[**3.6 Promote it** 15](#_Toc92144230)

[**4. Network** 17](#_Toc92144231)

[**4.1 Popular Hashtag** 17](#_Toc92144232)

[**4.2 Tokenizer** 17](#_Toc92144233)

[**4.3 Top** 17](#_Toc92144234)

# **Random vulnerability**

You've stumbled onto a significant vulnerability in a commonly used cryptographic library. It turns out that the random number generator it uses frequently produces the same primes when it is generating keys.

Exploit this knowledge to factor the (hexadecimal) keys below, and enter your answer as the last six digits of the largest factor you find (in decimal).

Key 1:

1c7bb1ae67670f7e6769b515c174414278e16c27e95b43a789099a1c7d55c717b2f0a0442a7d49503ee09552588ed9bb6eda4af738a02fb31576d78ff72b2499b347e49fef1028182f158182a0ba504902996ea161311fe62b86e6ccb02a9307d932f7fa94cde410619927677f94c571ea39c7f4105fae00415dd7d

Key 2:

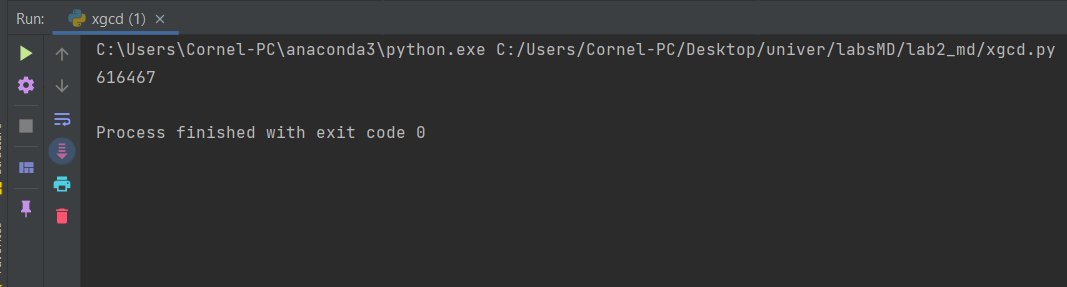
2710e45014ed7d2550aac9887cc18b6858b978c2409e86f80bad4b59ebcbd90ed18790fc56f53ffabc0e4a021da2e906072404a8b3c5555f64f279a21ebb60655e4d61f4a18be9ad389d8ff05b994bb4c194d8803537ac6cd9f708e0dd12d1857554e41c9cbef98f61c5751b796e5b37d338f5d9b3ec3202b37a32f

# **Solution:**

A simple implementation of extended Euclidean algorithm

def extended\_gcd(a: int, b: int) -> int:  
 prev\_x, x = 1, 0  
 prev\_y, y = 0, 1  
 while b:  
 q = a / b  
 x, prev\_x = prev\_x - q \* x, x  
 y, prev\_y = prev\_y - q \* y, y  
 a, b = b, a % b  
 return a  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 inp: hex = '1c7bb1ae67670f7e6769b515c174414278e16c27e95b43a789099a1c7d55c717b2f0a0442a7d49503ee09552588ed9bb6eda4af738a02fb31576d78ff72b2499b347e49fef1028182f158182a0ba504902996ea161311fe62b86e6ccb02a9307d932f7fa94cde410619927677f94c571ea39c7f4105fae00415dd7d'  
 inp1: hex = '2710e45014ed7d2550aac9887cc18b6858b978c2409e86f80bad4b59ebcbd90ed18790fc56f53ffabc0e4a021da2e906072404a8b3c5555f64f279a21ebb60655e4d61f4a18be9ad389d8ff05b994bb4c194d8803537ac6cd9f708e0dd12d1857554e41c9cbef98f61c5751b796e5b37d338f5d9b3ec3202b37a32f'  
  
 n = int(inp, 16)  
 n1 = int(inp1, 16)  
 print(extended\_gcd(n, n1) % 10 \*\* 6)

### **Result:**



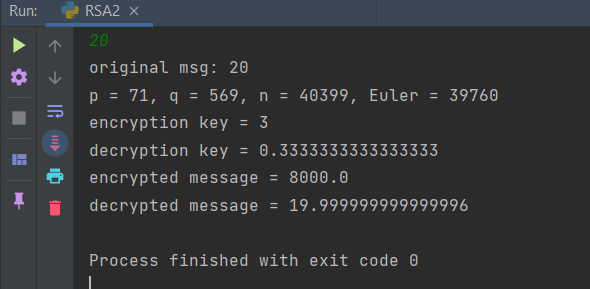
# **RSA**

Make a program that will implement RSA algorithm made by you and with help of this program you can encrypt any string and then decrypt it.

## **Solution**

import random  
import math  
from Crypto.Util.number import isPrime  
  
  
def get\_prime() -> int:  
 primes = [i for i in range(0, 1000) if isPrime(i)]  
 return random.choice(primes)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 p = get\_prime()  
 q = get\_prime()  
 n = p \* q  
 phi = (p - 1) \* (q - 1)  
 e = 2  
 while e < phi:  
 if math.gcd(e, phi) == 1:  
 break  
 else:  
 e += 1  
 msg = int(input())  
 dd = pow(e, -1)  
 d = math.fmod(dd, phi)  
 encryption = pow(msg, e)  
 decryption = pow(encryption, d)  
 encryption = math.fmod(encryption, n)  
 decryption = math.fmod(decryption, n)  
  
 print(f'original msg: {msg}')  
 print(f'p = {p}, q = {q}, n = {n}, Euler = {phi}')  
 print(f'encryption key = {e}')  
 print(f'decryption key = {d}')  
 print(f'encrypted message = {encryption}')  
 print(f'decrypted message = {decryption}')

### **Result**



#### **Algorithm**

* Generating keys:
  + Select two prime numbers **p** and **q**, and compute

**n = p \* q**

* + Calculate the **totient** function
  + Select an integer **e**, such that **e** is a co-prime to and 1 < e <
  + Calculate **d** such that
* Encryption:
* Decryption:

# **3**. **Matrix**

You have a set of 20 people connected via a friendship matrix. The whole list is given in matrix.txt.

**Getting the data:**

def get\_data() -> None:  
 with open('./matrix.txt') as f:  
 names = f.readline().split('|')  
 names = [name.strip() for name in names]  
 for line in f:  
 name = re.search(r'[A-Z][a-z]+ [A-Z][a-z]+', line).group()  
 adj = re.findall(r'\d+', line)  
 adj = [int(adj) for adj in adj]  
 graph\_adj.append(adj)  
 person = {  
 'name': name,  
 'friends': 0,  
 'rating': 0,  
 'adj': [],  
 }  
 for i, \_ in enumerate(adj):  
 person['adj'].append({  
 'name': names[i],  
 'adjacent': adj[i]  
 })  
 persons.append(person)  
 count\_friends(persons)

Create a list of objects with the following structure:

person = {

‘name’,

‘friends’,

‘rating’,

‘adjacencies’: [

{‘name’, ‘adjacent’}

…

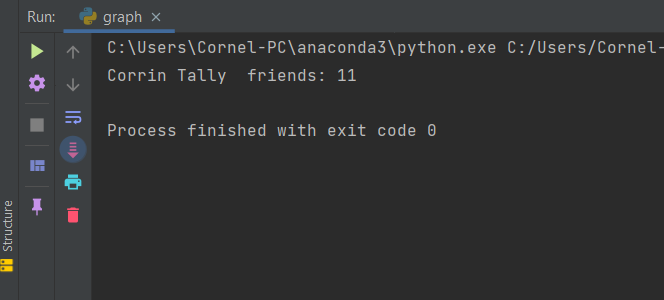
]

## **3.1 Friends**

Find the person with the most friends.

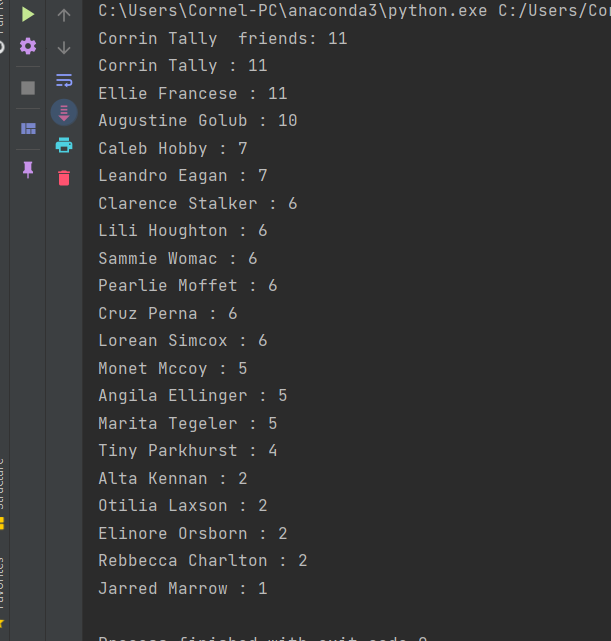
def count\_friends(person\_list: list) -> list:  
 for pers in person\_list:  
 counter: int = 0  
 for friends in pers['adj']:  
 if friends['adjacent']:  
 counter += 1  
 pers['friends'] = counter

def first(persons\_db: list) -> None:  
 persons\_db = sort(persons\_db, 'friends')  
 print(persons\_db[0]['name'], ' friends:', persons\_db[0]['friends'])



## **3.2 Sort**

Sort all the people by the number of friends.



## **3.3 Let's do ratings**

How to do that? Well, each person in the graph is connected to everyone else at some level. Therefore, each person will have a list of connections which is as long as the total list of people in the graph (in our case, 20). You then have to compute the *shortest path* from each of the nodes to each of the other nodes.

For example, let’s say that you found that from node 0 you can reach node 3 in 5 steps (that is, the shortest path connecting nodes 0 and 3 has 5 steps). That means that node 3 will be a connection of level 5 to node 0 and will therefore contribute to 0 with 4 points.

As a procedure, you can take each item *n* and then compute the distances between *n* and all the other vertices of the graph. You can use these distances to compute the value that is added by each of the other *n − 1* vertices to *n*. Sum it and you’ll have the value of vertex *n*.

In order to find the shortest path between two vertices, you’ll have to use Dijkstra's *algorithm*. You can find plenty of implementations of that algorithm online.

Compute the points for each person in our network. Let’s call it ‘Rating’

def create\_dict(person\_list: list) -> dict:  
 dictionary: dict = {}  
 for person in person\_list:  
 values: list = []  
 for pers in person['adj']:  
 if pers['adjacent'] == 1:  
 values.append(pers['name'])  
 dictionary[person['name']] = values  
  
 return dictionary

Makes a dictionary with the following structure:

dictionary = {

‘Person name’: ‘adjacency list’

…

{

def assign\_rating(graph: list, dictionary: dict) -> list:  
 for subgraph in graph:  
 for person in subgraph['adj']:  
 if BFS\_SP(dictionary, subgraph['name'], person['name']) == 0:  
 subgraph['rating'] += BFS\_SP(dictionary, subgraph['name'], person['name'])  
 subgraph['rating'] += BFS\_SP(dictionary, subgraph['name'], person['name']) - 1  
 return graph

Function for assigning each person its corresponding rating which was computed using **Breadth First Search** algorithm implemented below:

def BFS\_SP(graph: list, start: str, goal: str) -> int:  
 explored: list = []  
 queue: list = [[start]]  
  
 if start == goal:  
 return 0  
  
 while queue:  
 path: list = queue.pop(0)  
 node: str = path[-1]  
  
 if node not in explored:  
 neighbours: str = graph[node]  
  
 for neighbour in neighbours:  
 new\_path: list = list(path)  
 new\_path.append(neighbour)  
 queue.append(new\_path)  
  
 if neighbour == goal:  
 return len(new\_path)  
 explored.append(node)  
 return 0

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

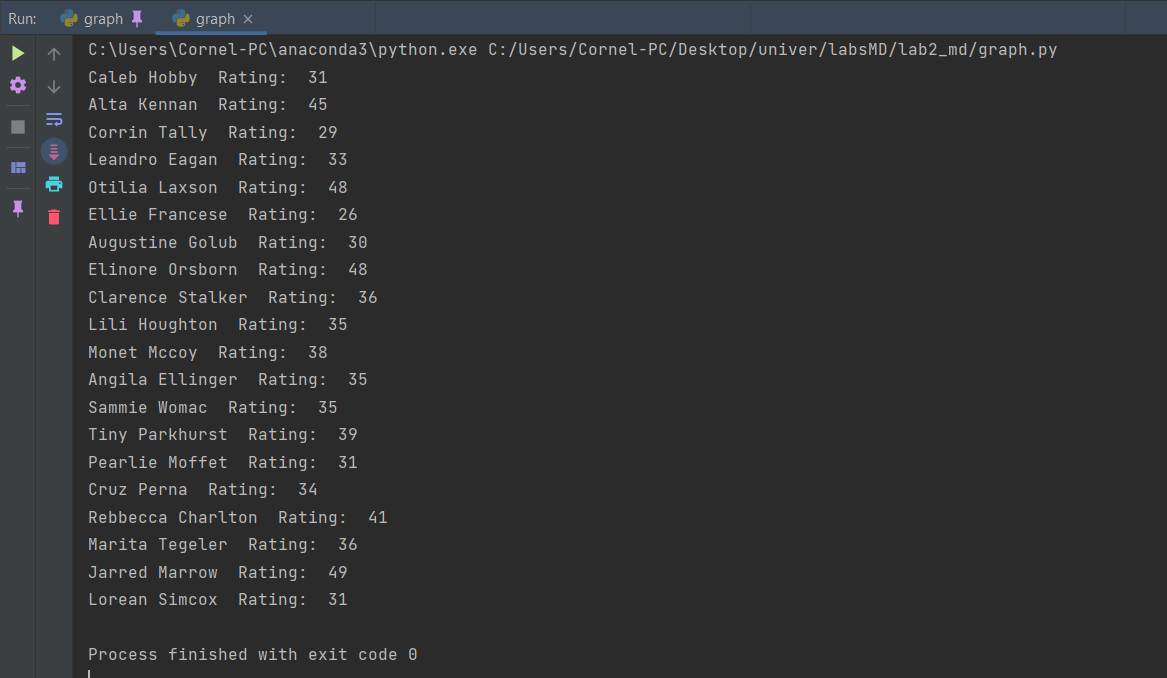
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty or the goal is reached

**And lastly a function grouping everything together:**

def third(persons\_db: list) -> None:  
 dict\_obj: dict = create\_dict(persons\_db)  
 assign\_rating(persons\_db, dict\_obj)  
 for entity in persons\_db:  
 print(entity['name'], ' Rating: ', entity['rating'])



## **3.4 Influential people**

Let’s say that each of these people has a certain rate of posting content. Obviously, people who communicate more are much more influential. Suppose that you need to promote a new brand using social media. We found out how often each of these 20 people writes something on their walls. You can find it in influence.txt

Whom of these people will you contact? Why? Be advised that not only the frequency of posting matters, but also the number of friends!

Use the data from the previous exercise and find the new ‘Rating’ for each person by multiplying it with 0.5 of the posting rate.

Please sort the people by the newly computed rating.

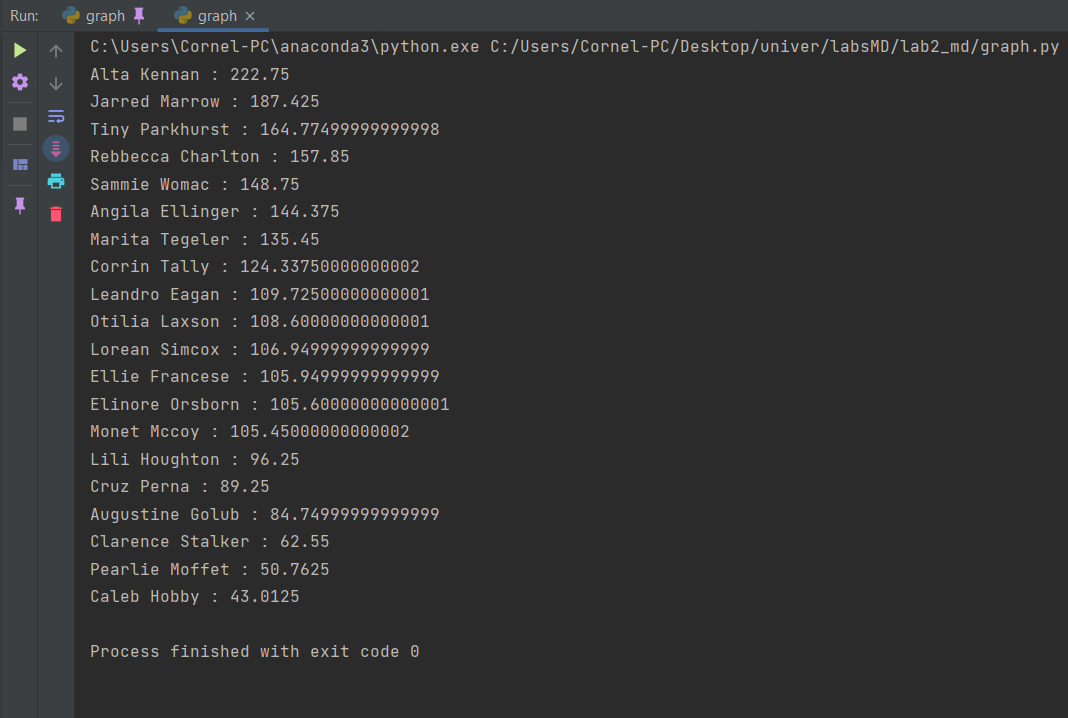
**Getting the data from the file:**

def read\_influence() -> dict:  
 with open('influence.txt', 'r') as file:  
 dictionary: dict = {}  
 for line in file:  
 values: list = line.split(' : ')  
 val: str = ''  
 for x in values[1:len(values)]:  
 val += x  
 dictionary[values[0]] = float(val)  
 return dictionary

**A function grouping everything together**

def fourth(persons\_db: list) -> None:  
 influence\_dict: dict = read\_influence()  
 dict\_obj: dict = create\_dict(persons\_db)  
 persons\_db = assign\_rating(persons\_db, dict\_obj)  
 for entity in persons\_db:  
 entity['rating'] = entity['rating'] \* influence\_dict[entity['name']] \* 0.5  
 persons\_db = sort(persons, 'rating')  
 for person in persons\_db:  
 print(person['name'], ':', person['rating'])

**Result**



## **3.5 Analyze your content**

You are publishing a book and would like to promote it through the use of social media. The book’s title is” From T-Rex to Multi Universes: How the Internet has Changed Politics, Art and Cute Cats.” You have done some research in the world’s most popular social network and have found that the range of interests is stored in interests.txt

Analyze your title and see what specter of interests is your book marketable to.

## **3.6 Promote it**

We have provided you with a list of interests of each of these people. You can find it in interests.txt.

Considering the set of interests, you have chosen, who of them would you market the book to? Let’s say that a person has 5 of her interests coinciding with your books and she has a Rating of 346. Multiply her rating with the 0.2 \* coinciding interests to see a final score. Sort the people by this final score.

Provide us with a list of 5 people we should contact to make your book a bestseller! Please use the names found in people\_interests.txt.

**Function for getting the interests from the file:**

def get\_interests(string: str) -> list:  
 string = string.replace(',', '')  
 string = string.replace('.', '')  
 words: list = string.split(' ')  
 interests: list = []  
 with open('interests.txt', 'r') as file:  
 for line in file:  
 interests.append(line.strip('\n'))  
 return list(set(interests).intersection(words))

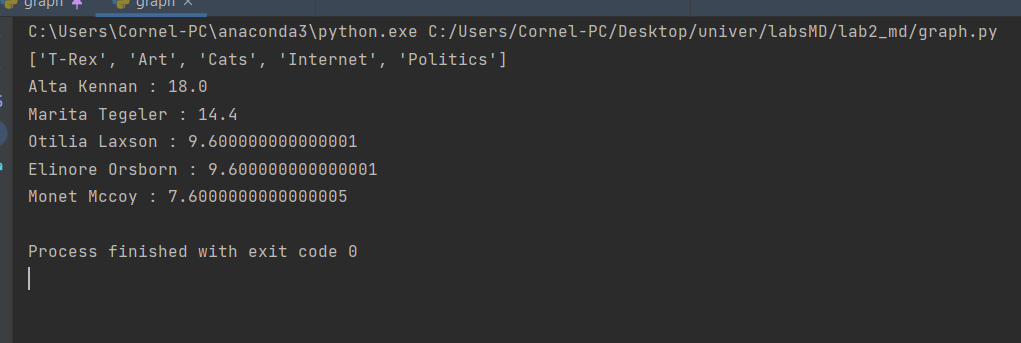
**Function for getting the promotion rating:**

def promotion\_rating(tags: list, persons\_db: list) -> list:  
 with open('people\_interests.txt', 'r') as file:  
 people\_tags: dict = {}  
 for line in file:  
 values: list = line.split(' : ')  
 val: str = ''  
 for x in values[1:len(values)]:  
 val += x  
 val = val.replace('\n', '')  
 val\_list: list = val.split(' ')  
 people\_tags[values[0]] = val\_list  
  
 for entity in persons\_db:  
 entity['rating'] = entity['rating'] \* 0.2 \* len(list(set(tags).intersection(people\_tags[entity['name']])))  
 return persons\_db

**A function grouping everything together:**

def last(persons\_db: list) -> None:  
 title: str = 'From T-Rex to Multi Universes: How the Internet has Changed Politics, Art and Cute Cats.'  
 interests: list = get\_interests(title)  
 print(interests)  
  
 dict\_obj: dict = create\_dict(persons\_db)  
 assign\_rating(persons\_db, dict\_obj)  
 persons\_db = promotion\_rating(interests, persons\_db)  
  
 persons\_db = sort(persons\_db, 'rating')  
 top5: list = itertools.islice(persons\_db, 5)  
  
 for person in top5:  
 print(person['name'], ':', person['rating'])

**Result**



# **4. Network**

#### The dataset

The dataset is a text file where every line represents a JSON object that describes a tweet (tweet.txt). It was fetched using twitter stream API, hence we're dealing with real life data (yay).

**Libraries used throughout the program:**

import itertools  
import json  
import operator  
from nltk.tokenize import word\_tokenize  
import regex as re

**Reading the data (also separating deleted tweets with existing ones)**

def get\_data() -> dict:  
 tweets: list = []  
 with open('./tweets.txt') as f:  
 for l in f:  
 tweets.append(json.loads(l))  
  
 updated\_tweets: list = []  
 for tweet in tweets:  
 if 'created\_at' in tweet:  
 updated\_tweets.append(tweet)  
 return updated\_tweets

## **4.1 Popular Hashtag**

Write a program that prints on the screen 10 most popular #hashtags followed by the number of occurrences of the #hashtag.

def get\_hashtags(db: list) -> dict:  
 tags: dict = {}  
 for entity in db:  
 words: list = entity['text'].split()  
 for word in words:  
 if re.match(r"#[a-zA-Z]\*", word):  
 if word in tags:  
 tags[word] += 1  
 else:  
 tags[word] = 1  
 return tags

**Looking for hashtags by separating every word from each tweet and using Regex (regular expression) as a helper function.**

## **4.2 Tokenizer**

Let's do some emotional analyses.

In this file AFINN-111.txt you'll find an emotion dictionary for English words. Every word mentioned in the dictionary is followed by a numerical value in the range of -5 to 5. The numerical value describes the word emotional impact where -5 is the most negative and 5 is the most positive.

Your task is to find the emotional value for every tweet. First step would be to extract every word from the tweet body. I recommend using an nltk tokenizer (similar to TPI Lab 3). Then you find out the emotional value for every word (if it has one). You finish by summing the emotion rating.

Write a program that will store the computed result in a text file. Every line should represent the tweet id followed by the computed emotional value.

def get\_emotionalRating() -> dict:  
 emotional\_dict: dict = {}  
 with open('AFINN-111.txt', 'r') as file:  
 for line in file:  
 values: list = line.split('\t')  
 val: str = ''  
 for x in values[1:len(values)]:  
 val += x  
 val = val.replace('\n', '')  
 emotional\_dict[values[0]] = int(val)  
 return emotional\_dict

**With this function we are creating a dictionary which we are going to reference later on for creating the ratings of each tweet (positive/negative)**

def tokenize(db: dict, emo\_rating: dict) -> dict:  
 tweets\_rating: list = []  
 for entity in db:  
 words = (word\_tokenize(entity['text']))  
 rating: int = 0  
 tweet = {  
 'id': entity['id'],  
 'text': entity['text'],  
 'rating': int  
 }  
 for word in words:  
 if word in emo\_rating:  
 rating += emo\_rating[word]  
 tweet['rating'] = rating  
 tweets\_rating.append(tweet)  
 return tweets\_rating

**This function is looking for each word in the tweet and then looking for it in the dictionary created earlier for the rating, in the end assigning each tweet with the structure { id , text, rating } its rating, text and id (per instructions).**

## **4.3 Top**

Write a program that prints on the screen 10 most positive tweets and 10 most negative tweets.

def top(db: list):  
 db.sort(key=operator.itemgetter('rating'), reverse=True)  
 return db[:10]  
  
  
def bottom(db: list):  
 db.sort(key=operator.itemgetter('rating'))  
 return db[:10]

**These two helper functions will get us first 10 most positive tweets, and last 10 most negative tweets**

**Results:**