# Exercise 1

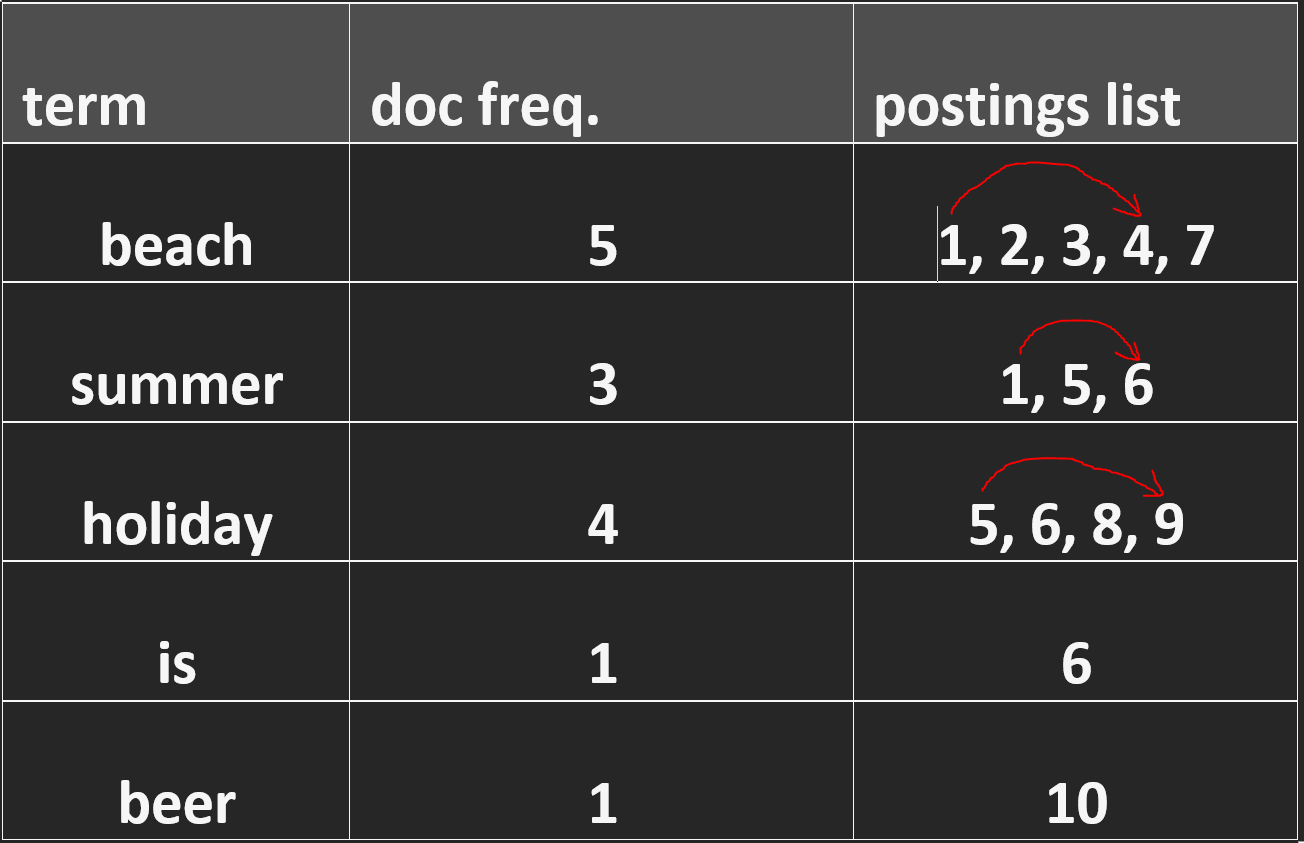
# Task 1

## Subtask A

|  |  |  |
| --- | --- | --- |
| **term** | **doc freq.** | **postings list** |
| **beach** | **5** | **1, 2, 3, 4, 7** |
| **summer** | **3** | **1, 5, 6** |
| **holiday** | **4** | **5, 6, 8, 9** |
| **is** | **1** | **6** |
| **beer** | **1** | **10** |

## Subtask B

Skip pointers added:



Example:

* Add skip pointer for postings list for the term beach from 1 to 4
* The query:
  + **beach AND holiday**
* We start at the postings lists with the entries **beach = 1** and **holiday = 5**
* So we can take the skip pointer at the term beach from 1 to 4, so we are at beach = 4
* Next we compare beach = 7 🡪 Now we know that there is no match
* **The skip pointer saved us the comparisons ob 1,2,3 for the term beach.**
  + Without skip pointers we need more comparisons, thus this query can be answered in a more efficient way with these skip pointers.

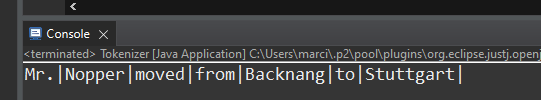
# Task 2

## Pseudo-Code:

1. Initialize currentChar = 0
2. Initialize lastStop = 0
3. Initialize resultList = []
4. Iterate over inputString at position currentChar with condition currentChar < inputString.length
   1. If inputString[currentChar] is Whitespace add substring of inputString starting from index lastStop, ending at index currentChar to resultList. Set lastStop = currentChar+1
   2. If currentChar equals inputString.length – 1 add substring of inputString starting from index lastStop, ending at index currentChar+1 to resultList.
   3. currentChar++ and continue iteration at 4
5. return resultList

## Java-Code implementation:

1. **public class Tokenizer {**
2. **public** **static** String example = "Mr. Nopper moved from Backnang to Stuttgart";
4. **static** List<String> tokenize(String input) {
5. **List<String> result = new ArrayList();**
7. **int** lastStop = 0;
9. **for**(**int** currentChar=0; currentChar<input.length(); currentChar++) {
10. **if(Character.isWhitespace(input.charAt(currentChar))) {**
11. result.add(input.substring(lastStop, currentChar));
12. lastStop = currentChar+1;
13. }
14. **if**(currentChar == input.length()-1) {
15. **result.add(input.substring(lastStop, currentChar+1));**
16. }
17. }
18. **return** result;
19. }
21. **public** **static** **void** main(String[] args) {
22. List<String> test = tokenize(example);
23. test.**forEach**(t -> System.out.print(t+ "|"));
24. }
25. **}**

Output:

The presented algorithm has in general a linear runtime, because it only iterates the inputString once with the for-Loop and the loop variable currentChar is never changed/reset inside the loop. Therefore the for-loop is executing exactly n-times for an inputString of length n. All other operations (variable initialization) have a runtime of O(1). This results in a O(n) runtime for the algorithm.

# Task 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | i → | H | u | a | s |
| j ↓ | 0 | 1 | 2 | 3 | 4 |
| H | 1 | 0  stay | 1  delete | 2  delete | 3  delete |
| a | 2 | 1  insert | 1  replace | 1  stay | 2  delete |
| u | 3 | 2 insert | 1  stay | 1  transpose | 2  delete, replace |
| s | 4 | 3 insert | 2 insert | 2  insert, replace | 1  stay |

|  |  |  |  |
| --- | --- | --- | --- |
| **Operations** | | **add** | **condition** |
| **stay** | Di-1, j-1 | 0 | if ui = vj |
| **replace** | Di-1, j-1 | 1 |  |
| **insert** | Di,j-1 | 1 |  |
| **delete** | Di-1,j | 1 |  |
| **transpose** | Di-2,j-2 | 1 | ui = vj-1 && ui-1 = vj |

## Explanation

* The Levenshtein distance between „Haus“ and „Huas“ is 2, because we need to perform two times the replace operation, replacing a with u and the u with a.
* The Damerau-Levenshtein distance between „Haus“ and „Huas“ is 1, because we need to perform one transpose operation, which switches the letters u and a.
* The operations that produced the value in the matrix are described in the table. If there is more than one possible operation these are mentioned in a comma separated list.
* The final distance can be retrieved from the bottom right corner of the matrix.  
  We can backtrack the steps starting from the bottom right corner:
  + Stay 🡪 step -1, -1 in matrix
  + Transpose 🡪 step -2,-2 in matrix
  + Stay 🡪 We reached the the top left corner of the matrix

# Task 4

## Export Tweets to disk 🡪 createTweetDB.py

**import** pandas **as** pd

**import** numpy **as** np

# this script exports all tweets as .txt files to disk. The filename is the tweet handle

nrows = 0

folder = "E:/tweets/"

# read csv

**print**("reading csv...")

**if**(nrows>0):

# only read nrows number of tweets

data = pd.read\_csv('twitter.csv', sep="\t",

names=["handle", "userid", "username", "tweet"],

dtype={"handle": np.int64, "userid": **str**, "username": **str**, "tweet": **str**},

header=None, nrows=nrows,

quoting=3)

**else**:

# read all tweets

data = pd.read\_csv('twitter.csv', sep="\t",

names=["handle", "userid", "username", "tweet"],

dtype={"handle": np.int64, "userid": **str**, "username": **str**, "tweet": **str**},

header=None,

quoting=3)

# export tweets to txt files where filname is equal to the tweet handle

**for** item **in** data.iloc:

**with** **open**(folder+**str**(item.handle)+'.txt', 'w', encoding='utf-8') **as** f:

f.write(item.tweet)

## Build index file 🡪 createIndex.py

**import** csv

**from** datetime **import** datetime

**from** itertools **import** chain

**import** numpy **as** np

**import** nltk

**import** pandas **as** pd

**import** pickle

# This script builds a non-positional inverted index. Postings lists are stored as .csv files on disk.

# The index is stored as pickled python object on disk and can be loaded by the queryEngine later.

# Stopwords of the 8 most common languages of the tweets are filtered, as well as words with length = 1.

# The index is a dictionary with the term as key and the values docFrequency and postings

# docFrequency is an integer which counts how often the term occurs in all documents

# postings is an integer with the file ID for the postings list.

# E.g. postings=100 means, the posting list is stored in the file p100.csv

# config

indexName = datetime.today().strftime('%Y%m%d-%H\_%M\_%S') + "\_index.pickle"

nrows = 0

# token normalization

**def** normalize(line):

# this function removes special characters, newlines and tab from tweets, tokenizes the text and sets all terms

# to lowercase

text = line.tweet

**if** (**type**(text) != **str**):

**return** ""

**for** ch **in** ['[NEWLINE]', '[TAB]', '#', '\\', '`', '\*', '\_', '{', '}', '[', ']',

'(', ')', '>', '+', '-', '.', '!', '?', '$', '\'', '"', '/']:

**if** ch **in** text:

text = text.replace(ch, " ")

**return** text.lower().split()

# index function

**def** index(filename):

# read csv

**print**("reading csv...")

**if** (nrows > 0):

data = pd.read\_csv(filename, sep="\t",

names=["handle", "userid", "username", "tweet"],

dtype={"handle": np.int64, "userid": **str**, "username": **str**, "tweet": **str**},

header=None,

nrows=nrows,

quoting=3)

**else**:

data = pd.read\_csv(filename, sep="\t",

names=["handle", "userid", "username", "tweet"],

dtype={"handle": np.int64, "userid": **str**, "username": **str**, "tweet": **str**},

header=None,

quoting=3)

# get stopwords

nltk.download('stopwords')

languages = ['english', 'german', 'spanish', 'portuguese', 'italian', 'french', 'turkish', 'dutch']

stopwords = **dict**.fromkeys([i **for** i **in** chain.from\_iterable([nltk.corpus.stopwords.words(l) **for** l **in** languages])])

# gather data

progress = 0;

**print**("building inverted index ...")

results = {}

**for** item **in** data.iloc:

progress = progress + 1;

**if** (progress % 10000 == 0):

**print**("processing line " + **str**(progress) + " ...")

# tokenize tweet

terms = normalize(item)

**for** term **in** terms:

# ignore stopwords and single characters

**if** term **in** stopwords **or** **len**(term) < 2:

**continue**

**if** term **in** results:

entry = results[term]

entry["docFreq"] = entry["docFreq"] + 1

entry["postings"].add(item.handle)

**else**:

results[term] = {"docFreq": 1, "postings": **set**([item.handle])}

# save postings lists as file

**print**("create postings lists files...")

fileId = 0

**for** term **in** results:

# for each term write postings lists to csv file on disk

entry = results[term]

fileName = **str**(fileId) + '.csv'

**file** = **open**('E:/postings\_lists/p' + fileName, 'w', newline='')

writer = csv.writer(**file**)

postings = **sorted**(**list**(entry['postings']))

writer.writerows([[handle] **for** handle **in** postings])

entry['postings'] = fileId

fileId = fileId + 1

**if** fileId % 10000 == 0:

**print**("writing file " + **str**(fileId))

# pickle index as file on disk

**print**("pickling results...")

f = **open**(indexName, 'wb')

pickle.dump(results, f)

f.close()

**print**("done.")

**return** results

# run

indexData = index("twitter.csv")

## Query engine 🡪 queryEngine.py

**import** pickle

**from** pathlib **import** Path

**import** pandas **as** pd

# config

indexName = "20211108-14\_03\_27\_index.pickle"

postings\_dir = "E:/postings\_lists/"

tweets\_dir = "E:/tweets/"

showResults = 0

index = None

# functions

**def** query(term):

**if** term **in** index:

# read csv file from disk and return values as list

postings = pd.read\_csv(postings\_dir + 'p' + **str**(index[term]['postings']) + '.csv')

**return** postings.iloc[:, 0].values.tolist()

**else**:

# term not in index, return empty list

**return** []

**def** queryAND(term1, term2):

# get postings lists

list1 = **iter**(query(term1))

list2 = **iter**(query(term2))

# intersection algorithm:

# get start values smallValue, bigValue not sorted yet!

smallV = next(list1, None)

bigV = next(list2, None)

# if one term has no results, AND query has no results

**if** smallV == None **or** bigV == None:

**return** []

# init conditions 🡪 list1 with smallV tries to catchup to list2 to bigV

**if** (smallV > bigV):

temp = smallV

temp2 = list1

smallV = bigV

list1 = list2

bigV = temp

list2 = temp2

matches = []

# intersect lists

**while** (smallV <= bigV):

# get value

smallV = next(list1, None)

# one list reached the end, break and return

**if** (smallV == None):

**break**

# match found!

**if** (smallV == bigV):

matches.append(smallV)

# smallValue passed bigValue => swap values and lists

**if** (smallV > bigV):

temp = smallV

temp2 = list1

smallV = bigV

list1 = list2

bigV = temp

list2 = temp2

**return** matches

**def** displayResults(postingslists, nOfResults=showResults):

results = []

# if results > nOfResults show only nOfResults many results

maxIndex = **len**(postingslists)

**if** nOfResults != 0 **and** nOfResults < maxIndex:

maxIndex = nOfResults

# if no results

**if** maxIndex == 0:

**print**("No results...")

**return** results

# iterate over ressults, open tweet.txt file from disk and print tweet in a line

**for** i **in** **range**(0, maxIndex):

**file** = tweets\_dir + **str**(postingslists[i]) + ".txt"

**with** **open**(**file**, 'r', encoding='utf-8') **as** f:

results.append(**str**(postingslists[i])+" : "+f.read())

**for** i **in** **range**(0, **len**(results)):

**print**(results[i])

**return** results

# run the query loop:

# check if index file exists

results = Path(indexName);

**if** results.is\_file():

# unpickle index file

**print**("file found, loading...")

pickleFile = **open**(indexName, 'rb')

index = pickle.load(pickleFile)

# UI loop, let user enter queries

**while** True:

queryString = **input**("Enter your search query: ")

# exit program with q

**if** (queryString == "q"):

**print**("Quit called. Goodbye, see you later alligator...")

**break**

# process query

terms = queryString.lower().split()

**if** (**len**(terms) == 1):

# one term, use simple query

results = query(terms[0])

**else**:

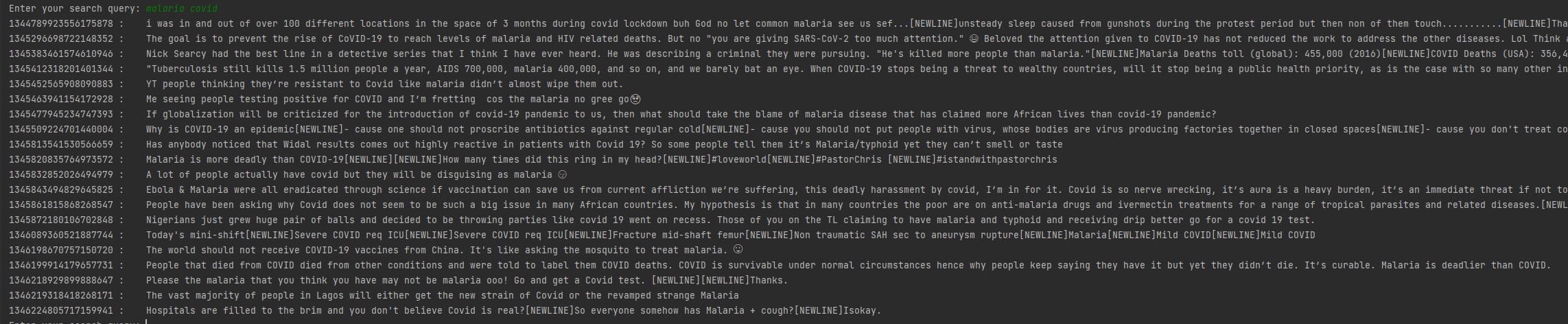
# two terms, use AND query

results = queryAND(terms[0], terms[1])

# display results

resultDocuments = displayResults(results)

## Results & Example query



* The information need **„show me tweets of people who talk about the side effects of malaria and COVID vaccines“** is a phrase query
* We need **3 or 4 words** to describe the information need 🡪 e.g. **malaria covid vaccine effect**
* However our query engine is currently build to allow phrase queries only for 2 words, therefore the query **„malaria covid“** is the best we can do with this implementation.
* But it is easy to extend the query engine to allow phrase queries with more than 2 words, by consecutively intersecting the postings lists of the words.
  + E.g. r1 = query(term1, term2)
  + Intersect r1 with query(term3) etc.