

Cunico, Whaley, Hoxha - IRTM

Exercise 2 Solutions

Task 1

- What information does the task description contain that the master gives to the parser?
 - The master assigns the parser a split (of document IDs) and specifies the intermediate (segment) file to which the parser must write results.
- What information does the parser report back to the master upon completing the task?
 - The parser reads each document and emits term/document ID pairs.
- What information does the task description contain that the master gives to an inverter?
 - The master assigns each term partition to the inverter.
- What information does the inverter report back to the master upon completing the task?
 - The inverter gathers all document IDs for a given term ID into a unified list in the reduce phase.

Task 2

2.1 Compute the coefficients k and b .

Heap's law states that $M = k * T^b$ where T is the number of tokens in the collection and M is the vocabulary size. The k and b parameters are both variable. Thus it is a method for predicting vocab size. To find k and b , we can solve the equations:

$$10000 = k * 1000000^b$$

$$3000 = k * 100000^b$$

$$\log(10000) = b * \log(1000000) + k$$

$$\log(3000) = b * \log(100000) + k$$

$$b = 0.5228, k = 7.2979$$

2.2 Compute the expected vocabulary size for a larger collection (100,000,000 tokens)

$$7.2979 * (100000000^{0.5228}) = 111070.82$$

The estimated vocabulary size is therefore approximately 111070.82.

Task 3

Calculate the variable byte code and gamma code for 216

Variable byte code: $216_{10} = 11011000_2$ The binary code can not be stored within 7 bits, so the continuation bit goes to 1 and we encode the rest: **0000000111011000**.

Gamma code: First we take the offset: 1011000. Then take the length (7 ones followed by a 0) 11111110. Combining them we get **111111101011000**.

Task 4

From the following sequence of gamma-coded gaps, reconstruct first the gap sequence, then the postings sequence.

1111011000100110000

1. 111101100 => gap1 is $11100_2 = 28_{10}$
2. 0 => gap2 is 1
3. 100 => gap3 is $10_2 \Rightarrow 2_{10}$
4. 11000 => gap4 is $100_2 \Rightarrow 4_{10}$
5. 0 => gap5 is 1

So the posting list is: 28 -> 1 -> 2 -> 4 -> 1 and this refers to posting list with document ids: 28 -> 29 -> 31 -> 35 -> 36

Task 5

We have the following dictionary content given:

• A AB ABACUS ABACUSES ABAFT ABAKA ABAKAS ABALONE ABALONES
ABAMP ABAMPERE ABAMPERES ABAMPS AC

How much memory do you need if you allocate a fixed amount of memory to each string (i.e., the minimum required given these instances)?

- So the longest word here is 9 characters so it will be stored in 9 bytes. Every string should be allocated the same fixed size so in this case each string is allocated 9 bytes. There are 14 strings x 9 bytes, so in total we need to allocate 126 bytes to store them.

How much do you save if you store all these strings as one string? (called “Dictionary as a string” in the lecture)

- If we store these strings as one string we would get:
**AABABACUSABACUSESABAFTABAKAABAKASABALONEABALONESABAMPAB
AMPEREABAMPERESABAMPSAC**

So in total there would be 78 characters which can be stored in 78 bytes. But when “dictionary as a string” is used, pointers are required too, which requires more memory allocation. In this case we need memory for 14 pointers. So in total there would be 78 bytes to store the “dictionary string” and 14 x memory allocated for each pointer.

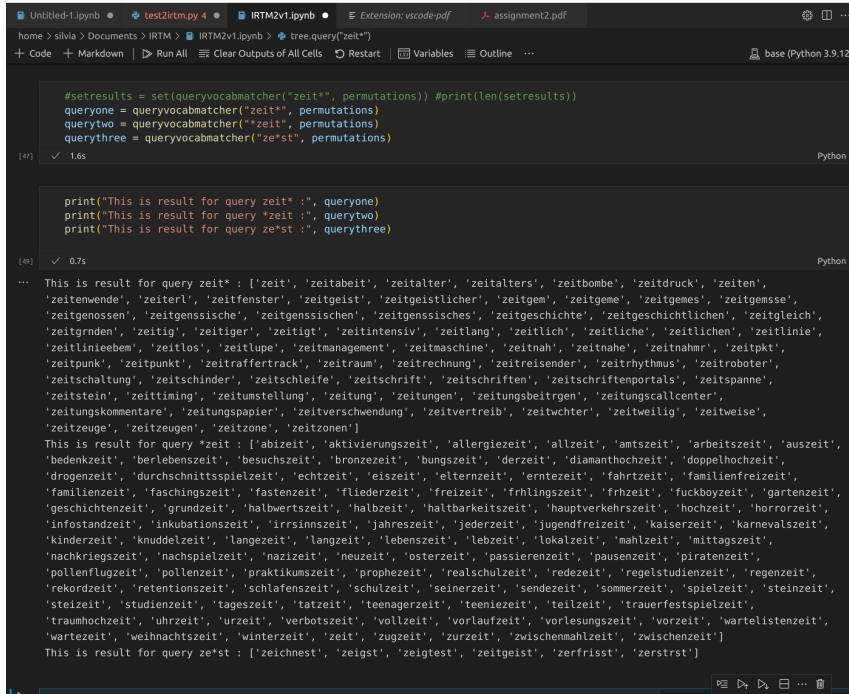
How much memory would you save if you do blocking in addition (no front coding)?
(set parameters like the memory consumption of a pointer appropriately, if necessary)

- With blocking in addition the “dictionary string” becomes:
**1A2AB6ABACUS8ABACUSE5ABAFT5ABAKA6ABAKAS7ABALONE8ABALONES5AB
AMP8ABAMPERE9ABAMPERES6ABAMPS2AC**

This upper string has 92 bytes (78 characters + 14 blockings).

Results of the coding task:

With permutator:

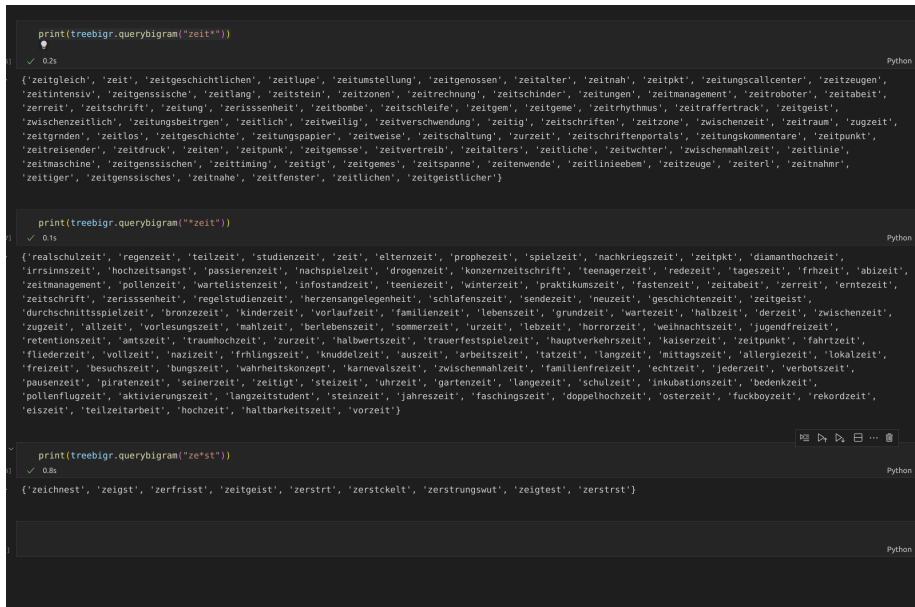


```
#setresults = set(queryvocabulary("zeit", permutations)) #print(len(setresults))
queryone = queryvocabulary("zeit", permutations)
querytwo = queryvocabulary("zeit", permutations)
querythree = queryvocabulary("ze*st", permutations)

print("This is result for query zeit :", queryone)
print("This is result for query *zeit :", querytwo)
print("This is result for query ze*st :", querythree)

This is result for query zeit : ['zeit', 'zeitabel', 'zeitalter', 'zeitalters', 'zeitbombe', 'zeitdruck', 'zeiten',
'zeitewende', 'zeiterl', 'zeitfenster', 'zeitgeist', 'zeitgeistlicher', 'zeitgem', 'zeitgeme', 'zeitgemess', 'zeitgemesse',
'zeitgenossen', 'zeitgenossische', 'zeitgenossischen', 'zeitgenossisches', 'zeitgeschichte', 'zeitgeschichtlichen', 'zeitgleich',
'zeitgrnden', 'zeitig', 'zeitiger', 'zeitigt', 'zeitintensiv', 'zeitlang', 'zeitlich', 'zeitliche', 'zeitlichen', 'zeitlinie',
'zeitlinieeben', 'zeitlos', 'zeitlupe', 'zeitmanagement', 'zeitmaschine', 'zeitnah', 'zeitnahe', 'zeitnahm', 'zeitpkt',
'zeitpunkt', 'zeitpunkt', 'zeitraffertrack', 'zeitraum', 'zeitrechnung', 'zeitreisender', 'zeitrhythmus', 'zeitroboter',
'zeitschaltung', 'zeitschinder', 'zeitschleife', 'zeitschrift', 'zeitschriften', 'zeitschriftenportals', 'zeitspanne',
'zeitstein', 'zeittiming', 'zeitumstellung', 'zeitung', 'zeitungen', 'zeitungsbeitrgen', 'zeitungscallcenter',
'zeitungskommentare', 'zeitungspapier', 'zeitverschwendung', 'zeitvertreib', 'zeitwchter', 'zeitweilig', 'zeitweise',
'zeitzeuge', 'zeitzeugen', 'zeitzone', 'zeitzone']
This is result for query *zeit : ['abizeit', 'aktivierungszeit', 'allergiezeit', 'allzeit', 'amtszeit', 'arbeitszeit', 'auszeit',
'bedenzeit', 'berlebenszeit', 'besuchszeit', 'bronzzeit', 'bungszeit', 'derzeit', 'diamanthochzeit', 'doppelhochzeit',
'drogenzeit', 'durchschnittsspielzeit', 'echtzeit', 'eiszeit', 'elternzeit', 'erntzeit', 'fahrtzeit', 'familienfreizeit',
'familienzeit', 'faschingszeit', 'fastenzeit', 'fliederzeit', 'freizeit', 'frhlingszeit', 'frhzeit', 'fuckboyzeit', 'gartenzeit',
'geschichtenzeit', 'grundzeit', 'halbwertszeit', 'halbezeit', 'haltbarkeitszeit', 'hauptverkehrszeit', 'hochzeit', 'horrorzeit',
'infostandzeit', 'inkubationszeit', 'irrsinnszeit', 'jahreszeit', 'jederzeit', 'jugendfreizeit', 'kaiserzeit', 'karnevalszeit',
'kinderzeit', 'knuddelzeit', 'langezeit', 'lebenszeit', 'lebezeit', 'lokalzeit', 'mahlzeit', 'mittagszeit',
'nachkriegszeit', 'nachspielzeit', 'nazizeit', 'neuzzeit', 'osterzeit', 'passierenzeit', 'pausenzeit', 'piratenzeit',
'pollenflugzeit', 'pollenzeit', 'praktikumszeit', 'prophezeit', 'realschulzeit', 'redezeit', 'regelstudienzeit', 'regenzeit',
'rekordzeit', 'retentionszeit', 'schlafenszeit', 'schulzeit', 'seinerzeit', 'sondezeit', 'sommerzeit', 'spielzeit', 'steinzeit',
'steizeit', 'studienzeit', 'tageszeit', 'tatzeit', 'teenagerzeit', 'teeniezeit', 'teilzeit', 'trauerfestspielzeit',
'traumhochzeit', 'uhrzeit', 'urzeit', 'verbotszeit', 'vollzeit', 'vorlaufzeit', 'vorlesungszeit', 'vorzeit', 'wartelistenzeit',
'wartezeit', 'weihnachtszeit', 'winterzeit', 'zeit', 'zugzeit', 'zurzeit', 'zwischenmahlzeit', 'zwischenzeit']
This is result for query ze*st : ['zeichnest', 'zeitgst', 'zeitgest', 'zeitgeist', 'zerfrisst', 'zerstst']
```

With bigramindex:



```
print(treebigr.querybigram("zeit"))

('zeitgleich', 'zeit', 'zeitgeschichtlichen', 'zeitlupe', 'zeitumstellung', 'zeitgenossen', 'zeitalter', 'zeitnah', 'zeitpkt', 'zeitungscallcenter', 'zeitzeugen',
'zeitintensiv', 'zeitgenossische', 'zeitlang', 'zeitstein', 'zeitzone', 'zeitrechnung', 'zeitschinder', 'zeitungen', 'zeitmanagement', 'zeitroboter', 'zeitabel',
'zerzeit', 'zeitschrift', 'zeitung', 'zerissenheit', 'zeitbombe', 'zeitschleife', 'zeitgem', 'zeitgeme', 'zeitrhythmus', 'zeitraffertrack', 'zeitgeist',
'zwischenzeitlich', 'zeitungsbeitrgen', 'zeitlich', 'zeitweilig', 'zeitverschwendung', 'zeitig', 'zeitschriften', 'zeitzone', 'zwischenzeit', 'zeitraum', 'zugzeit',
'zeitgrnden', 'zeitlos', 'zeitgeschichte', 'zeitungspapier', 'zeitweise', 'zeitschaltung', 'zurzeit', 'zeitschriftenportals', 'zeitungskommentare', 'zeitpunkt',
'zeitreisender', 'zeitdruck', 'zeiten', 'zeitpunkt', 'zeitgemess', 'zeitvertreib', 'zeitalters', 'zeitliche', 'zeitwchter', 'zwischenmahlzeit', 'zeitlinie',
'zeitmaschine', 'zeitgenossischen', 'zeittiming', 'zeitigt', 'zeitgemess', 'zeitspanne', 'zeitewende', 'zeitlinieeben', 'zeitzeuge', 'zeiterl', 'zeitnahm',
'zeitiger', 'zeitgenossisches', 'zeitnahe', 'zeitfenster', 'zeitlichen', 'zeitgeistlicher')

print(treebigr.querybigram("*zeit"))

('realschulzeit', 'regenzeit', 'teilzeit', 'studienzeit', 'zeit', 'elternzeit', 'prophezeit', 'spielzeit', 'nachkriegszeit', 'zeitpkt', 'diamanthochzeit',
'irrsinnszeit', 'hochzeitsangst', 'passierenzeit', 'nachspielzeit', 'drogenzeit', 'konzernzeitschrift', 'teenagerzeit', 'redezeit', 'tageszeit', 'frhzeit', 'abizeit',
'zeitmanagement', 'pollenzeit', 'wartelistenzeit', 'infostandzeit', 'teeniezeit', 'winterzeit', 'praktikumszeit', 'fastenzeit', 'zeitobzeit', 'zerzeit', 'erntzeit',
'zeitschrift', 'zerissenheit', 'herzensangelegenheit', 'schlafenszeit', 'sondezeit', 'neuzzeit', 'geschichtenzeit', 'zeitgeist',
'durchschnittsspielzeit', 'bronzzeit', 'kinderzeit', 'vorlaufzeit', 'familienzeit', 'lebenszeit', 'grundzeit', 'wartezeit', 'halbezeit', 'derzeit', 'zwischenzeit',
'zugzeit', 'allzeit', 'vorlesungszeit', 'mahlzeit', 'berlebenszeit', 'sommerzeit', 'urzeit', 'lebezeit', 'horrorzeit', 'weihnachtszeit', 'jugendfreizeit',
'retentionszeit', 'amtszeit', 'traumhochzeit', 'zurzeit', 'halbwertszeit', 'trauerfestspielzeit', 'hauptverkehrszeit', 'kaiserzeit', 'zeitpunkt', 'fahrtzeit',
'fliederzeit', 'vollzeit', 'nazizeit', 'frhlingszeit', 'knuddelzeit', 'auszeit', 'arbeitszeit', 'tatzeit', 'langezeit', 'mittagszeit', 'allergiezeit', 'lokalzeit',
'freizeit', 'besuchszeit', 'bungszeit', 'wahrheitskonzept', 'karnevalszeit', 'zwischenmahlzeit', 'familienfreizeit', 'echtzeit', 'jederzeit', 'verbotszeit',
'pausenzeit', 'piratenzeit', 'seinerzeit', 'schulzeit', 'steizeit', 'teilzeit', 'gartenzeit', 'langezeit', 'schulzeit', 'inkubationszeit', 'bedenzeit',
'pollenflugzeit', 'aktivierungszeit', 'langzeitstudent', 'steinzeit', 'jahreszeit', 'faschingszeit', 'doppelhochzeit', 'osterzeit', 'fuckboyzeit', 'rekordzeit',
'eiszeit', 'teilzeitarbeit', 'hochzeit', 'haltbarkeitszeit', 'vorzeit']

print(treebigr.querybigram("ze*st"))

('zeichnest', 'zeitgst', 'zerfrisst', 'zeitgeist', 'zerstst', 'zerstickelt', 'zerstrungswut', 'zeitgest', 'zerstst')
```

```
In [5]: import pandas as pd
from nltk.stem import PorterStemmer
import nltk
import re
import preprocessor as p
```

```
In [6]: def read_csv(path):
data = pd.read_csv(path, sep='\t', on_bad_lines='skip',
names=['DATE', 'ID', 'HANDLE', 'NAME', 'DATA'])
#drop duplicates
data = data.drop_duplicates(subset=None, keep='first', inplace=False)
mydata = data[['ID', 'DATA']].copy()
return mydata
textt = mydata['DATA']
return textt
#print(mydata.iloc[:3])
```

```
In [7]: mydata = read_csv("/home/silvia/Documents/IRTM/tweets.csv")
print(mydata.iloc[:3])
```

```

          ID                                     DATA
0  965734992633565184  @knakatani @ChikonJugular @jooofford @SteveBlog...
1  965706998946893824      @FischerKurt Lady, what's a tumor? #KippCharts
2  965695626150326273  @Kings_of_Metal Ohne Diagnoseverdacht ist es n...
```

```
In [8]: def clean_and_tokenize(text):
# Use twitter preprocessor to remove mentions, links, hashtags, and emojis
text = p.clean(text)
# Remove special characters
text = re.sub(r'\[NEWLINE\]', " ", text)
text = re.sub(r'\[TAB\]', " ", text)# Lower case
text = text.lower()
# Expand common contractions
text = re.sub(r"won't", "will not", text)
text = re.sub(r"can't", "cannot", text)
text = re.sub(r"d", " would", text)
text = re.sub(r"ll", " will", text)
text = re.sub(r"t", " not", text)
text = re.sub(r"n't", " not", text)
text = re.sub(r"re", " are", text)
text = re.sub(r"ve", " have", text)
text = re.sub(r"m", " am", text)

tokens = text.split()
# Stem and tokenize
#ps = PorterStemmer()
#text = [ps.stem(word.strip(string.punctuation)) for word in text.split()]
# Return tokens
return [i for i in tokens if i != '']
```

```
In [9]: def tokenize_df(df):
# Progress bar for text cleaning and tokenization
#tqdm.pandas(desc='Loading Database')
# Cleans and tokenizes
df['TOKENIZED'] = [clean_and_tokenize(text) for text in df['DATA']]

# Creates a list of tuples mapping each token to a tweet id
normalized_toks = df.TOKENIZED.tolist()
IDs = df.ID.tolist()
ds = list(zip(normalized_toks, IDs))
return ds
```

```
In [10]: ds = tokenize_df(mydata)
```

```
In [11]: #ds[:3]
```

```
In [12]: from collections import defaultdict

def create_idx(ids):
inverted_index = defaultdict(list)
for words, doc in ds:
    for word in words:
```

```

        inverted_index[word].append(doc)
    vocab = sorted(inverted_index.keys())
    return vocab, inverted_index

```

```

In [13]: vocab, idx = create_idx(ds)
         #print(vocab)

```

```

In [14]: from collections import defaultdict
         from typing import *

def permutator(vocab):
    #we create a dictionary to store terms of
    #vocabulary as keys and their permutations as values
    permutations = defaultdict(list)
    #we iterate over each term in our vocabulary
    for term in vocab:
        #we make sure to exclude punctuations and numbers --> maybe change?
        if term.isalpha() == True:
            #in case the term is one letter or multiple same letters we do not
            #need permutations
            if len(term) == 1 or term == term[0] * len(term):
                permutations[term].append(term)
            #in all other cases we first add the end of string symbol $ to then
            #perform permutations
            else:
                termdollar = term + "$"

                #now we iterate over each character in each term
                for char in range(len(termdollar)):
                    #and we store iteratively the first char followed by next in the term -1
                    allpermut = termdollar[char:] + termdollar[:char]
                    #we append to our dictionary all the permutations of a specific term (key) as list (value)
                    permutations[term].append(allpermut)

    return permutations

# this function transforms the query input of user by adding $ symbol and the kleene star
# at the end of the word
def query_convertor(querywd):
    size = len(querywd)
    # we add the end of string symbol "$" to the query word
    querywd = querywd + "$"
    # we find where the index of the kleene star is
    index = querywd.find("*")
    # we transform the query word in the shape of what originally comes after the kleene start
    # + what originally comes before, followed by the start itself (end of string)
    querywd = querywd[index+1:size+1] + querywd[0:index]
    return querywd

# this function takes a queryword and a vocabulary as input
# this function converts the query word according to the converter and
# as output we get a list of terms of the vocabulary whose permutation match with the query
def queryvocabmatcher(queryword, vocabu):
    queryword = query_convertor(queryword)
    #container to store where permutation or bigram of a term matches the query
    matches = []

    # first we unwrap the terms and their lists of permutations or bigrams
    for key, val in vocabu.items():
        # from the permutations or bigrams we check whether there is some whose
        # start matches the query word
        for word in val:
            if word.startswith(queryword):
                matches.append(key)
    return matches

# we instantiate a class TreeNode so that we can recursively store and
# retrieve the different permutations of each term in our vocabulary. We create such
# a tree to store more efficiently
class TreeNode():
    def __init__(self) -> None:
        # in the children attribute we will keep track of all children nodes
        self.children : Dict[str, TreeNode] = defaultdict(TreeNode)
        # here we'll store the terms matching the query
        self.terms = []

    @staticmethod
    def construct_from_vocab( words: List[str]): # -> TreeNode:
        # we initialize our Tree object by setting the root node
        root = TreeNode()
        # we create all the permutations
        permuated_dict = permutator(words)
        for term, permutations in permuated_dict.items():
            for permutation in permutations:
                root.insert(permutation, term)
        return root

```

```

def insert(self, permutation: str, term: str):
    # each time we reach the end of the bigrams we add the final matching term
    # to the list of all terms matching
    if len(permutation) == 0:
        self.terms.append(term)
    # whereas if we (still) have permutations we construct a new branch
    else:
        # e.g. dog$
        ch = permutation[0] # e.g. "d"
        tail = permutation[1:] # e.g. "og$"
        child = self.children[ch]
        child.insert(tail, term)

def query(self, query: str):
    # depending on the query we will return the nodes
    query = query_convertor(query)
    return self.find_node_with_prefix(query)

def find_node_with_prefix(self, query: str): # $do
    # we want to find recursively the subtree where the edges lead
    # to the subtree: the word on the edges leading to subtree is query
    if len(query) == 0:
        return self.collect_match()
    else:
        # if the length is not zero we descend recursively
        ch = query[0] # d
        tail = query[1:] # og$
        child = self.children[ch]
        return child.find_node_with_prefix(tail)

def collect_match(self):
    # we visit all the child nodes recursively in subtree
    # and collect all terms stored in subtree
    result = []
    result += self.terms
    for child in self.children.values():
        result += child.collect_match()
    return result

```

```

In [15]: treeperm = TreeNode.construct_from_vocab(vocab)
         permutations = permutator(vocab)

```

```

In [ ]: #print(treeperm.query("zeit*"))

```

```

In [3]: #setresults = set(queryvocabmatcher("zeit*", permutations)) #print(len(setresults))
         queryoneperm = queryvocabmatcher("zeit*", permutations)
         querytwoperm = queryvocabmatcher("*zeit", permutations)
         querythreeperm = queryvocabmatcher("ze*st", permutations)

```

```

-----
NameError                                Traceback (most recent call last)
Cell In [3], line 2
      1 #setresults = set(queryvocabmatcher("zeit*", permutations)) #print(len(setresults))
----> 2 queryoneperm = queryvocabmatcher("zeit*", permutations)
      3 querytwoperm = queryvocabmatcher("*zeit", permutations)
      4 querythreeperm = queryvocabmatcher("ze*st", permutations)

NameError: name 'permutations' is not defined

```

```

In [19]: # print("This is result for query zeit* :", queryoneperm)
         # print("This is result for query *zeit :", querytwoperm)
         # print("This is result for query ze*st :", querythreeperm)

```

```

In [20]: queryonetreeperm = treeperm.query("zeit*")
         querytwotreeperm = treeperm.query("*zeit")
         querythreetreeperm = treeperm.query("ze*st")

```

```

In [21]: # print("This is result for query zeit* :", queryonetreeperm)
         # print("This is result for query *zeit :", querytwotreeperm)
         # print("This is result for query ze*st :", querythreetreeperm)

```

```

In [22]: # here bigram index
         def bigramidx(vocab):
             #we create a dictionary to store terms of vocabulary as keys and the bigrams as values

```

```

bigrams = defaultdict(list)
#we iterate over each term in our vocabulary
for term in vocab:
    #we make sure to exclude punctuations and numbers
    if term.isalpha() == True:
        termdollar = "$" + term + "$"
        for i in range(len(termdollar)-1):
            termbigrs = termdollar[i:i+2]
            #termsbigrs = nltk.ngrams(termdollar, 2) #nltk version
            bigrams[term].append(termbigrs)

return bigrams

# this changes from the permut convertor because before returning
# we split into bigrams
def query_convertorbigr(querywd):
    size = len(querywd)
    # we add the end of string symbol "$" to the query word
    querywd = querywd + "$"
    # we find where the index of the kleene star is
    index = querywd.find("*")
    # we transform the query word in the shape of what originally comes after the kleene start
    # + what originally comes before, followed by the start itself (end of string)
    querywd = querywd[index+1:size+1] + querywd[0:index]
    bigqueryres = []
    for i in range(len(querywd)-1):
        bigqueryres.append(querywd[i:i+2])

return bigqueryres

```

In [23]: *# this tree is similar to the one for the permuterm but we get a different query funct*
specific for queries where we want bigrams as output and then we will look for intersections

```

class TreeNode:
    def __init__(self) -> None:
        self.children: Dict[str, TreeNode] = defaultdict(TreeNode)
        self.terms = []

    @staticmethod
    def construct_from_bigramidx(words: List[str]): # -> TreeNode:
        # we initialize our Tree object by setting the root node
        root = TreeNode()
        # we create all the permutations
        bigrm_dict = bigramidx(words)
        for term, bigrams in bigrm_dict.items():
            # we add a bigram to a new root
            for bigram in bigrams:
                root.insert(bigram, term)
        return root

    def insert(self, bigram: str, term: str):

        # each time we reach the end of the bigrams we add the final matching term
        # to the list of all terms matching
        if len(bigram) == 0:
            self.terms.append(term)
        else:
            ch = bigram[0]
            tail = bigram[1:]
            child = self.children[ch]
            child.insert(tail, term)

    def querybigram(self, query: str):
        query = query_convertorbigr(query)
        #print(query)
        q_results = []
        for bigram in query:
            r = self.find_node_with_prefix(bigram)
            r = set(r)
            #print(r)
            q_results.append(r)

        result = set.intersection(*q_results)
        return result

    def find_node_with_prefix(self, query: str): # do
        if len(query) == 0:
            return self.collect_match()
        else:
            ch = query[0] # d
            tail = query[1:] # o
            return self.children[ch].find_node_with_prefix(tail)

    def collect_match(self):
        result = []
        result += self.terms

```



```
for child in self.children.values():
    result += child.collect_match()
return result
```

```
In [24]: treebigr = TreeNode.construct_from_bigramidx(vocab)
bigrms = bigramidx(vocab)
```

```
In [25]: print(treebigr.querybigram("zeit*"))
```

```
{'zeitgleich', 'zeit', 'zeitgeschichtlichen', 'zeitlupe', 'zeitumstellung', 'zeitgenossen', 'zeitalter', 'zeitnah',
'zeitpkt', 'zeitungscallcenter', 'zeitzeugen', 'zeitintensiv', 'zeitgenssische', 'zeitlang', 'zeitstein', 'zeit-
zonen', 'zeitrechnung', 'zeitschinder', 'zeitungen', 'zeitmanagement', 'zeitroboter', 'zeitarbeit', 'zerreit', 'zei-
eitschrift', 'zeitung', 'zerisssenheit', 'zeitbombe', 'zeitschleife', 'zeitgem', 'zeitgeme', 'zeitrhythmus', 'zei-
traffertrack', 'zeitgeist', 'zwischenzeitlich', 'zeitungsbeitrgen', 'zeitlich', 'zeitweilig', 'zeitverschwendung',
'zeitig', 'zeitschriften', 'zeitzone', 'zwischenzeit', 'zeitraum', 'zugzeit', 'zeitgrnden', 'zeitlos', 'zeitges-
chichte', 'zeitungspapier', 'zeitweise', 'zeitschaltung', 'zurzeit', 'zeitschriftenportals', 'zeitungskommentare',
'zeitpunkt', 'zeitreisender', 'zeitdruck', 'zeiten', 'zeitpunk', 'zeitgmsse', 'zeitvertreib', 'zeitalters', 'zei-
eitliche', 'zeitwchter', 'zwischenmahlzeit', 'zeitlinie', 'zeitmaschine', 'zeitgenssischen', 'zeittiming', 'zeiti-
gt', 'zeitgemes', 'zeitspanne', 'zeitenwende', 'zeitlinieebem', 'zeitzeuge', 'zeiterl', 'zeitnaahr', 'zeitiger',
'zeitgenssisches', 'zeitnahe', 'zeitfenster', 'zeitlichen', 'zeitgeistlicher'}
```

```
In [27]: print(treebigr.querybigram("*zeit"))
```

```
{'realschulzeit', 'regenzeit', 'teilzeit', 'studienzeit', 'zeit', 'elternzeit', 'prophezeit', 'spielzeit', 'nachk-
riegszeit', 'zeitpkt', 'diamanthochzeit', 'irrsinnszeit', 'hochzeitsangst', 'passierenzeit', 'nachspielzeit', 'dr-
ogenzeit', 'konzernzeitschrift', 'teenagerzeit', 'redezeit', 'tageszeit', 'frhzeit', 'abizeit', 'zeitmanagement',
'pollenzeit', 'wartelistenzeit', 'infostandzeit', 'teeniezeit', 'winterzeit', 'praktikumszeit', 'fastenzeit', 'zei-
tarbeit', 'zerreit', 'erntezeit', 'zeitschrift', 'zerisssenheit', 'regelstudienzeit', 'herzensangelegenheit', 'sc-
hlafenszeit', 'sendezeit', 'neuzeit', 'geschichtenzeit', 'zeitgeist', 'durchschnittsspielzeit', 'bronzezeit', 'ki-
nderzeit', 'vorlaufzeit', 'familienzeit', 'lebenszeit', 'grundzeit', 'wartezeit', 'halbzeit', 'derzeit', 'zwische-
nzeit', 'zugzeit', 'allzeit', 'vorlesungszeit', 'mahlzeit', 'berlebenszeit', 'sommerzeit', 'urzeit', 'lebzeit', '
horrorzeit', 'weihnachtszeit', 'jugendfreizeit', 'retentionszeit', 'amtszeit', 'traumhochzeit', 'zurzeit', 'halbw-
ertszeit', 'trauerfestspielzeit', 'hauptverkehrszeit', 'kaiserzeit', 'zeitpunkt', 'fahrtzeit', 'fliederzeit', 'vo-
llzeit', 'nazizeit', 'frhlingszeit', 'knuddelzeit', 'auszeit', 'arbeitszeit', 'tatzeit', 'langzeit', 'mittagszeit',
'allergiezeit', 'lokalzeit', 'freizeit', 'besuchszeit', 'bungszeit', 'wahrheitskonzept', 'karnevalszeit', 'zwi-
schenmahlzeit', 'familienfreizeit', 'echtzeit', 'jederzeit', 'verbotszeit', 'pausenzeit', 'piratenzeit', 'seinerz-
eit', 'zeitigt', 'steizeit', 'uhrzeit', 'gartenzeit', 'langezeit', 'schulzeit', 'inkubationszeit', 'bedenkzeit',
'pollenflugzeit', 'aktivierungszeit', 'langzeitstudent', 'steinzeit', 'jahreszeit', 'faschingszeit', 'doppelhochz-
eit', 'osterzeit', 'fuckboyzeit', 'rekordzeit', 'eiszeit', 'teilzeitarbeit', 'hochzeit', 'haltbarkeitszeit', 'vor-
zeit'}
```

```
In [26]: print(treebigr.querybigram("ze*st"))
```

```
{'zeichnest', 'zeigst', 'zerfrisst', 'zeitgeist', 'zerstrt', 'zerstckelt', 'zerstrungswut', 'zeigtest', 'zerstrst',
'}
```

In []:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js